1. Basic Git Commands (For managing repositories)

- `git init`: Initializes a new Git repository in the current directory.
- `git clone <repo_url>`: Clones a repository from a remote URL to your local machine.
- `git status`: Shows the working directory status (which files are modified, staged, etc.).
- `git add` : Stages a file or files to be committed.
- `git commit -m "message"`: Commits staged changes to the local repository with a message.
- `git push`: Pushes committed changes to the remote repository.
- `git pull`: Fetches and merges changes from the remote repository to your local working directory.
- `git fetch`: Downloads objects and refs from another repository (without merging).
- `git merge` : Merges changes from the specified branch into the current branch.
- `git log`: Shows the commit history of the repository.
- `git diff`: Shows the differences between your working directory and the index (staged changes).

2. Branching Commands (Managing branches)

- `git branch`: Lists all branches in your repository.
- `git branch <branch_name>`: Creates a new branch.
- `git checkout <branch_name>`: Switches to an existing branch.
- `git checkout -b <branch_name>`: Creates and switches to a new branch in one step.
- `git merge <branch_name>`: Merges changes from the specified branch into the current branch.

- `git rebase <branch_name>`: Applies changes from one branch onto another branch (instead of merging).
- `git branch -d <branch_name>`: Deletes the specified branch (locally).
- `git branch -D <branch_name>`: Forcefully deletes a branch (locally).
- `git remote set-head origin <branch_name>`: Sets the default branch for the remote repository.

3. Remote Commands (Managing remotes and syncing with the cloud)

- `git remote -v`: Shows the remote repository URLs.
- `git remote add` : Adds a new remote repository.
- `git remote remove` : Removes a remote repository.
- `git push origin <branch_name>`: Pushes a branch to a remote repository.
- `git push -u origin <branch_name>`: Pushes a branch to a remote repository and sets the upstream for the branch.
- `git pull origin <branch_name>`: Fetches and merges a branch from the remote repository.
- `git fetch origin`: Downloads objects and refs from the remote repository.
- `git remote show origin`: Displays detailed information about a remote repository.

4. Staging and Committing (Managing changes)

- `git add .` : Stages all modified files in the working directory.
- `git add -A`: Stages all modified files (including file deletions).
- `git reset` : Unstages a file (reverts it to the working directory).

- `git commit --amend`: Modifies the last commit (can change the commit message or add changes).
- `git commit --all`: Automatically stages tracked files and commits them.
- `git commit -a`: Stages and commits all modified files.
- `git commit --no-verify`: Skips pre-commit hooks while committing.
- `git reset --soft HEAD~1`: Moves the HEAD pointer back one commit but leaves your changes staged.
- `git reset --hard HEAD~1`: Moves the HEAD pointer back one commit and discards changes in the working directory.

5. Viewing and Comparing Changes (Inspecting changes)

- `git diff`: Shows changes between your working directory and the index (unstaged changes).
- `git diff --staged`: Shows changes between the index (staged files) and the last commit.
- `git diff <commit_id>`: Shows the differences between your working directory and a specific commit.
- `git log`: Displays the commit history.
- `git log --oneline`: Displays the commit history in a simplified one-line format.
- `git log --graph`: Displays the commit history as a graph.
- `git log --author="name"`: Filters commits by a specific author.
- `git log --since="2 weeks ago"`: Filters commits made after a specific date.
- `git blame` : Shows line-by-line annotations for a file, telling who last modified each line.
- `git show <commit_id>`: Shows details of a specific commit (including diff and metadata).

- `git show <commit_id>:<file_path>`: Shows a specific file at a particular commit.

6. Reverting and Resetting (Undoing changes)

- `git reset <commit_id>`: Moves HEAD to a specific commit, leaving your working directory intact (can be used with --soft, --mixed, or --hard). by default soft commit.
- `git reset --hard <commit_id>`: Resets your working directory, index, and HEAD to a specific commit (all changes will be lost).
- `git reset --soft <commit_id>`: Resets only the HEAD to a specific commit, leaving staged changes.
- `git reset --mixed <commit_id>`: Resets the HEAD and index to a specific commit but keeps the working directory unchanged.
- `git revert <commit_id>`: Creates a new commit that undoes the changes of the specified commit.
- `git restore` : Restores the file(s) in the working directory from the index or a commit.
- `git restore --staged` : Removes a file from the staging area.
- `git clean -f`: Removes untracked files in the working directory.
- `git clean -fd`: Removes untracked files and directories.

7. Stashing (Temporarily saving changes)

- `git stash`: Stashes the changes in your working directory (both staged and unstaged).
- `git stash list`: Lists all the stashes.

- `git stash pop`: Applies the latest stash and removes it from the stash list.
- `git stash apply`: Applies a stash without removing it from the list.
- `git stash drop`: Removes a specific stash from the stash list.
- `git stash clear`: Clears all stashes.
- `git stash save "message"`: Stashes changes with an optional message for identification.
- `git stash branch <branch_name>`: Creates a new branch from the stash and applies it.

8. Tags (Managing versions)

- `git tag`: Lists all tags in the repository.
- `git tag <tag_name>`: Creates a new tag at the current commit.
- `git tag -a <tag_name> -m "message"`: Creates an annotated tag with a message.
- `git tag -d <tag_name>`: Deletes a tag locally.
- `git push origin <tag_name>`: Pushes a specific tag to the remote repository.
- `git push origin --tags`: Pushes all tags to the remote repository.
- `git fetch --tags`: Fetches all tags from the remote repository.

9. Git Config and Info (Configuration and repository information)

- `git config --global user.name "Your Name"`: Sets your global username for commits.

- `git config --global user.email "youremail@example.com"`: Sets your global email for commits.
- `git config --list`: Lists all configuration settings.
- `git config` : Sets a specific configuration option.
- `git config --global core.editor` : Sets the default text editor for Git.
- `git config --global color.ui true`: Enables colored output in Git.
- `git config --global alias.st status`: Creates a custom Git alias (e.g., git st for git status).

10. Git Aliases (Custom commands)

- `git config --global alias.<alias_name> <actual_command>`: Creates a new Git alias for an existing command.
  Example: git config --global alias.co checkout

11. Git Hooks (Automated scripts triggered by Git actions)

- `git init`: Creates a .git/hooks directory, where hook scripts are stored.
- `git commit-msg`: A hook that runs before a commit message is saved (you can use this to enforce message formats).
- `git pre-commit`: A hook that runs before a commit is made (can be used for checks like linting).
- `git post-commit`: A hook that runs after a commit is made.

12. Git Submodules (Managing external repositories within a project)

- `git submodule add <repo_url>` : Adds a new submodule to the repository (downloads an external repo).
- `git submodule init`: Initializes the submodules configured in the repository (after cloning).
- `git submodule update`: Updates the submodules to the commit specified in the superproject.
- `git submodule status`: Displays the current commit of the submodule.
- `git submodule deinit` : Removes a submodule from the working directory.
- `git submodule update --remote`: Updates the submodule to the latest commit from the remote repository.
- `git submodule foreach` : Runs a command in each submodule.

13. Git Workflow Commands (Working with others and managing collaboration)

- `git pull --rebase`: Fetches and applies remote changes, rebasing your local commits on top of them.
- `git rebase -i <commit_id>`: Starts an interactive rebase to modify commit history (e.g., squash, reword commits).
- `git rebase --continue`: Continues the rebase after resolving conflicts.
- `git rebase --abort`: Aborts the rebase process and restores the original state.
- `git merge --no-ff` : Merges a branch with a "no fast-forward" option to ensure a merge commit is created even when possible to do a fast-forward merge.

- `git pull --no-commit`: Pulls changes but doesn't automatically commit them.
- `git push --force-with-lease`: Forces push but checks if the remote branch has been updated (safer than git push --force).
- `git push --force`: Forces a push to the remote branch, potentially overwriting changes.
- `git cherry-pick <commit_id>`: Applies the changes from a specific commit onto the current branch.

14. Git Merge Strategies (Handling merge conflicts and strategies)

- `git merge --strategy=ours <branch_name>`: Resolves merge conflicts by favoring the current branch's changes.
- `git merge --strategy=theirs <branch_name>`: Resolves merge conflicts by favoring the other branch's changes.
- `git merge --abort`: Aborts the merge process if there are conflicts and restores the working directory.

15. Git bisect (Finding the commit that introduced a bug)

- `git bisect start`: Starts a binary search for the commit that introduced a bug.
- `git bisect bad`: Marks the current commit as "bad" (where the bug is present).
- `git bisect good <commit_id>`: Marks a commit as "good" (where the bug was not present).
- `git bisect reset`: Ends the bisect process and restores the repository to the state it was before starting the bisect.
- `git bisect log`: Displays the log of the bisect process.

16. Git Hooks for Automation (Pre-commit and post-commit hooks)

- `git commit-msg`: A hook script that runs before the commit message is finalized (you can use this to enforce rules, like requiring certain keywords).
- `git pre-commit`: A hook that can be used to run checks on files before the commit is finalized, such as linters or tests.
- `git post-commit`: A hook that is triggered after a commit has been completed, useful for triggering actions like notifications or build processes.
- `git pre-push`: A hook that runs before a push to the remote repository is initiated.
- `git post-merge`: A hook that runs after a merge, ideal for cleanup or setup tasks.

17. Git Clean and Prune (Cleaning up untracked files and garbage collection)

- `git clean -n`: Shows which untracked files would be removed, but doesn't actually delete them.
- `git clean -f`: Removes untracked files from the working directory.
- `git clean -fd`: Removes untracked files and directories.
- `git gc`: Runs garbage collection, cleaning up unnecessary files and optimizing the repository.
- `git prune`: Removes objects that are no longer needed (generally used for cleaning up in repositories with lots of history).

18. Git Archive (Creating a snapshot of the repository)

- `git archive --format=tar --output=<output_file>.tar <branch_name>`: Creates a tarball archive of the repository at a specific branch.
- `git archive --format=zip --output=<output_file>.zip <branch_name>`: Creates a zip archive of the repository at a specific branch.

19. Git Blame and Annotate (Tracing the history of file content)

- `git blame <file_path>`: Shows who last modified each line of a file and when.
- `git annotate <file_path>`: An alias for git blame, which displays commit information for each line of a file.

20. Git Reflog (Tracking changes in the HEAD reference)

- `git reflog`: Displays a log of all movements of the HEAD pointer (useful for tracking changes like reset and rebase operations).
- `git reflog show`: Displays a detailed view of HEAD and the state of the repository.
- `git reflog expire --expire=now --all`: Removes all reflog entries that have expired.
- `git reflog delete <ref_id>`: Deletes a specific reflog entry.

Git for DevOps

1. Git for Continuous Integration/Continuous Deployment (CI/CD)

- **git fetch --all**: Fetches all branches from all remotes, often used before triggering a build or deployment in a CI/CD pipeline.
- **git merge origin/**: Merges the latest changes from a remote branch into the local branch to ensure that the CI/CD pipeline tests the most up-to-date version.
- **git push --force-with-lease**: When integrated with CI/CD, this command can be used to push changes forcefully while ensuring that the remote branch hasn't changed unexpectedly (prevents accidental overwrites in collaborative environments).
- **git rebase origin/**: Rebasing is often part of CI/CD to ensure your branch is up to date with the base branch before merging or deploying.
- **git tag <version_number>**: Tagging is essential in DevOps for versioning releases, allowing the CI/CD system to deploy specific versions of the code (e.g., for staging or production).

2. Git for Managing Infrastructure as Code (IaC)

DevOps teams often use Git to store **Infrastructure as Code** (e.g., Terraform, Ansible, etc.). Here's how Git commands come into play:

- **git clone <IaC_repo_url>**: Clone repositories that contain configuration files or infrastructure code.
- **git checkout** : Switch to a branch that contains a specific environment's infrastructure code (e.g., production, staging).
- **git pull origin <branch_name>**: Pull the latest changes to infrastructure code before applying configurations or deploying.
- **git submodule**: Git submodules are commonly used to manage shared code or dependencies across multiple infrastructure repositories.

3. Git Hooks for DevOps Automation

Git hooks can be a key part of automating tasks within the DevOps pipeline, such as checking code quality or triggering build/deployment processes.

- **git pre-commit**: Automatically run tests or linters on code before it's committed. This can ensure that the code meets quality standards before being pushed to the repository.
- **git post-merge**: Automatically trigger deployment processes or run tests after a merge happens, ensuring that the newly merged code is stable.
- **git pre-push**: Can be used to run a script to test whether the pushed code will break something in production, such as triggering a CI pipeline that runs integration tests before pushing to a remote.
- **git commit-msg**: Enforce commit message conventions, often integrated into CI/CD pipelines to ensure semantic versioning or the correct Jira ticket references in commit messages.

4. Git for Feature Flag Management

DevOps teams use **feature flags** to control feature deployment. Git plays an important role in this workflow by helping track which features are being developed and merged.

- **git branch <feature_flag_name>**: Create a branch dedicated to developing a specific feature, often linked to a feature flag.
- **git checkout -b <feature_flag_branch>**: Create a new branch to work on a feature behind a feature flag. The flag can be turned on/off in the code depending on the deployment environment.
- **git merge <feature_flag_branch>**: Merge a completed feature flag branch into the main codebase, which can be toggled on or off during deployment via configuration.

5. Git for Multi-Environment Deployments

Git repositories often contain multiple branches corresponding to different environments (e.g., development, staging, production). Here's how Git fits in with deploying to multiple environments:

- **git checkout <environment_branch>**: Switch to the branch corresponding to the environment you are deploying to.
- **git pull origin <environment_branch>**: Make sure your local branch is up to date with the latest code before deploying to a specific environment.
- **git merge --no-ff <feature_branch>**: Merge feature branches into specific environment branches with a merge commit to preserve the history.
- **git tag <release_version>**: Tag releases when promoting code to staging or production environments. Versioning tags allow for consistent, repeatable deployments across environments.

6. Git and Deployment Automation

In DevOps, deployment automation is often triggered directly from Git repositories. Some examples:

- **git push origin** : This triggers CI/CD pipelines that automatically deploy code to specific environments.
- **git tag** : Trigger specific deployment pipelines (e.g., deploy-prod, deploy-staging) to automatically deploy code to staging or production when a tag is created.
- **git reset --hard**: In deployment automation scenarios, this command is used to reset the working directory to a known state (e.g., the latest production commit) before deploying.

7. GitOps Workflow

GitOps is an approach to Continuous Deployment where Git repositories are the source of truth for infrastructure and application configuration.

- **git push** : In GitOps, pushing changes to a Git repository automatically triggers deployment pipelines. For instance, when code or configuration files are pushed to Git, the repository can trigger an automation system (like ArgoCD or Flux) to apply those changes to a Kubernetes cluster or other environments.
- **git clone <repo_url>**: Developers clone GitOps repositories to inspect or modify deployment configurations stored as code.

8. Advanced Git for Managing CI/CD Pipelines

Many CI/CD systems, such as Jenkins, GitLab CI, and CircleCI, integrate directly with Git to trigger build and deployment pipelines. Some useful Git-related practices:

- **git push --tags**: Trigger a CI/CD pipeline that specifically looks for tags to start a deployment (e.g., deploying to production when a v1.0.0 tag is created).
- **git fetch --tags**: Use this command in CI/CD pipelines to fetch the latest tags and ensure the correct version of the code is deployed.
- **git log --oneline --graph**: Useful in CI/CD pipeline scripts to visualize or check commit histories during pipeline execution, helping to verify which changes are deployed to which environments.
- **git diff <commit_id> <commit_id>**: In CI/CD, you might want to see the changes between commits to decide whether a particular build should be triggered (e.g., when testing, you might only want to run tests for certain changed files).

9. Git and Docker Integration

In DevOps, Git is frequently used alongside Docker for containerized applications. Here's how Git plays a role in the Docker workflow:

- **git clone <repo_url>**: Clone a repository that contains a Dockerfile and application code, which you can then use to build Docker images.

- **git checkout** : Switch to a branch where different Docker images or container configurations might be defined.
- **git pull origin <branch_name>**: Pull the latest code from the branch and rebuild Docker images to include the latest changes.
- **git tag** : Use Git tags to tag a version of the Docker image (e.g., v1.0.0), which can then be used for tagging Docker images when pushed to a container registry.

10. Git and Secrets Management

DevOps teams often use Git to manage code along with secret management tools for deployment, often using .gitignore to keep sensitive files like secrets out of version control.

- **git rm --cached** : Remove a file from version control, useful if secrets were accidentally committed.
- **git commit --amend**: If secrets were committed accidentally, this can be used to amend the last commit and remove them from history.
- **git rebase -i**: Used to rewrite history if secrets were committed in previous commits, ensuring that sensitive information is completely removed from Git history.

11. Git Rebase and Interactive Rebase (Managing commit history)

Rebase is essential in DevOps to clean up commit history before merging feature branches into the main branch, especially when maintaining a clean and linear Git history for CI/CD pipelines.

- **git rebase -i <commit_id>**: Starts an interactive rebase, allowing you to reorder, squash, or amend commits in a branch.
- **git rebase --onto** : Used to rebase a branch onto a new base commit. Useful when moving a feature branch or integrating changes that originated from a different base.

- **git rebase --skip**: Skips the current commit while rebasing, useful if there's a conflict or you don't want to include a specific commit in the rebase.
- **git rebase --continue**: Continues the rebase process after resolving any conflicts.

12. Git Merge Conflict Resolution (Handling merge issues)

Merge conflicts are common when multiple developers work on the same files or parts of the code. Git provides tools for resolving conflicts efficiently.

- **git mergetool**: Opens an external merge tool to help resolve conflicts. Can be configured to use tools like vimdiff, meld, kdiff3, etc.
- **git merge --abort**: Aborts the merge process if conflicts occur and restores the state before the merge.
- **git diff <branch_1> <branch_2>**: Shows the differences between two branches to help you identify and resolve conflicts before merging.
- **git status**: Shows which files are in conflict and need to be resolved manually.

13. Git Commit Signing (Ensuring commit authenticity)

In DevOps, ensuring the authenticity of code changes and that commits are made by authorized developers is important. Git supports signing commits and tags using GPG keys.

- **git config --global user.signingkey <key_id>**: Configures the GPG key to sign commits and tags.
- **git commit --gpg-sign**: Signs a commit with your GPG key.
- **git tag -s <tag_name>**: Creates a signed tag (useful for marking important release points).

- **git log --show-signature**: Displays the GPG signature of commits in the log to verify their authenticity.

14. Git for Continuous Delivery (Automated release management)

In DevOps, Git repositories are frequently tied to automated release pipelines. Here are a few practices related to release management.

- **git push --tags**: Pushes tags to the remote repository, triggering deployment processes in CI/CD pipelines. For example, pushing a release tag like v1.0.0 might trigger deployment to production.
- **git push origin <branch_name>:<remote_branch>**: Pushes a local branch to a remote branch, ensuring that your changes are reflected in the remote environment. Often used in CI/CD to sync code with the cloud.
- **git branch -m <old_name> <new_name>**: Renames a branch, which can be useful in DevOps workflows when a feature is moved to a new stage (e.g., from feature-xyz to staging).
- **git merge --no-ff <branch_name>**: Forces Git to always create a merge commit, even if it could be a fast-forward merge. This is useful for preserving the context of feature branches in release management.

15. Git for Rollbacks (Reverting to previous versions)

In DevOps, the ability to quickly roll back code changes to a stable state is critical, especially in production environments.

- **git revert <commit_id>**: Reverts the changes from a specific commit by creating a new commit that undoes the changes. This is often used when a production deployment causes issues.
- **git reset --hard <commit_id>**: Resets the repository to a previous commit and discards changes in the working directory. This can be used to roll back to a stable state.

- **git log --oneline**: Use this command to find the commit hash for the state you want to roll back to.
- **git checkout <commit_id>**: Checks out an older commit, allowing you to inspect the repository at that point in time or even deploy an older version of the app.

16. Git Submodules and Subtrees (Managing dependencies and libraries)

Git submodules and subtrees are used in DevOps to manage dependencies or external libraries in your repository.

- **git submodule update --recursive**: Updates all submodules, ensuring that all dependencies are in sync.
- **git submodule init**: Initializes submodules after cloning a repository that contains them.
- **git submodule add <repo_url>** : Adds a submodule to your repository. Submodules allow you to manage external dependencies as part of your Git repository.
- **git subtree add --prefix= <repo_url> <branch_name>**: Allows you to integrate an external repository into a subdirectory within your repository, useful when managing code that needs to be bundled with your application.

17. Git for Team Collaboration and Code Review (Facilitating teamwork)

Git plays a crucial role in facilitating collaboration between DevOps teams, particularly for code reviews, pull requests, and continuous integration.

- **git push origin <feature_branch>**: Pushes a feature branch to a remote, which can be used in collaboration tools (e.g., GitHub, GitLab, Bitbucket) for code review and pull requests.

- **git fetch origin <feature_branch>**: Fetches a feature branch from a remote repository so that team members can review it or test it locally.
- **git pull origin <feature_branch>**: Pulls the latest changes from a feature branch into your local repository, often used when testing code changes before merging.
- **git diff <branch_name>**: Compare different branches (e.g., comparing feature-xyz with main branch) before merging them.

18. Git and Containerized Environments (Kubernetes, Docker) (Managing deployments in containers)

In DevOps environments, Git is frequently used to manage the source code for applications that run in containers. Git commands can facilitate container image builds and deployment pipelines.

- **git clone <repo_url>**: Clones a repository containing code and configurations for building Docker containers or Kubernetes manifests.
- **git checkout** : Switches between branches to test different versions of a Dockerized application or configuration files.
- **git pull origin <branch_name>**: Pull the latest code from a Git repository before rebuilding Docker images or Kubernetes configurations.
- **git tag** : Tags releases to facilitate versioning of Docker images or Kubernetes deployments. For example, pushing a tag v1.0.0 may automatically trigger a CI/CD pipeline that builds and deploys a Docker image.
- **git submodule add <repo_url>** : Can be used to add a Git repository as a submodule, allowing you to include external configuration files, such as Dockerfiles or Helm charts, in your repository.

19. Git for Managing Build and Deployment Pipelines

Git is tightly integrated with CI/CD systems, where it triggers build and deployment processes based on repository changes. Here are key Git commands related to build automation:

- **git push origin** : Pushes your branch to the remote repository, triggering the build process in your CI/CD tool.
- **git tag <build_version>**: Tags a commit with a version number, which might trigger deployment pipelines for testing or production.
- **git log --oneline**: View a condensed log of the commit history to identify the changes associated with a specific build or deployment.
- **git push --force-with-lease**: Pushes changes to a branch with a safer force push, useful when you need to overwrite history in collaborative CI/CD workflows but without overwriting someone else's work.

20. Git for Documentation and Versioning (Managing configuration files and documentation)

For DevOps, Git repositories are often used to version configuration files, scripts, and documentation.

- **git add** : Add configuration or documentation files to the repository for versioning.
- **git commit -m "add/update documentation"**: Commit changes related to documentation, often used in documentation-driven development.
- **git diff**: Compare different versions of documentation files or configuration files, useful when tracking changes in infrastructure or environment files.