# The Definitive Guide to Git: From Core Concepts to Command Mastery

## Part 1: The Foundations of Git: A Conceptual Framework

Before mastering the extensive list of Git commands, it is essential to build a solid conceptual framework of how Git operates. Understanding the "why" behind Git's design makes the "how" of its commands intuitive and predictable. This section establishes the fundamental mental model required for effective version control.

### 1.1 Introduction to Version Control with Git

A Version Control System (VCS) is a tool that helps manage changes to a collection of files over time. It solves critical problems in software development and other collaborative work by providing a structured way to track history, coordinate work among multiple people, and safely explore new ideas without disrupting the main project.

Git is a specific type of VCS known as a **Distributed Version Control System (DVCS)**. In older, centralized models (like Subversion), a single central server contains the entire project history, and developers "check out" files to work on them. In contrast, Git's distributed model means that every developer's local copy of the project is a complete repository with the full history. This architecture provides significant advantages in speed, as most operations are performed locally without network latency, and in resilience, as the project's history is duplicated across multiple machines.

It is crucial to distinguish between Git, the command-line tool, and platforms like GitHub, GitLab, and Bitbucket. Git is the open-source software that performs the version control on a local machine.GitHub and similar services are web-based platforms that host Git repositories, providing a centralized location for collaboration, code review, and project management. These platforms are referred to in Git terminology as "remotes." While you do not need a remote to use Git, they are indispensable for sharing code and working in a team.

### 1.2 The Three Trees: Git's Core Architecture

To use Git effectively, one must understand its core architecture, which is often described as "the three trees." These are not literal trees in a data structure sense but are three distinct areas where your project's files reside during the version control process.

1. **The Working Directory (or Working Tree):** This is the most straightforward concept. The working directory is the set of files and folders currently on your local filesystem that you can see and edit with your standard tools. It represents a single checkout of one version of the project, pulled from the repository's database.
2. **The Staging Area (or Index):** The staging area is an intermediate space that sits between your working directory and your repository history. It is a file, typically located in the hidden .git directory, that stores information about what will go into your next commit.When you run the  
   git add command, you are not moving the file itself, but rather taking a snapshot of its current state and adding it to the staging area. This allows you to selectively prepare changes for a commit, carefully crafting a snapshot before it becomes a permanent part of the project's history.
3. **The Repository (or Git Directory):** The repository is the heart of your Git project. When you initialize a repository with git init, Git creates a hidden subdirectory named .git.This directory contains all the metadata and the object database for your project. Every version of every file, every commit, every branch, and every tag is stored here.2 This is the most important part of Git; it is the self-contained history that is copied when you execute git clone.

The design of the staging area is not merely a technical buffer; it is a feature that fundamentally encourages and enables best practices in version control. It decouples the act of saving files in the working directory from the act of recording historical snapshots in the repository. This separation provides a "commit construction zone" where a developer can meticulously assemble the next commit. For instance, a developer might make several unrelated changes in their working directory—fixing a typo, refactoring a function, and adding a new feature. Without a staging area, they might be forced to record all these changes in a single, messy commit. With the staging area, they can use commands like git add to stage the typo fix and commit it with a clear message, then stage the refactoring and commit it separately, and finally stage the new feature for a third, distinct commit. This process transforms the project history from a simple logbook of file saves into a well-curated, logical narrative of the project's evolution, which provides immense value for code review, debugging, and long-term maintainability.

### 1.3 The Lifecycle of a File: Tracking States

Building on the concept of the three trees, every file in a Git repository exists in one of three main states, which correspond to its location in the workflow :

1. **Modified:** A file is in a *modified* state when it has been changed in the working directory, but those changes have not yet been added to the staging area. Git knows about the file from the last snapshot, sees that it has been altered, but is not yet prepared for the next commit.
2. **Staged:** A file is in a *staged* state after you have run git add on it. This means you have marked a modified file in its current version to be included in the next commit snapshot. The snapshot of the file is now in the staging area, waiting to be permanently recorded.
3. **Committed:** A file is in a *committed* state when the data is safely stored in your local repository. This happens after you run git commit. The version of the file in the staging area is taken and stored permanently in the Git directory.

The basic Git workflow is a continuous cycle through these states: you modify files in your working directory, selectively stage the changes you want to include in your next snapshot, and then commit that snapshot, which permanently records it in the repository's history.

### 1.4 Core Git Objects: The Building Blocks of History

Git is fundamentally a content-addressable filesystem, meaning it stores its data as a series of objects in the .git directory. Understanding these core objects is key to understanding how Git constructs and navigates history.

* **Commits:** A commit is the fundamental unit of Git history. It is a **snapshot** of your entire repository at a specific point in time, not a "diff" or a set of changes from the previous version. This architectural choice is a primary reason for Git's remarkable speed and powerful local capabilities. Because every clone contains the full history of these snapshots, operations like viewing history (git log) or comparing versions (git diff) are nearly instantaneous and do not require network access to a central server. Each commit object contains:
  + A unique SHA-1 hash (a 40-character checksum) that serves as its identifier.
  + Metadata, including the author's name and email, the committer's name and email, the date, and a descriptive commit message.
  + A pointer to the root tree object, which represents the snapshot of the project's content.
  + One or more pointers to its parent commit(s). An initial commit has zero parents, a standard commit has one parent, and a merge commit has two or more parents. This chain of parent pointers forms the directed acyclic graph (DAG) that constitutes the project's history.
* **Branches:** A branch in Git is not a copy of your codebase. It is simply a lightweight, movable **pointer** to a specific commit.The default branch in a new repository is typically named  
  main or master. When you make a new commit, the branch pointer you are currently on automatically moves forward to point to that new commit. Because branches are just simple pointers (a 41-byte file containing a commit hash), they are incredibly cheap and fast to create, switch between, and delete. This technical implementation is not a minor detail; it is the direct enabler of modern software development workflows. The low cost of branching encourages developers to create new branches for every feature, bug fix, or experiment, isolating work until it is complete and ready for integration. This fosters parallel development and innovation by providing a safe environment to try new ideas without affecting the stability of the main codebase.
* **HEAD:** HEAD is a special pointer that indicates your current location within the repository. In most cases, HEAD points to the local branch you are currently working on (e.g., HEAD -> main). When you use git switch or git checkout to change branches, you are simply moving the HEAD pointer to a different branch pointer.It is also possible for  
  HEAD to point directly to a commit hash instead of a branch, a state known as a "detached HEAD."
* **Tags:** A tag is another type of pointer to a specific commit, much like a branch. However, unlike a branch, a tag does not move. It is a permanent marker for a specific point in history, typically used to label release versions like v1.0.0 or v2.1.

## Part 2: Core Command Reference

This section provides a comprehensive reference for the most essential Git commands, organized by common workflow categories. Each command entry includes its purpose, common syntax with key options, and detailed, practical examples.

### 2.1 Configuration & Repository Setup

These commands are used to configure your Git environment and to create or obtain a Git repository.

#### git config

* **Purpose:** The git config command is used to view and set configuration variables that control all aspects of how Git looks and operates. These settings can be stored at three different levels: system-wide, global (user-specific), and local (repository-specific).1 The local level overrides the global level, which in turn overrides the system level.The first action after installing Git should be to configure your user name and email address.
* **Common Syntax:**  
  Bash  
  # Set a configuration value  
  git config [--local | --global | --system] <section.key> <value>  
    
  # View a configuration value  
  git config <section.key>  
    
  # List all configuration settings  
  git config --list
* **Key Options:**
  + --global: Sets the configuration for the current user, applying to all of their repositories. The settings are stored in ~/.gitconfig.
  + --local: Sets the configuration for the current repository only. This is the default scope if none is specified. The settings are stored in .git/config within the repository.
  + --system: Sets the configuration for all users on the system. The settings are stored in $(prefix)/etc/gitconfig.
  + --list or -l: Displays all configuration settings from all levels that are currently in effect.
  + --edit or -e: Opens the relevant configuration file in your default text editor for manual editing.
* **Detailed Examples:**
  1. **Set User Information:** This is essential as this information is embedded into every commit you create.  
     Bash  
     # Set your name for all of your repositories  
     git config --global user.name "John Doe"  
       
     # Set your email for all of your repositories  
     git config --global user.email "johndoe@example.com"  
       
     These settings ensure that all contributions are correctly attributed.
  2. **Set a Default Text Editor:** Git will use this editor for tasks like writing commit messages.  
     Bash  
     # Set Visual Studio Code as the default editor  
     git config --global core.editor "code --wait"  
       
     If not set, Git defaults to the system's default editor, often vi or vim.
  3. **Create a Command Alias:** Aliases are shortcuts for longer Git commands.  
     Bash  
     # Create an alias 'st' for 'status'  
     git config --global alias.st status  
       
     # Create a more complex alias 'lg' for a formatted log view  
     git config --global alias.lg "log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold yellow)%d%C(reset)' --all"  
       
     After setting these, you can run git st instead of git status, and git lg for a detailed, colorized log output.
  4. **View Current Settings:**  
     Bash  
     # List all settings currently applied  
     git config --list

#### git init

* **Purpose:** The git init command creates a new Git repository. It can be used to convert an existing, unversioned project into a Git repository or to initialize a new, empty repository from scratch.This is typically the very first command you run in a new project.
* **Common Syntax:**  
  Bash  
  # Initialize a repository in the current directory  
  git init  
    
  # Initialize a repository in a specific new directory  
  git init <directory>
* **Key Options:**
  + --bare: Creates a "bare" repository. A bare repository does not have a working directory, meaning you cannot edit files or make commits directly within it. Its primary purpose is to serve as a central repository on a server that developers can push to and pull from. Conventionally, bare repository directories end in .git.
* **Detailed Examples:**
  1. **Initialize an Existing Project:** Navigate to the root directory of your project and run the command.  
     Bash  
     # Navigate to your project folder  
     cd my-existing-project  
       
     # Initialize it as a Git repository  
     git init  
       
     This command creates a .git subdirectory in my-existing-project, which contains all the necessary repository files. The existing project files are not altered but are now ready to be tracked.
  2. **Create a New Project from Scratch:**  
     Bash  
     # Create a new directory for the project  
     mkdir new-project  
     cd new-project  
       
     # Initialize the Git repository  
     git init  
       
     This creates an empty Git repository. The next step would typically be to create a README file, add it, and make the initial commit.
  3. **Create a Bare Repository for a Server:** This is an administrative task, typically performed on a remote server to set up a central collaboration point.  
     Bash  
     # On a server, create a bare repository  
     git init --bare my-project.git  
       
     Developers would then clone this repository to their local machines to start working. Pushing to a non-bare repository can overwrite changes, which is why central repositories should always be bare.

#### git clone

* **Purpose:** The git clone command is used to create a local copy of an existing remote repository. It is the most common way to obtain a working copy of a project. The command downloads the entire repository, including all files, branches, and the complete project history.
* **Common Syntax:**  
  Bash  
  # Clone a repository using HTTPS  
  git clone https://github.com/user/repo.git  
    
  # Clone a repository using SSH  
  git clone git@github.com:user/repo.git  
    
  # Clone into a specific directory  
  git clone <url> <directory-name>
* **Key Options:**
  + --branch <branch-name> or -b <branch-name>: Clones the repository and checks out a specific branch instead of the default branch.
  + --single-branch: Clones only the history of the specified (or default) branch, which can significantly reduce download time and disk space for very large repositories.
  + --depth <number>: Creates a "shallow clone" with only the specified number of commits from the tip of each branch's history. git clone --depth=1 is very common for continuous integration (CI) systems that only need the latest version of the code.
* **Detailed Examples:**
  1. **Standard Clone:** This is the most common use case. It creates a directory with the same name as the repository.  
     Bash  
     git clone https://github.com/libgit2/libgit2.git  
       
     After this command completes, a new directory named libgit2 will exist on your local machine. This local repository is automatically configured with a remote named origin that points back to the original URL, making it easy to pull updates and push changes later.
  2. **Clone a Specific Branch:** If you only need to work on a specific feature branch and not the main branch.  
     Bash  
     git clone --branch develop https://github.com/user/repo.git  
       
     This command clones the entire repository but leaves you on the develop branch instead of the default main branch.
  3. **Create a Shallow Clone:** Useful for large projects when you don't need the entire history.  
     Bash  
     git clone --depth 1 https://github.com/torvalds/linux.git  
       
     This clones the Linux kernel repository but only fetches the most recent commit, saving a massive amount of time and disk space.

### 2.2 The Daily Workflow: Recording Changes

These commands form the core loop of a developer's daily interaction with Git: checking status, staging changes, and committing snapshots.

#### git status

* **Purpose:** The git status command displays the state of the working directory and the staging area. It is arguably the most frequently used Git command, as it provides a clear overview of which files have been modified, which changes are staged for the next commit, and which files are untracked by Git. It is essential for understanding the current context of your repository before proceeding with other commands.
* **Common Syntax:**  
  Bash  
  git status
* Detailed Example and Output Analysis:  
  Running git status in a repository with various changes might produce the following output:  
  On branch main  
  Your branch is up to date with 'origin/main'.  
    
  Changes to be committed:  
   (use "git restore --staged <file>..." to unstage)  
   modified: README.md  
   new file: src/feature.js  
    
  Changes not staged for commit:  
   (use "git add <file>..." to update what will be committed)  
   (use "git restore <file>..." to discard changes in working directory)  
   modified: package.json  
    
  Untracked files:  
   (use "git add <file>..." to include in what will be committed)  
   .env  
   dist/  
  + **Changes to be committed**: This section lists files that are in the staging area. In this example, README.md was modified and staged, and a new file src/feature.js was created and staged. These changes will be included in the next commit.
  + **Changes not staged for commit**: This lists modified files in the working directory that have not yet been staged. Here, package.json has been changed, but git add has not been run on it yet.
  + **Untracked files**: This section shows files that exist in the working directory but are not tracked by Git at all. .env and the dist/ directory are new and have never been added to the repository. To prevent certain files (like build artifacts or environment variables) from ever showing up here, their paths should be added to a .gitignore file.

#### git add

* **Purpose:** The git add command moves changes from the working directory to the staging area, preparing them for the next commit. This command is the gateway to the "commit construction zone," allowing you to build a snapshot piece by piece.
* **Common Syntax:**  
  Bash  
  # Add a specific file  
  git add <file>  
    
  # Add all changes (new, modified, deleted) in the current directory and subdirectories  
  git add.  
    
  # Add all changes in the entire repository  
  git add -A
* **Key Options:**
  + -p or --patch: Enters an interactive mode that allows you to review changes "hunk by hunk" (small sections of changes) and decide whether to stage each hunk individually. This is an incredibly powerful feature for creating clean, atomic commits from a file with multiple logical changes.
  + -i or --interactive: Enters a more comprehensive interactive mode with a menu of options for staging, unstaging, and reviewing files.
* **Detailed Examples:**
  1. **Staging Specific Files:**  
     Bash  
     # Stage changes made to the README and a specific source file  
     git add README.md src/main.js
  2. **Staging All Changes:** A common but potentially blunt approach. It's often better to be explicit.  
     Bash  
     # Stage all new, modified, and deleted files from the current directory downwards  
     git add.  
       
     It is good practice to run git status before and after git add. to ensure you are not staging unintended files.
  3. **Interactive Staging with Patch Mode:** This is a hallmark of an advanced Git user.  
     Bash  
     git add -p src/utils.js  
       
     Git will then display the first block of changes in src/utils.js and prompt you with options like y (stage this hunk), n (do not stage this hunk), s (split this hunk into smaller hunks), e (manually edit this hunk), and q (quit). This provides fine-grained control over the contents of your next commit.

#### git commit

* **Purpose:** The git commit command takes the snapshot of changes currently in the staging area and permanently records it in the repository's history. Each commit is a new point in the project's timeline that you can return to later. Writing a clear, descriptive, and standardized commit message is a critical best practice.
* **Common Syntax:**  
  Bash  
  # Commit staged changes with an inline message  
  git commit -m "Your descriptive commit message"  
    
  # Open the default text editor to write a more detailed commit message  
  git commit
* **Key Options:**
  + -m "<message>": Provides the commit message on the command line. This is suitable for short, simple messages.
  + -a or --all: Automatically stages every file that is already tracked by Git (i.e., modified or deleted files) before committing. It will *not* stage new, untracked files. This is a shortcut that bypasses the need to run git add explicitly, but it can lead to less carefully crafted commits.
  + --amend: Modifies the most recent commit. Instead of creating a new commit, it combines any staged changes with the contents of the previous commit to create a new, replacement commit. This is extremely useful for fixing a typo in the last commit message or adding a file you forgot to include. This action  
    *rewrites history* and should not be used on commits that have already been pushed to a shared remote.
* **Detailed Examples:**
  1. **Standard Commit Workflow:**  
     Bash  
     # Stage the files you want to commit  
     git add file1.js file2.js  
       
     # Commit the staged snapshot with a clear message  
     git commit -m "feat: Implement user authentication endpoint"
  2. **Using the -a Shortcut:**  
     Bash  
     # Assuming file1.js and file2.js were already tracked and have been modified  
     git commit -a -m "refactor: Improve error handling in API calls"
  3. **Amending the Previous Commit:** Imagine you just committed but realized you made a typo in the message.  
     Bash  
     # No need to add files if you are only changing the message  
     git commit --amend -m "fix: Correct typo in configuration variable"  
       
     Or, if you forgot to add a file to the last commit:  
     Bash  
     # Add the forgotten file to the staging area  
     git add forgotten-file.css  
       
     # Amend the previous commit to include it  
     git commit --amend --no-edit  
       
     The --no-edit flag prevents the editor from opening, keeping the original commit message.

#### git diff

* **Purpose:** The git diff command is used to show the differences between various states in a Git repository. It is a powerful tool for reviewing changes before staging, committing, or merging them.
* **Common Syntax:**  
  Bash  
  # Show changes in the working directory that are not yet staged  
  git diff  
    
  # Show changes that are staged but not yet committed  
  git diff --staged  
    
  # Show changes between two commits or branches  
  git diff <commit1>..<commit2>
* **Detailed Examples:**
  1. **Viewing Unstaged Changes:** This is the most common use. It shows the difference between your working directory and the staging area.  
     Bash  
     git diff  
       
     The output will show every line that has been added, removed, or modified in your tracked files since the last time you ran git add.
  2. **Viewing Staged Changes:** This shows the difference between the staging area and your last commit (HEAD). This is a final review of exactly what will be included in your next commit.  
     Bash  
     git diff --staged  
       
     This command is invaluable for ensuring your commit is clean and contains only the intended changes.
  3. **Comparing Two Branches:** This is essential for code review, such as before merging a feature branch into the main branch.  
     Bash  
     # Show all changes on `feature-branch` that are not on `main`  
     git diff main..feature-branch  
       
     The output will be a patch that represents the cumulative changes made on the feature-branch since it diverged from main.

### 2.3 Branching and Merging: Parallel Development

Branching is one of Git's most powerful features, enabling isolated, parallel development. These commands allow you to create, manage, and integrate these parallel lines of work.

#### git branch

* **Purpose:** The git branch command is the primary tool for managing branches. It can be used to list, create, and delete branches. As established in Part 1, branches are simply lightweight pointers to commits, making these operations nearly instantaneous.
* **Common Syntax:**  
  Bash  
  # List all local branches  
  git branch  
    
  # Create a new branch  
  git branch <branch-name>  
    
  # Delete a branch (safe: prevents deletion if unmerged changes exist)  
  git branch -d <branch-name>  
    
  # Force delete a branch (discards unmerged changes)  
  git branch -D <branch-name>
* **Key Options:**
  + -a or --all: Lists both local and remote-tracking branches.
  + -v or --verbose: Shows the last commit on each branch.
  + -d or --delete: Deletes a branch. Git will prevent the deletion if the branch contains work that has not been merged into another branch.
  + -D: Force-deletes a branch, regardless of its merged status. This should be used with caution as it can lead to losing work.
* **Detailed Examples:**
  1. **Listing Branches:**  
     Bash  
     git branch  
       
     Output:  
      develop  
     \* main  
      feature/user-auth  
       
     The asterisk (\*) indicates the currently checked-out branch (HEAD).
  2. **Creating a New Branch:**  
     Bash  
     # Create a new branch called 'refactor-database' based on the current commit  
     git branch refactor-database  
       
     This creates the new branch but does *not* switch to it. The current branch remains main.
  3. **Deleting a Branch:** After a feature branch has been successfully merged, it is common practice to delete it to keep the repository clean.  
     Bash  
     # Assuming 'feature/user-auth' has been merged into 'main'  
     git branch -d feature/user-auth

#### git checkout & git switch

* **Purpose:** These commands are used to navigate between branches. git checkout is the traditional, multi-purpose command, while git switch is a modern, safer command introduced in Git version 2.23 specifically for branch operations.The introduction of  
  git switch (and git restore) was a direct response to the common confusion caused by git checkout's overloaded functionality, which handles not only branch switching but also file restoration. For modern workflows,  
  git switch is the recommended command for changing branches due to its clarity and single-responsibility design.
* **git switch (Recommended):**
  + **Common Syntax:**  
    Bash  
    # Switch to an existing branch  
    git switch <branch-name>  
      
    # Create a new branch and switch to it  
    git switch -c <new-branch-name>  
      
    # Switch back to the previously checked-out branch  
    git switch -
  + **Detailed Examples:**
    1. **Switching to an Existing Branch:**  
       Bash  
       git switch develop  
         
       This moves the HEAD pointer to the develop branch, and updates the files in the working directory to match the snapshot of the commit that develop points to.
    2. **Creating and Switching:**  
       Bash  
       # Create a new branch 'fix/login-bug' from the current branch and switch to it  
       git switch -c fix/login-bug  
         
       This is the modern equivalent of git checkout -b.
* **git checkout (Traditional):**
  + **Common Syntax:**  
    Bash  
    # Switch to an existing branch  
    git checkout <branch-name>  
      
    # Create a new branch and switch to it  
    git checkout -b <new-branch-name>  
      
    # Restore a file in the working directory to its last committed state  
    git checkout -- <file-name>
  + **Detailed Examples:**
    1. **Branch Switching:**

git checkout feature/user-auth This performs the same action as `git switch feature/user-auth`.[28, 29]  
  
2. \*\*Create and Switch (Shorthand):\*\*  
 ```bash  
 git checkout -b hotfix/critical-issue  
 ```  
 This is a very common idiom for starting new work.[28, 30]

#### git merge

* **Purpose:** The git merge command integrates changes from an independent line of development (a branch) into the current branch.It combines two or more development histories together.
* **Common Syntax:**  
  Bash  
  # Switch to the receiving branch  
  git switch <receiving-branch>  
    
  # Merge the source branch into the current branch  
  git merge <source-branch>
* **Merge Strategies:**
  1. **Fast-Forward Merge:** If the history of the receiving branch is a direct ancestor of the source branch (i.e., no new commits have been made on the receiving branch), Git will perform a fast-forward merge. It simply moves the receiving branch's pointer forward to match the source branch's pointer. No new merge commit is created.
  2. **Three-Way Merge:** If the branches have diverged, Git creates a new "merge commit." This special commit has two parents: one from the receiving branch and one from the source branch. This new commit unifies the two independent histories.
* **Key Options:**
  + --no-ff: Prevents a fast-forward merge and forces the creation of a merge commit, even when a fast-forward is possible. This can be useful for record-keeping, as it clearly marks where a feature was merged in the history.
  + --abort: If a merge results in conflicts, this command can be used to abort the merge process and return the repository to its state before the merge was attempted.
* **Detailed Examples:**
  1. **Fast-Forward Merge Workflow:**  
     Bash  
     # Start on the main branch  
     git switch main  
       
     # Create and switch to a new feature branch  
     git switch -c new-feature  
       
     #... make some commits on new-feature...  
       
     # Switch back to the main branch  
     git switch main  
       
     # Merge the feature branch. Assuming no new commits on main, this will be a fast-forward.  
     git merge new-feature
  2. **Three-Way Merge and Conflict Resolution:**  
     Bash  
     #... starting from a state where both 'main' and 'feature' have new, divergent commits...  
       
     # Switch to the receiving branch  
     git switch main  
       
     # Attempt to merge the feature branch  
     git merge feature  
       
     If there are conflicts, the output will be:  
     Auto-merging index.html  
     CONFLICT (content): Merge conflict in index.html  
     Automatic merge failed; fix conflicts and then commit the result.  
       
     To resolve the conflict:  
     a. Open index.html. You will see conflict markers:  
     HTML  
     <<<<<<< HEAD  
     <p>This is the content from the main branch.</p>  
     =======  
     <p>This is the new content from the feature branch.</p>  
     >>>>>>> feature  
       
     b. Edit the file to remove the markers and create the desired final version. For example:  
     HTML  
     <p>This is the final, merged content.</p>  
       
     c. Stage the resolved file and commit the merge:  
     Bash  
     git add index.html  
     git commit -m "Merge branch 'feature' and resolve conflicts"  
       
     Alternatively, after adding the resolved files, you can just run git commit, and Git will provide a default merge commit message.30

### 2.4 Collaboration: Working with Remotes

These commands are essential for multi-developer workflows, allowing you to synchronize your local repository with repositories hosted on other machines or services.

#### git remote

* **Purpose:** The git remote command is used to manage the set of tracked remote repositories. A "remote" is a bookmark or alias for a URL of a repository, making it easier to interact with. When you clone a repository, Git automatically creates a remote named  
  origin pointing to the URL you cloned from.
* **Common Syntax:**  
  Bash  
  # List the shortnames of all configured remotes  
  git remote  
    
  # List remotes with their corresponding URLs  
  git remote -v  
    
  # Add a new remote  
  git remote add <name> <url>  
    
  # Remove a remote  
  git remote remove <name>  
    
  # Rename a remote  
  git remote rename <old-name> <new-name>
* **Detailed Examples:**
  1. **Viewing Remotes:**  
     Bash  
     git remote -v  
       
     Output:  
     origin https://github.com/user/repo.git (fetch)  
     origin https://github.com/user/repo.git (push)  
       
     This shows the origin remote and the URLs used for fetching and pushing.
  2. **Adding a New Remote:** This is common when you want to collaborate with another developer's repository directly or have a separate remote for deployment.  
     Bash  
     # Add a remote named 'upstream' pointing to the original project repository  
     git remote add upstream https://github.com/original-author/repo.git  
       
     You can now fetch changes from this upstream remote.
  3. **Removing a Remote:**  
     Bash  
     git remote remove upstream

#### git fetch

* **Purpose:** The git fetch command downloads commits, files, and refs from a remote repository into your local repository, but it **does not** automatically merge or modify your current work. It updates your local copies of the remote branches (e.g.,  
  origin/main). This is a "safe" operation because it never changes your local branches (like main) or your working directory.
* **Common Syntax:**  
  Bash  
  # Fetch updates from the 'origin' remote  
  git fetch origin  
    
  # Fetch updates from all configured remotes  
  git fetch --all  
    
  # Fetch and also remove any remote-tracking branches that no longer exist on the remote  
  git fetch --prune
* Detailed Example:  
  Imagine your colleague has pushed new commits to the main branch on origin. Your local main is behind.  
  Bash  
  # Download all new information from the origin remote  
  git fetch origin  
    
  After running this command, your origin/main pointer will be updated to reflect the new state of the remote repository. Your local main branch, however, remains untouched. You can now inspect the new changes before deciding how to integrate them.

The professional workflow for staying synchronized with a remote repository prioritizes safety and control over convenience. Instead of using git pull directly, which combines fetching and merging into one opaque step, a more robust approach is to "Fetch, Inspect, Integrate."

1. **Fetch:** Always start by running git fetch origin to safely download all remote changes without affecting your local work.
2. **Inspect:** Use a command like git log main..origin/main to see a list of the commits that have been downloaded but are not yet in your local branch. This allows you to review the incoming changes.
3. Integrate: After reviewing, make a deliberate choice to integrate the changes using either git merge origin/main or git rebase origin/main.  
   This three-step process prevents unexpected merge conflicts in your working directory and gives you complete control over how your local history is combined with the remote history.37

#### git pull

* **Purpose:** The git pull command is a convenient shortcut that combines two operations: it runs git fetch to download content from the remote, and then immediately runs git merge (or git rebase) to integrate the downloaded content into your current local branch.
* **Common Syntax:**  
  Bash  
  # Fetch from and merge the corresponding remote branch into the current branch  
  git pull  
    
  # Explicitly pull from a specific branch on a remote  
  git pull <remote-name> <branch-name>
* **Key Options:**
  + --rebase: Instead of using git merge to integrate the changes, this option uses git rebase. It takes your local commits, sets them aside, applies the new commits from the remote, and then re-applies your local commits on top of the updated branch. This results in a clean, linear history without a merge commit.
* **Detailed Examples:**
  1. **Standard Pull (Merge Strategy):**  
     Bash  
     # Assuming you are on the 'main' branch which tracks 'origin/main'  
     git pull  
       
     This is equivalent to running git fetch origin followed by git merge origin/main. If there are divergent changes, this will create a merge commit.
  2. **Pull with Rebase:** This is a very common workflow for teams that prefer a linear commit history.  
     Bash  
     # Assuming you are on your feature branch  
     git pull --rebase origin main  
       
     This command fetches the latest main branch from origin and then re-applies your feature branch's commits on top of it. This is an excellent way to keep your feature branch up-to-date with the main project line.

#### git push

* **Purpose:** The git push command is the counterpart to git fetch. It is used to upload your local repository content (commits) to a remote repository, sharing your changes with others.
* **Common Syntax:**  
  Bash  
  # Push the current branch to its configured upstream remote  
  git push  
    
  # Explicitly push a local branch to a remote  
  git push <remote-name> <local-branch-name>  
    
  # Push a local branch to a remote branch with a different name  
  git push <remote-name> <local-branch-name>:<remote-branch-name>
* **Key Options:**
  + --force: This is a destructive option that overwrites the history on the remote branch with your local history. It should be used with extreme caution and generally only on feature branches that you are the sole contributor to, typically after a local rebase. It can erase commits that your teammates have pulled, causing major problems.
  + --force-with-lease: A safer alternative to --force. It will only force the push if the remote branch is in the state you expect (i.e., no one else has pushed new commits to it since your last fetch). This helps prevent overwriting a colleague's work.
  + --tags: By default, git push does not send tags. This flag pushes all of your local tags to the remote.
  + --delete: Deletes a branch on the remote repository. git push origin --delete <branch-name>.
* **Detailed Examples:**
  1. **Pushing a Feature Branch:**  
     Bash  
     # Push the local 'feature/user-auth' branch to the 'origin' remote  
     git push origin feature/user-auth  
       
     If the branch does not exist on the remote, it will be created.
  2. **Setting an Upstream Branch:** The first time you push a new local branch, you can set its remote counterpart as the "upstream" tracking branch.  
     Bash  
     git push --set-upstream origin feature/user-auth  
     # or the shorthand:  
     git push -u origin feature/user-auth  
       
     After this is set, you can simply run git push from that branch in the future, and Git will know where to send the commits.
  3. **Force Pushing After a Rebase:** Imagine you cleaned up your local feature branch history with an interactive rebase. Your local history now diverges from the remote version of the branch. A normal push will be rejected.  
     Bash  
     # DANGEROUS: Overwrites the remote branch. Ensure no one else is using it.  
     git push origin feature/user-auth --force-with-lease

### 2.5 Inspecting History and State

These commands allow you to explore the commit history and inspect the state of your repository at different points in time.

#### git log

* **Purpose:** The git log command is the primary tool for viewing the commit history of a repository. By default, it displays commits in reverse chronological order.
* **Common Syntax:**  
  Bash  
  # Show the full commit history  
  git log  
    
  # Show history for a specific file or directory  
  git log -- <path>
* **Key Formatting Options:**
  + --oneline: Condenses each commit to a single line, showing the abbreviated commit hash and the commit message subject.
  + --stat: For each commit, shows a summary of the files changed and the number of lines added and removed.
  + --patch or -p: For each commit, shows the full diff (the "patch") of the changes introduced.
  + --graph: Displays an ASCII art graph showing the branching and merging history.
  + --decorate: Displays the names of branches and tags that point to each commit.
  + --pretty=format:"<string>": Allows for complete customization of the log output using placeholders like %h (abbreviated hash), %an (author name), %ar (author date, relative), and %s (subject).
* **Detailed Examples:**
  1. **A Clean, Graph-based Log:** This is a very popular and useful combination of options.  
     Bash  
     git log --graph --oneline --decorate --all  
       
     This command provides a compact, visual representation of the entire repository history across all branches.
  2. **Viewing Changes for a Specific File:**  
     Bash  
     # Show the full diff for every commit that changed the package.json file  
     git log -p -- package.json
  3. **Filtering Commits:**  
     Bash  
     # Show all commits by a specific author since last week  
     git log --author="John Doe" --since="1 week ago"

#### git show

* **Purpose:** The git show command displays information about a specific Git object, most commonly a commit. For a commit, it shows the commit metadata (author, date, full message) followed by the diff of the changes introduced by that commit.
* **Common Syntax:**  
  Bash  
  # Show the most recent commit (HEAD)  
  git show  
    
  # Show a specific commit  
  git show <commit-hash>  
    
  # Show the commit a branch or tag points to  
  git show <branch-or-tag-name>
* **Detailed Example:**  
  Bash  
  # Inspect a specific commit from the log  
  git show a1b2c3d4  
    
  The output will provide the full commit message and a detailed patch showing exactly what changed in that commit, making it perfect for reviewing a single, specific change.

#### git blame

* **Purpose:** The git blame command is a code archaeology tool. For a given file, it annotates each line with information about the commit that last modified it, including the commit hash, the author, and the timestamp.8 It is invaluable for understanding the history and context of a specific line of code—for finding out who wrote it, when, and why.
* **Common Syntax:**  
  Bash  
  git blame <file-name>
* **Detailed Example:**  
  Bash  
  git blame src/database.js  
    
  The output will show the entire content of src/database.js, with each line prefixed by the details of the commit that last changed it. This helps answer questions like, "Why was this line of code written this way?" by pointing you to the exact commit where the change was made.

### 2.6 Undoing Changes and Rewriting History

Git provides a powerful set of tools for undoing mistakes and cleaning up history. A critical concept governing the use of these tools is the distinction between **private** and **public** history. Private history consists of commits on your local machine that have not yet been pushed to a shared remote. This history can be freely and safely rewritten. Public history consists of commits that have been pushed and may be in use by other developers. This history should be considered immutable; mistakes should be corrected with new commits that preserve the existing history.

#### git revert

* **Purpose:** git revert is the **safe** way to undo a commit that is part of the public history. It does not delete or alter the original commit. Instead, it creates a  
  *new* commit that applies the inverse of the changes from the specified commit. This preserves the project history, which is essential for collaboration.
* **Common Syntax:**  
  Bash  
  # Revert the changes from a specific commit  
  git revert <commit-hash>  
    
  # Revert the most recent commit  
  git revert HEAD
* Detailed Example:  
  Suppose you discover that commit a1b2c3d4 introduced a bug, but it has already been pushed and other work has been built on top of it.  
  Bash  
  git revert a1b2c3d4  
    
  This will open your text editor to create a commit message for the new revert commit (e.g., "Revert 'feat: Add experimental caching'"). After you save the message, a new commit will be added to the end of your branch, effectively undoing the changes from a1b2c3d4 without rewriting the shared history.

#### git reset

* **Purpose:** git reset is a powerful and potentially destructive command used to move the current branch pointer (HEAD) to a specified commit, and optionally modify the staging area and working directory. It is primarily used for undoing changes in your **private**, local history. The cardinal rule of  
  git reset is to **never use it on commits that have been pushed to a public branch**.
* **Modes of Operation:** git reset operates in three main modes, corresponding to how it affects Git's "three trees".
  1. --soft: Moves the HEAD pointer to the specified commit. The staging area and working directory are left untouched. The changes from the "reset" commits are left as staged changes.
  2. --mixed (the default): Moves the HEAD pointer and resets the staging area to match the specified commit. The working directory is untouched. The changes from the "reset" commits are preserved in the working directory as unstaged changes.
  3. --hard: Moves the HEAD pointer and resets both the staging area and the working directory to match the specified commit. **This is destructive; any uncommitted changes and all commits after the target commit are permanently lost.**
* **Detailed Examples:**
  1. **Unstaging a File:** A common, safe use of reset.  
     Bash  
     # Oops, I added a file I didn't mean to  
     git add.  
     git status # shows log.txt is staged  
       
     # Unstage log.txt but keep it in the working directory  
     git reset log.txt  
       
     This is equivalent to git restore --staged log.txt.
  2. **Squashing Local Commits (--soft):** Suppose you have three messy local commits you want to combine into one.  
     Bash  
     # Go back three commits, but keep all changes from them staged  
     git reset --soft HEAD~3  
       
     # Now, create a single, clean commit  
     git commit -m "feat: Implement the entire new feature"
  3. **Completely Discarding Local Commits (--hard):**  
     Bash  
     # DANGER: This will permanently delete the last two commits and all uncommitted work.  
     git reset --hard HEAD~2  
       
     This command should be used with extreme care, typically when you want to throw away local experiments that have gone wrong.

#### git restore

* **Purpose:** git restore is a modern, safer command introduced to handle two specific "undo" scenarios that were previously managed by the overloaded git checkout and git reset commands: discarding changes in the working directory and unstaging changes from the index.
* **Common Syntax:**  
  Bash  
  # Discard changes in a working directory file, reverting it to its staged state (or last commit if not staged)  
  git restore <file>  
    
  # Unstage a file, moving it from the staging area back to the working directory  
  git restore --staged <file>
* **Detailed Examples:**
  1. **Discarding Local Modifications:** You've made changes to a file but decide you don't want them.  
     Bash  
     # Revert config.yml to the version in the last commit  
     git restore config.yml
  2. **Unstaging a File:** This is the modern, recommended way to unstage a file.  
     Bash  
     # Unstage the README.md file  
     git restore --staged README.md

#### git clean

* **Purpose:** The git clean command is used to remove **untracked** files from the working directory. This is useful for cleaning up build artifacts, log files, or other generated files that are not part of the repository and are not listed in .gitignore.
* **Common Syntax & Safety Workflow:** git clean is destructive and cannot be undone. Always perform a dry run first.  
  Bash  
  # 1. Dry Run: See what would be deleted  
  git clean -n  
    
  # 2. Dry Run (including directories):  
  git clean -nd  
    
  # 3. Execute: If the dry run looks correct, force the deletion  
  git clean -f  
    
  # 4. Execute (including directories):  
  git clean -fd
* **Key Options:**
  + -n or --dry-run: Shows what would be done without actually doing it.
  + -f or --force: Required to actually delete the files.
  + -d: Also removes untracked directories.

#### git rebase

* **Purpose:** git rebase is a history-rewriting command that re-applies a series of commits on top of a different base commit. Its primary goal is to maintain a clean, linear project history. Instead of a merge commit, rebasing makes it appear as if the work was done sequentially on top of the latest changes. Like  
  git reset, this rewrites history and should only be used on **private**, un-pushed branches.
* **Common Syntax:**  
  Bash  
  # Re-apply the current branch's commits on top of the 'main' branch  
  git rebase main  
    
  # Start an interactive rebase session for the last 3 commits  
  git rebase -i HEAD~3
* **Key Options:**
  + -i or --interactive: This is the most powerful feature of rebase. It opens an editor with a list of the commits being rebased, allowing you to reorder, edit, squash (combine), or remove them.
* **Detailed Examples:**
  1. **Updating a Feature Branch:** This is the most common use case.  
     Bash  
     # Switch to your feature branch  
     git switch feature/new-api  
       
     # Fetch the latest changes from the remote  
     git fetch origin  
       
     # Re-apply your commits on top of the latest main branch  
     git rebase origin/main  
       
     This integrates the latest project updates into your branch while keeping your branch's history linear and easy to follow.
  2. **Interactive Rebase for Cleanup:** Before creating a pull request, you can clean up your messy commit history.  
     Bash  
     git rebase -i HEAD~4  
       
     This opens an editor with the last four commits:  
     pick a1b2c3d4 Add initial API structure  
     pick e5f6g7h8 WIP  
     pick i9j0k1l2 fix typo  
     pick m3n4o5p6 Add final endpoint  
       
     You can change pick to squash (or s) to combine commits, reword (or r) to change the message, edit (or e) to amend a commit mid-rebase, or reorder the lines to reorder the commits. This allows you to present a clean, logical set of commits for code review.

### 2.7 Advanced Tools and Techniques

These commands provide additional functionality for managing specific situations in your workflow.

#### git stash

* **Purpose:** The git stash command temporarily shelves (or stashes) your uncommitted changes (both staged and unstaged) in the working directory, allowing you to revert to a clean working state. This is useful when you need to quickly switch branches to work on an urgent bug but aren't ready to commit your current work.
* **Common Syntax:**  
  Bash  
  # Stash your current changes  
  git stash  
    
  # List all stashes  
  git stash list  
    
  # Re-apply the most recent stash and remove it from the list  
  git stash pop  
    
  # Re-apply the most recent stash but keep it in the list  
  git stash apply  
    
  # Discard the most recent stash  
  git stash drop
* **Detailed Example:**
  1. You are working on feature-A and have uncommitted changes.
  2. An urgent bug report comes in.
  3. git stash - Your working directory is now clean.
  4. git switch hotfix-branch - Switch to the hotfix branch.
  5. ... fix the bug, commit, and merge the fix.
  6. git switch feature-A - Return to your feature branch.
  7. git stash pop - Your uncommitted changes are restored, and you can continue where you left off.15

#### git tag

* **Purpose:** The git tag command is used to create a permanent marker for a specific point in the repository's history, typically used to mark a software release version (e.g., v1.0).
* **Common Syntax:**  
  Bash  
  # Create a lightweight tag at the current commit  
  git tag <tag-name>  
    
  # Create an annotated tag with a message  
  git tag -a <tag-name> -m "Tag message"  
    
  # List all tags  
  git tag  
    
  # Push all tags to the remote repository  
  git push origin --tags
* **Detailed Example:**  
  Bash  
  # Create an annotated tag for version 1.0 release  
  git tag -a v1.0 -m "Version 1.0 Release"  
    
  # Push the new tag to the origin remote  
  git push origin v1.0

#### git rm & git mv

* **Purpose:** When working with files already tracked by Git, you should use git rm and git mv instead of the standard shell commands rm and mv. These Git commands perform the filesystem operation (deleting or renaming the file) and also automatically stage that change for the next commit.
* **Common Syntax:**  
  Bash  
  # Remove a file from the working directory and stage the deletion  
  git rm <file>  
    
  # Rename a file and stage the move  
  git mv <old-name> <new-name>
* **Detailed Example:**  
  Bash  
  # Rename an old file  
  git mv old-feature.js new-feature.js  
    
  # Remove a deprecated file  
  git rm deprecated-style.css  
    
  # Commit the changes  
  git commit -m "refactor: Rename feature and remove deprecated styles"

## Part 3: Appendix: Quick Command Reference

This table provides a high-density summary of the most common Git commands for quick reference during daily development. Each command is linked to its detailed section in this guide for further reading.

| **Command** | **Common Syntax** | **Brief Description** | **See Section** |
| --- | --- | --- | --- |
| git config | git config --global user.name "Name" | Configure user, global, or repository settings. | 2.1 |
| git init | git init | Create a new local repository. | 2.1 |
| git clone | git clone [url] | Copy a remote repository to your local machine. | 2.1 |
| git status | git status | Show the status of the working directory and staging area. | 2.2 |
| git add | git add [file] or git add. | Add file changes to the staging area. | 2.2 |
| git commit | git commit -m "Message" | Record staged changes to the repository history. | 2.2 |
| git branch | git branch [name] or git branch -d [name] | List, create, or delete branches. | 2.3 |
| git switch | git switch [name] or git switch -c [name] | Switch to a different branch or create a new one. | 2.3 |
| git merge | git merge [branch] | Combine the history of another branch into the current one. | 2.3 |
| git remote | git remote -v or git remote add [name][url] | Manage connections to remote repositories. | 2.4 |
| git fetch | git fetch [remote] | Download changes from a remote, but do not integrate. | 2.4 |
| git pull | git pull [remote] | Fetch and integrate changes from a remote. | 2.4 |
| git push | git push [remote][branch] | Upload local commits to a remote repository. | 2.4 |
| git log | git log --oneline --graph | View the commit history. | 2.5 |
| git reset | git reset --hard [commit] | Reset HEAD to a specified state (potentially destructive). | 2.6 |
| git revert | git revert [commit] | Create a new commit that undoes a previous commit (safe). | 2.6 |
| git rebase | git rebase -i [base] | Re-apply commits on top of another base tip. | 2.6 |
| git stash | git stash or git stash pop | Temporarily shelve changes not ready for a commit. | 2.7 |

#### Works cited

1. Git Branching Commands Explained with Examples - freeCodeCamp, accessed on September 7, 2025, <https://www.freecodecamp.org/news/git-branching-commands-explained/>
2. What is Git? - Git, accessed on September 7, 2025, <https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F>
3. Git Guides - git clone - GitHub, accessed on September 7, 2025, <https://github.com/git-guides/git-clone>
4. An Intro to Git and GitHub for Beginners (Tutorial), accessed on September 7, 2025, <https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners>
5. What's the use of the staging area in Git? - Stack Overflow, accessed on September 7, 2025, <https://stackoverflow.com/questions/49228209/whats-the-use-of-the-staging-area-in-git>
6. Understanding the differences between `git commit` and `git push`, accessed on September 7, 2025, <https://graphite.dev/guides/git-commit-vs-push>
7. git init | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init>
8. Useful Git Commands and Basic Concepts - GeeksforGeeks, accessed on September 7, 2025, <https://www.geeksforgeeks.org/git/useful-git-commands-and-basic-concepts/>
9. Branches in a Nutshell - Git, accessed on September 7, 2025, <https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>
10. Reference - Git, accessed on September 7, 2025, <https://git-scm.com/docs>
11. git-config Documentation - Git, accessed on September 7, 2025, <https://git-scm.com/docs/git-config>
12. git config | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config>
13. Top 12 Git commands every developer must know - The GitHub Blog, accessed on September 7, 2025, <https://github.blog/developer-skills/github/top-12-git-commands-every-developer-must-know/>
14. Git config commands | Git tutorial - Nulab, accessed on September 7, 2025, <https://nulab.com/learn/software-development/git-tutorial/git-commands-settings/git-config-commands/>
15. GIT CHEAT SHEET - GitHub Education, accessed on September 7, 2025, <https://education.github.com/git-cheat-sheet-education.pdf>
16. Git Configuration - Git, accessed on September 7, 2025, <https://git-scm.com/book/be/v2/Customizing-Git-Git-Configuration>
17. Git Guides - git init - GitHub, accessed on September 7, 2025, <https://github.com/git-guides/git-init>
18. Start a new git repository - Karl Broman, accessed on September 7, 2025, <https://kbroman.org/github_tutorial/pages/init.html>
19. git clone | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone>
20. gittutorial Documentation - Git, accessed on September 7, 2025, <https://git-scm.com/docs/gittutorial>
21. Git Status: Inspecting a repository | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-status>
22. Learn How to Use the Git Add Command | All, Interactive, Undo, accessed on September 7, 2025, <https://www.gitkraken.com/learn/git/git-add>
23. 8 categories of Git - DEV Community, accessed on September 7, 2025, <https://dev.to/danyson/8-categories-of-git-2127>
24. Basic Git Commands | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/glossary>
25. git-diff Documentation - Git, accessed on September 7, 2025, <https://git-scm.com/docs/git-diff>
26. git switch - Switching branches | Learn Version Control with Git, accessed on September 7, 2025, <https://www.git-tower.com/learn/git/commands/git-switch>
27. Git Switch Branch: A Guide With Practical Examples - DataCamp, accessed on September 7, 2025, <https://www.datacamp.com/tutorial/git-switch-branch>
28. Basic Branching and Merging - Git, accessed on September 7, 2025, <https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>
29. Git Merge | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/using-branches/git-merge>
30. git-merge Documentation - Git, accessed on September 7, 2025, <https://git-scm.com/docs/git-merge>
31. Git Merge Tutorial: A Comprehensive Guide with Examples | DataCamp, accessed on September 7, 2025, <https://www.datacamp.com/tutorial/git-merge>
32. git-remote Documentation - Git, accessed on September 7, 2025, <https://git-scm.com/docs/git-remote>
33. Handling Repositories with Git Remote - GeeksforGeeks, accessed on September 7, 2025, <https://www.geeksforgeeks.org/git/handling-repositories-with-git-remote/>
34. git fetch - Downloading Remote Data | Learn Version Control with Git, accessed on September 7, 2025, <https://www.git-tower.com/learn/git/faq/git-fetch>
35. How to use Git fetch - Graphite, accessed on September 7, 2025, <https://graphite.dev/guides/how-to-use-git-fetch>
36. Git Pull | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/syncing/git-pull>
37. What is Git Pull? - GeeksforGeeks, accessed on September 7, 2025, <https://www.geeksforgeeks.org/git/what-is-git-pull/>
38. Git Push | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/syncing/git-push>
39. Git Push Command Tutorial - Hostman, accessed on September 7, 2025, <https://hostman.com/tutorials/git-push-how-it-works-and-how-to-use-it/>
40. Git Log Command Explained - freeCodeCamp, accessed on September 7, 2025, <https://www.freecodecamp.org/news/git-log-command/>
41. Advanced Git Log | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/git-log>
42. git-show Documentation - Git, accessed on September 7, 2025, <https://git-scm.com/docs/git-show>
43. Git Revert | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/undoing-changes/git-revert>
44. Git Reset | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/undoing-changes/git-reset>
45. git-restore Documentation - Git, accessed on September 7, 2025, <https://git-scm.com/docs/git-restore>
46. Git Clean | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/undoing-changes/git-clean>
47. git rebase | Atlassian Git Tutorial, accessed on September 7, 2025, <https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase>
48. Rebase and resolve merge conflicts - GitLab Docs, accessed on September 7, 2025, <https://docs.gitlab.com/topics/git/git_rebase/>