Git Workflow Guide

you are the documentation creation of my git questions give good response with proper description and example

I can help you create documentation for your Git-related questions! If you need assistance with a specific Git concept, command, or workflow, feel free to ask, and I'll provide a detailed explanation with an example.

For instance, here’s an example of how to document a basic Git workflow:

Git Workflow Documentation

1. Initializing a Git Repository

To start using Git in your project, you need to initialize a Git repository. This will create a `` `.git` `` directory that tracks all changes.

Command:

```
bash

git init
```

Example:

```
bash

cd my-project
git init
```

This will initialize the repository in the `` `my-project` `` folder. Once initialized, you can start tracking files.

---

2. Staging Files for Commit

After modifying files in your project, you need to stage them before committing. This tells Git which files should be included in the next commit.

Command:

```
bash

git add <file-name>
```

To stage all files, use:

```bash
git add .
```

Example:

```bash
git add index.html
git add styles.css
```

This stages `index.html` and `styles.css` files.

---

3. Committing Changes

After staging the files, commit them to the repository. Each commit should have a meaningful message describing what changes were made.

Command:

```bash
git commit -m "Descriptive commit message"
```

Example:

```bash
git commit -m "Add initial webpage layout"
```

---

4. Pushing Changes to Remote Repository

If you're working with a remote repository (like GitHub, GitLab, or Bitbucket), you need to push your commits to sync your local changes with the remote.

Command:

```bash
git push origin main
```

Example:

```bash
git push origin main
```

This pushes changes to the `main` branch on the remote repository.

---

5. Pulling Changes from Remote Repository

If you're collaborating with others and want to update your local repository with changes from the remote repository, use `git pull`.

Command:

```bash
git pull origin main
```

Example:

```bash
git pull origin main
```

This fetches and merges changes from the `main` branch on the remote repository.

Would you like more details or specific examples on any Git commands or workflows? Feel free to ask!

reverting the commit

To **revert a commit** in Git, there are different methods depending on what you want to achieve. Below are the most common ways to revert commits, each with explanations and examples.

1. Reverting a Commit (Creating a New Commit that Undoes the Changes)

The `git revert` command is used to create a new commit that undoes the changes from a previous commit. This is often used in shared repositories because it maintains the history of all changes.

Command:

```bash
git revert <commit-hash>
```

- `<commit-hash>` is the identifier of the commit you want to revert.
- Git will create a new commit that undoes the changes made in the specified commit.

Example:

Let's say you want to revert a commit with the hash `abc1234`:

```bash
git revert abc1234
```

Git will open your default text editor to allow you to edit the commit message for the revert. You can keep the default message (e.g., "Revert commit abc1234") or modify it. Save and close the editor, and Git will create a new commit that undoes the changes from `abc1234`.

---

2. Resetting to a Previous Commit (Discarding Commits)

If you want to **discard** commits, `git reset` is a more powerful tool. This command is typically used for **local changes** and should be used with caution, as it can modify commit history.

There are three types of resets: **soft**, **mixed**, and **hard**.

a. Soft Reset

A **soft reset** will move the HEAD pointer to a previous commit but keep all changes in the staging area (index). This allows you to re-commit the changes if needed.

Command:

```bash
git reset --soft <commit-hash>
```

Example:

To reset to commit `abc1234` but keep changes staged for commit:

```bash
git reset --soft abc1234
```

This will move the HEAD to `abc1234` and keep all changes in the staging area, so you can create a new commit if necessary.

---

b. Mixed Reset (Default Reset)

A **mixed reset** will reset the HEAD to a previous commit and **unstage** changes (but keep them in your working directory).

Command:

```bash
git reset --mixed <commit-hash>
```

Example:

To reset to commit `abc1234` and unstage the changes:

```bash
git reset --mixed abc1234
```

This will move the HEAD to `abc1234` but leave the changes you made in the working directory, so you can decide whether to stage and commit them again.

---

c. Hard Reset

A **hard reset** will reset the HEAD to a previous commit and **discard all changes** in the working directory and staging area. **Be cautious** as this operation is destructive and cannot be easily undone.

Command:

bash

```
git reset --hard <commit-hash>
```

Example:

To reset to commit `abc1234` and discard all changes:

bash

```
git reset --hard abc1234
```

This will reset the HEAD to `abc1234`, remove any staged changes, and discard all changes in your working directory. The changes are lost unless you have them backed up.

---

3. Amending the Last Commit

If you want to modify the most recent commit (e.g., changing the commit message or adding files), you can use the `git commit --amend` command. This is useful when you realize you forgot to stage a file or want to correct a message.

Command:

bash

```
git commit --amend
```

Example:

To amend the last commit's message:

bash

```
git commit --amend
```

Git will open your text editor to modify the commit message. You can also stage changes before amending if you want to add files to the previous commit.

---

4. Undoing the Last Commit (Without Affecting the Working Directory)

If you want to undo the last commit but keep the changes in your working directory (so you can rework them), you can use:

Command:

```bash
git reset --soft HEAD~1
```

Example:

To undo the last commit but keep your changes:

```bash
git reset --soft HEAD~1
```

This command will move the HEAD back by one commit but keep your changes in the staging area for potential re-commit.

Summary of Commands:

- **Revert a commit (create a new commit that undoes it):**

  ```bash
  git revert <commit-hash>
  ```

- **Soft reset (move HEAD to a previous commit and keep changes staged):**

  ```bash
  git reset --soft <commit-hash>
  ```

- **Mixed reset (move HEAD to a previous commit and unstage changes):**

  ```bash
  git reset --mixed <commit-hash>
  ```

- **Hard reset (move HEAD to a previous commit and discard all changes):**

  ```bash
  git reset --hard <commit-hash>
  ```

- **Amend the last commit (change commit message or add changes):**

```bash
git commit --amend
```

- **Undo the last commit without affecting the working directory:**

  ```bash
  git reset --soft HEAD~1
  ```

Would you like more information on any of these options or examples on how to handle specific scenarios?

git stash

Git Stash: Saving Changes Temporarily

`git stash` is a useful command that allows you to temporarily save your uncommitted changes (both staged and unstaged) and revert your working directory back to the state of the last commit. This is helpful when you need to quickly switch contexts (e.g., switch branches) without committing incomplete work.

Basic Usage

1. Stash Changes

To stash your current changes (both staged and unstaged) and revert your working directory to the last commit, use:

Command:

```bash
git stash
```

This will:

- Stash both staged and unstaged changes.
- Revert your working directory to the last commit, so you can safely switch branches or work on something else.

Example:

You're working on a feature but need to switch to another branch to fix an urgent bug. Instead of committing incomplete work, you can stash your changes:

```bash
git stash
```

Now you can switch to the other branch:

```
bash

git checkout bug-fix
```

2. Stashing Specific Changes

If you only want to stash specific files (e.g., staged changes or only unstaged changes), you can specify them:

Command for stashing staged changes:

```
bash

git stash --staged
```

Command for stashing specific files:

```
bash

git stash push <file1> <file2>
```

Example:

To stash only changes to `index.html`:

```
bash

git stash push index.html
```

This will stash only the changes to `index.html` and leave other changes in your working directory.

Viewing and Managing Stashes

3. List All Stashes

To see all the stashes you've saved:

Command:

```
bash

git stash list
```

This will display all stashes in the format:

```
pgsql

stash@{0}: WIP on main: 9f5d5db initial commit
stash@{1}: WIP on feature-x: 4b4d1c0 work on feature-x
```

4. View a Specific Stash

To view the changes in a specific stash (without applying them):

Command:

```bash
git stash show stash@{<index>}
```

For example, to see the changes in the first stash:

```bash
git stash show stash@{0}
```

To view more detailed changes, add the `-p` flag:

```bash
git stash show -p stash@{0}
```

5. Apply a Stash

To reapply the changes from a stash (without removing it from the stash list), use:

Command:

```bash
git stash apply stash@{<index>}
```

Example:

To apply the first stash (`stash@{0}`):

```bash
git stash apply stash@{0}
```

If you don't specify an index, Git will apply the most recent stash by default.

6. Pop a Stash

If you want to reapply a stash **and** remove it from the stash list, use `git stash pop`:

Command:

```bash
git stash pop stash@{<index>}
```

Example:

To pop the most recent stash:

```bash
git stash pop
```

If you want to pop a specific stash:

```bash
git stash pop stash@{0}
```

This will apply the changes and remove the stash from the list.

Additional Stash Options

7. Drop a Stash

If you no longer need a stash, you can delete it:

Command:

```bash
git stash drop stash@{<index>}
```

Example:

To drop the first stash:

```bash
git stash drop stash@{0}
```

8. Clear All Stashes

To remove all stashes, use the following command:

Command:

```bash
git stash clear
```

This will permanently delete all stashes, so use it with caution.

Summary of `git stash` Commands

- **Stash changes (both staged and unstaged):**

```bash
git stash
```

- **Stash staged changes:**

```bash
git stash --staged
```

- **Stash specific files:**

```bash
git stash push <file1> <file2>
```

- **List all stashes:**

```bash
git stash list
```

- **View a specific stash:**

```bash
git stash show stash@{<index>}
```

- **Apply a stash (without removing it from the list):**

```bash
git stash apply stash@{<index>}
```

- **Pop a stash (apply and remove it from the list):**

```bash
git stash pop stash@{<index>}
```

- **Drop a specific stash:**

```bash
git stash drop stash@{<index>}
```

- **Clear all stashes:**

```bash
git stash clear
```

Let me know if you need more examples or have any other questions about `git stash`!

how to reset push

If you've pushed commits to a remote repository and want to **reset** those commits, you can do so using `git reset`. However, since these commits have already been pushed to a remote, you need to be careful when using these commands, as they can rewrite commit history.

Here are the options for resetting pushed commits:

1. Reset to a Previous Commit and Push Changes (Force Push)

If you want to **reset** your local branch to a previous commit and push the changes to the remote (effectively removing the pushed commits), you can use the `git reset` command followed by a **force push** (`git push --force` or `git push -f`).

Command:

```bash
git reset --hard <commit-hash>
git push --force
```

Steps:

1. **Find the commit hash** you want to reset to (you can use `git log` to get the commit hash).
2. **Reset** your local branch to that commit:

   ```bash
   git reset --hard <commit-hash>
   ```

3. **Force push** the changes to the remote:

   ```bash
   git push --force
   ```

This will reset the local branch to the specified commit and then forcefully push the changes to the remote repository, effectively removing any commits that came after that commit in the remote branch.

Example:

You want to reset to commit `abc1234` and remove the most recent commit from the remote:

```bash
git reset --hard abc1234
git push --force
```

---

2. Reset to a Previous Commit and Push (Force Push with a Specific Branch)

If you want to reset a specific branch and push the changes, specify the branch name.

Command:

```bash
git reset --hard <commit-hash>
git push --force origin <branch-name>
```

Example:

To reset the `main` branch to commit `abc1234` and push the reset state to the remote:

```bash
git reset --hard abc1234
git push --force origin main
```

---

3. Using `git push --force-with-lease`

If you're working in a shared repository, it's safer to use `git push --force-with-lease` instead of `git push --force`. This prevents overwriting commits made by others after you last fetched or pulled.

Command:

```bash
git reset --hard <commit-hash>
git push --force-with-lease
```

Explanation:

- `git push --force-with-lease` checks if the remote branch has been updated by someone else before allowing the force push.
- This is a safer option to avoid overwriting others' changes in a collaborative project.

Example:

```bash
git reset --hard abc1234
git push --force-with-lease
```

---

4. Reverting Commits Instead of Resetting

If you're concerned about rewriting commit history, you can **revert** the commits instead of resetting. This creates a new commit that undoes the changes made in the previous commits, preserving history.

Command:

```
bash

git revert <commit-hash>^..<commit-hash>
```

This will create a new commit that undoes the changes between the two commit hashes.

Example:

To revert the last 3 commits:

```
bash

git revert HEAD~3..HEAD
git push origin main
```

This avoids using `git push --force`, preserving the commit history.

---

5. Handling Merge Conflicts After Resetting

When you reset and force-push, there could be conflicts if others have pushed changes to the branch. In that case, collaborators might need to **fetch** and **rebase** their local changes on top of the reset branch.

Steps for collaborators:

1. **Fetch the latest changes:**

   ```
   bash

   git fetch origin
   ```

2. **Rebase their local changes:**

   ```
   bash

   git rebase origin/main
   ```

3. **Resolve any conflicts** and then continue the rebase:

   ```
   bash

   git rebase --continue
   ```

4. **Push their changes** after resolving conflicts.

Summary of Commands:

- **Hard reset and force push:**

  ```bash
  git reset --hard <commit-hash>
  git push --force
  ```

- **Reset and force push to a specific branch:**

  ```bash
  git reset --hard <commit-hash>
  git push --force origin <branch-name>
  ```

- **Force push with a safer option:**

  ```bash
  git push --force-with-lease
  ```

- **Reverting commits instead of resetting (preserving history):**

  ```bash
  git revert <commit-hash>^..<commit-hash>
  git push origin <branch-name>
  ```

Would you like more information on any of these methods, or need help with a specific scenario?

how to reset add

If you've **added** files to the staging area using `` `git add` ``, but you haven't yet committed them, and you want to **unstage** the changes (i.e., remove them from the staging area), you can use the following Git commands.

1. Unstage All Changes

To unstage **all changes** you've added to the staging area (but keep the changes in your working directory):

Command:

```bash
git reset
```

Explanation:

This command will unstage all the files that you've added with `git add` but will keep the actual changes in the working directory. In other words, it resets the index (staging area) to match the current commit, without affecting your working directory.

Example:

After adding files with `git add .`:

```
bash

git reset
```

This will unstage all the files, and your changes will still exist in the working directory.

---

2. Unstage a Specific File

If you want to unstage a specific file instead of all the files, you can specify the file name:

Command:

```
bash

git reset <file-name>
```

Example:

If you staged `index.html` and want to unstage it:

```
bash

git reset index.html
```

This will unstage `index.html`, but the changes will remain in the working directory.

---

3. Unstage Files (Staged by `git add -p`)

If you staged changes interactively (using `git add -p`), and you want to unstage the changes, the `git reset` command works similarly.

Command:

```
bash

git reset
```

Or, to unstage a specific hunk:

```bash
git reset <file-name>
```

4. Undo All Changes (Unstaging and Reverting)

If you want to not only unstage the changes but also **discard the changes** in your working directory (i.e., revert to the last committed version of the file), you can use:

Command:

```bash
git checkout -- <file-name>
```

Example:

To discard changes made to `index.html`:

```bash
git checkout -- index.html
```

This will unstage the file and revert it to the version in the last commit.

Summary of Commands:

- **Unstage all changes:**

  ```bash
  git reset
  ```

- **Unstage a specific file:**

  ```bash
  git reset <file-name>
  ```

- **Unstage changes and revert to the last committed state (discard changes):**

  ```bash
  git checkout -- <file-name>
  ```

Let me know if you need more clarification or examples!

branch

In Git, **branches** are used to create isolated environments for working on different features, bug fixes, or experiments without affecting the main codebase (usually the `main` or `master` branch). Here's a breakdown of how to work with Git branches, including creating, switching, and managing them.

---

1. List All Branches

To view all the branches in your repository, including the current branch you're on:

Command:

```bash
git branch
```

This will show a list of local branches, with the current branch marked by an asterisk (`*`).

Example:

```bash
$ git branch
* main
  feature-x
  bug-fix
```

---

2. Create a New Branch

To create a new branch, use the `git branch` command followed by the name of the new branch:

Command:

```bash
git branch <branch-name>
```

Example:

To create a new branch called `feature-xyz`:

```bash
git branch feature-xyz
```

This will create the branch locally but won't switch to it. To start working on it, you'll need to switch to the branch.

---

3. Switch to a Branch

To switch between branches, use the `` `git checkout` `` command (or `` `git switch` `` in newer Git versions):

Command (older versions of Git):

```
bash

git checkout <branch-name>
```

Command (newer versions of Git):

```
bash

git switch <branch-name>
```

Example:

To switch to the `` `feature-xyz` `` branch:

```
bash

git checkout feature-xyz
```

or in newer Git versions:

```
bash

git switch feature-xyz
```

---

4. Create and Switch to a New Branch in One Command

To both create a new branch and immediately switch to it, use the `` `-b` `` flag with `` `git checkout` `` (or `` `git switch -c` `` in newer Git versions):

Command (older versions of Git):

```
bash

git checkout -b <branch-name>
```

Command (newer versions of Git):

```
bash

git switch -c <branch-name>
```

Example:

To create a new branch called `feature-xyz` and switch to it:

```
bash

git checkout -b feature-xyz
```

or in newer versions:

```
bash

git switch -c feature-xyz
```

---

5. Merge a Branch

Once you've made changes in a branch and want to bring those changes into another branch (usually `main` or `master`), you can merge the branch.

First, switch to the branch you want to merge into (e.g., `main`):

Command:

```
bash

git checkout main
```

Then, use the `git merge` command to merge the branch:

Command:

```
bash

git merge <branch-name>
```

Example:

To merge `feature-xyz` into `main`:

```
bash

git checkout main
git merge feature-xyz
```

This will bring the changes from `feature-xyz` into `main`. If there are any conflicts, Git will ask you to resolve them.

---

6. Delete a Branch

Once you're done with a branch, you can delete it (locally). If the branch has already been merged, it's safe to delete it.

Command:

```bash
git branch -d <branch-name>
```

Example:

To delete the `feature-xyz` branch after merging it:

```bash
git branch -d feature-xyz
```

If you want to force-delete a branch (even if it hasn't been merged), use `-D` instead:

```bash
git branch -D <branch-name>
```

Example (force delete):

```bash
git branch -D feature-xyz
```

---

7. Push a Branch to Remote

To share your local branch with others (push it to the remote repository), use the `git push` command.

Command:

```bash
git push origin <branch-name>
```

Example:

To push the `feature-xyz` branch to the remote repository:

```bash
git push origin feature-xyz
```

---

8. Track a Remote Branch Locally

If you want to work on a branch that already exists on a remote (but not locally), you can fetch it and check it out:

Command:

```bash
git checkout --track origin/<branch-name>
```

Example:

To fetch and check out a remote branch called `feature-xyz`:

```bash
git checkout --track origin/feature-xyz
```

---

9. Rebase a Branch (Optional)

A **rebase** allows you to reapply your commits on top of another branch, making the history linear and cleaner (instead of having a merge commit).

Command:

```bash
git rebase <branch-name>
```

Example:

To rebase `feature-xyz` onto `main`:

```bash
git checkout feature-xyz
git rebase main
```

If there are conflicts during the rebase, Git will stop and ask you to resolve them.

Summary of Branching Commands:

- **List branches:**

  ```bash
  git branch
  ```

- **Create a new branch:**

  ```bash
  git branch <branch-name>
  ```

- **Switch to a branch (older Git):**

  ```bash
  git checkout <branch-name>
  ```

- **Switch to a branch (newer Git):**

  ```bash
  git switch <branch-name>
  ```

- **Create and switch to a new branch (older Git):**

  ```bash
  git checkout -b <branch-name>
  ```

- **Create and switch to a new branch (newer Git):**

  ```bash
  git switch -c <branch-name>
  ```

- **Merge a branch into another:**

  ```bash
  git merge <branch-name>
  ```

- **Delete a local branch:**

  ```bash
  git branch -d <branch-name>
  ```

- **Push a branch to remote:**

  ```bash
  git push origin <branch-name>
  ```

- **Track a remote branch locally:**

  ```bash
  git checkout --track origin/<branch-name>
  ```

- **Rebase a branch onto another:**

```bash
git rebase <branch-name>
```

Let me know if you need more details on any of these commands or have any specific scenarios in mind!

other important git command for project

Here's a list of **important Git commands** that are commonly used in various project workflows. These commands cover key aspects such as managing history, working with remotes, dealing with conflicts, and more.

---

1. Check the Status of Your Repository

The `git status` command shows the current state of the working directory and the staging area. It tells you which changes have been staged, which are not, and which files aren't being tracked by Git.

Command:

```bash
git status
```

Example:

Shows the current status of all files:

```bash
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified:   index.html
        new file:   style.css
```

---

2. Add Files to Staging Area

The `git add` command is used to add changes in files to the staging area. After staging files, you can commit them to the repository.

Command:

```bash
git add <file-name>
```

To add all modified files:

```bash
git add .
```

Example:

To add a specific file:

```bash
git add index.html
```

To add all changes:

```bash
git add .
```

---

3. Commit Changes

The `git commit` command is used to commit staged changes to the local repository. Each commit is a snapshot of your project at a particular point in time.

Command:

```bash
git commit -m "Your commit message"
```

Example:

To commit your changes with a message:

```bash
git commit -m "Add new feature to handle user input"
```

---

4. View Commit History

To view the commit history for your repository, use `git log`. This shows a list of commits made in the current branch, along with their commit hashes, author, date, and commit message.

Command:

```
bash

git log
```

You can customize the log output:

- For a brief view (showing commit hashes and messages):

  ```
  bash

  git log --oneline
  ```

Example:

To view the log in a concise format:

```
bash

git log --oneline
```

---

5. Undo the Last Commit (Reset)

If you've made a commit and want to undo it (keeping the changes in your working directory), use the **`` `git reset` ``** command.

Command:

```
bash

git reset --soft HEAD~1
```

This command undoes the last commit but keeps the changes in the working directory (not staged).

To remove the last commit completely (including the changes):

```
bash

git reset --hard HEAD~1
```

Example:

To undo the last commit but keep changes:

```
bash

git reset --soft HEAD~1
```

---

6. Push Changes to Remote Repository

Once you've committed changes locally, you may want to push those changes to a remote repository (like GitHub or GitLab). Use `git push` for this.

Command:

```bash
git push origin <branch-name>
```

Example:

To push the `main` branch to the remote repository:

```bash
git push origin main
```

---

7. Pull Changes from Remote Repository

If you want to get the latest changes from a remote repository, use `git pull`. This command fetches the latest changes and merges them into your local branch.

Command:

```bash
git pull origin <branch-name>
```

Example:

To pull the latest changes from the `main` branch:

```bash
git pull origin main
```

---

8. Clone a Remote Repository

To clone a remote Git repository to your local machine, use `git clone`. This copies the entire repository, including all branches and history.

Command:

```bash
git clone <repository-url>
```

Example:

To clone a repository from GitHub:

```bash
git clone https://github.com/user/repository.git
```

---

9. Fetch Changes from Remote

If you want to fetch the latest changes from a remote repository without merging them automatically, use `git fetch`. This gets the latest commits from the remote but doesn't update your local working directory or current branch.

Command:

```bash
git fetch origin
```

Example:

```bash
git fetch origin
```

---

10. Create a Tag

Tags are used to mark specific points in the repository's history, usually for releases or milestones.

Command:

```bash
git tag <tag-name>
```

Example:

To create a tag named `v1.0.0`:

```bash
git tag v1.0.0
```

To push tags to the remote:

```bash
git push origin v1.0.0
```

---

11. Resolve Merge Conflicts

If there's a conflict between branches when merging, Git will mark the conflicted files. You'll need to manually resolve the conflict, then mark the files as resolved:

1. Open the conflicted files and resolve the conflict (look for conflict markers).
2. After resolving the conflict, add the resolved files:

   ```bash
   git add <file-name>
   ```

3. Commit the merge:

   ```bash
   git commit
   ```

---

12. Stash Changes Temporarily

If you need to switch branches but don't want to commit your current changes, you can stash them temporarily. This saves your changes and allows you to apply them later.

Command to stash:

```bash
git stash
```

Command to apply stashed changes:

```bash
git stash apply
```

Example:

To stash changes:

```bash
git stash
```

To apply the most recent stash:

```bash
git stash apply
```

---

13. Rebase Your Branch onto Another

Rebasing is a way to integrate changes from one branch into another. It’s commonly used to keep the commit history clean and linear.

Command:

```
bash

git rebase <branch-name>
```

Example:

To rebase the `feature-xyz` branch onto the `main` branch:

```
bash

git checkout feature-xyz
git rebase main
```

---

14. Interactive Rebase

Interactive rebase allows you to rewrite history, modify commits, and reorder them.

Command:

```
bash

git rebase -i <commit-hash>
```

Example:

To start an interactive rebase from 3 commits ago:

```
bash

git rebase -i HEAD~3
```

---

15. View Changes Between Commits

To see what changes have been made between two commits, use the `git diff` command.

Command:

```
bash

git diff <commit1> <commit2>
```

Example:

To view the differences between `` `commit1` `` and `` `commit2` ``:

```bash
git diff abc1234 def5678
```

Summary of Important Git Commands:

- **Check repository status:**

  ```bash
  git status
  ```

- **Add files to staging area:**

  ```bash
  git add <file-name>
  ```

- **Commit changes:**

  ```bash
  git commit -m "Commit message"
  ```

- **View commit history:**

  ```bash
  git log
  ```

- **Undo the last commit:**

  ```bash
  git reset --soft HEAD~1
  ```

- **Push changes to remote:**

  ```bash
  git push origin <branch-name>
  ```

- **Pull changes from remote:**

  ```bash
  git pull origin <branch-name>
  ```

- **Clone a remote repository:**

  ```bash
  git clone <repository-url>
  ```

- **Fetch changes from remote:**

  ```bash
  git fetch origin
  ```

- **Create a tag:**

  ```bash
  git tag <tag-name>
  ```

- **Resolve merge conflicts:**
  (Manually resolve conflicts and use `` `git add` `` and `` `git commit` `` to finalize)

- **Stash changes:**

  ```bash
  git stash
  ```

- **Rebase a branch onto another:**

  ```bash
  git rebase <branch-name>
  ```

- **Interactive rebase:**

  ```bash
  git rebase -i <commit-hash>
  ```

Let me know if you need further clarification on any command or Git concept!

ChatGPT can make mistakes. Check important info.