As a software engineer, working with Git is a fundamental part of daily tasks, whether you're working on solo projects or collaborating with teams. Below is a comprehensive list of common Git commands, along with brief explanations of their purpose:

1. Basic Setup and Configuration

- **`git config --global user.name "Your Name"`**
  Set the name used for commits.

- **`git config --global user.email "youremail@example.com"`**
  Set the email used for commits.

- **`git config --list`**
  List all Git configuration settings.

- **`git config --global core.editor "editor-name"`**
  Set the default text editor for Git (e.g., `vim`, `nano`, `code`).

2. Repository Setup and Cloning

- **`git init`**
  Initialize a new Git repository in the current directory.

- **`git clone <repository-url>`**
  Clone an existing Git repository from a remote server to your local machine.

3. Branching and Merging

- **`git branch`**
  List all branches in the repository, and highlight the current branch.

- **`git branch <branch-name>`**
  Create a new branch.

- **`git checkout <branch-name>`**
  Switch to a different branch.

- **`git checkout -b <branch-name>`**
  Create and switch to a new branch in one command.

- **`git merge <branch-name>`**
  Merge changes from another branch into the current branch.

- **`git branch -d <branch-name>`**
  Delete a local branch (if it has been merged).

- **`git branch -D <branch-name>`**
  Force delete a local branch, even if it hasn't been merged.

4. Staging and Committing Changes

- **`git status`**
  Show the current status of the working directory, including changes that are staged, modified, or untracked.

- **`git add <file>`**
  Stage a specific file for commit.

- **`git add .`**
  Stage all modified files (including new files).

- **`git commit -m "Commit message"`**
  Commit staged changes with a message.

- **`git commit -a -m "Commit message"`**
  Commit all changes (including modified and deleted files) without needing to stage them first.

- **`git commit --amend`**
  Amend the most recent commit (e.g., to correct a commit message or add forgotten changes).

5. Viewing Commit History

- **`git log`**
  View the commit history.

- **`git log --oneline`**
  View a condensed, one-line version of the commit history.

- **`git log --graph --oneline`**
  View the commit history with a visual graph of the branch structure.

- **`git show <commit-hash>`**
  Show detailed information about a specific commit.

6. Undoing Changes

- **`git checkout -- <file>`**
  Discard changes in a specific file (reset to the last committed version).

- **`git restore <file>`**
  Similar to `git checkout`, but a safer way to discard changes from the working directory.

- **`git reset <file>`**
  Unstage a file (move it from the staging area back to the working directory).

- **`git reset --hard`**
  Reset the entire working directory and staging area to the last commit (WARNING: This will discard all uncommitted changes).

- **`git revert <commit-hash>`**
  Create a new commit that undoes the changes from a specific commit, useful for "undoing" a commit in a public history.

- **`git reset <commit-hash>`**
  Move the current branch pointer to a previous commit. This can be used with `--hard`, `--soft`, or `--mixed` to affect the working directory, staging area, or both.

7. Remote Repositories

- **`git remote -v`**
  Show the list of remote repositories associated with your local repository.

- **`git remote add <remote-name> <repository-url>`**
  Add a new remote repository.

- **`git fetch`**
  Download updates from the remote repository, but don't apply them to your working directory (fetches changes without merging).

- **`git pull`**
  Fetch and merge changes from the remote repository into your current branch.

- **`git push`**
  Push your local commits to a remote repository.

- **`git push origin <branch-name>`**
  Push your local branch to the remote repository.

- **`git push --force`**
  Force push changes (use with caution as it overwrites remote history).

8. Tagging

- **`git tag <tag-name>`**
  Create a lightweight tag for a commit.

- **`git tag -a <tag-name> -m "Message"`**
  Create an annotated tag (with a message).

- **`git tag`**
  List all tags in the repository.

- **`git push origin <tag-name>`**
  Push a specific tag to the remote repository.

- **`git push --tags`**
  Push all tags to the remote repository.

9. Collaborative Work and Reviewing Changes

- **`git diff`**
  Show differences between the working directory and the index (staged changes).

- **`git diff <commit1> <commit2>`**
  Compare two commits.

- **`git diff --staged`**
  Show differences between the staging area and the last commit.

- **`git log --stat`**
  Show commit history with a summary of changes made (number of lines added/removed).

- **`git blame <file>`**
  Show the last commit that modified each line of a file.

- **`git cherry-pick <commit-hash>`**
  Apply the changes from a specific commit onto the current branch (useful for applying specific fixes).

10. Working with Submodules

- **`git submodule add <repository-url> <path>`**
  Add a submodule to the project (embedding another repository inside your project).

- **`git submodule update`**
  Initialize, fetch, and checkout submodule content.

- **`git submodule status`**
  Display the status of the submodule(s).

11. Git Hooks

- Git hooks are scripts that run automatically on certain Git events. These aren't actual commands but are useful for automating tasks like running tests or checking commit messages before commits are made. Common hooks include:
  - **`pre-commit`**
  - **`commit-msg`**
  - **`pre-push`**

12. Advanced Features

- **`git reflog`**
  Show the history of changes to the tip of branches and the HEAD. This can be useful for recovering lost commits.

- **`git bisect`**
  A binary search tool to help identify which commit introduced a bug.

- **`git merge --no-ff`**
  Merge branches without fast-forwarding, preserving the commit history of the merged branch.

- **`git pull --rebase`**
  Fetch and rebase your changes on top of the current remote branch.

Usage Tips for Efficiency:

- **`git stash`**
  Save your uncommitted changes temporarily and revert your working directory to the last commit.

- **`git stash pop`**
  Retrieve the most recent stash and apply it to the working directory.

- **`git stash list`**
  List all stashes.

Git is a powerful version control tool with a lot of depth. Depending on the complexity of the project you're working on, you may only use a subset of these commands regularly. However, it's helpful to be familiar with most of them, as they can make collaboration, troubleshooting, and managing code much more efficient.