GitHub Workflow and Git Command Cheat Sheet

Contents

1 General Info

This guide uses a pull request (PR) workflow (standard in most organizations) so that:

- All changes are developed on separate branches (not on `main`).
- You open a PR for review.
- `main` is protected, and only reviewed/approved changes get merged.

2 Repository Model (Branches)

- `main`: Live/stable branch. Do *not* push directly here.
- Feature branches: created per task, like `dev/feature-short-name` or `fix/bug-short-name`.

3 Initial Setup (Done Only Once)

Before working with GitHub repositories, you should configure your identity and a few core behaviors. These settings tell Git who you are and how to handle line endings across operating systems (line ending behavior differs between Windows and UNIX-based operating systems).

3.1 Set your username and email

Every Git commit records an author name and email. These should match your GitHub account.

Set globally:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

This applies to all repositories on this machine and ensures commits are properly attributed on GitHub. If needed, you can set per-repository (overrides global):

```
git config user.name "Repo-Specific Name"
git config user.email "repo-specific@email.com"
```

3.2 Verify your configuration

To check what Git has stored:

```
git config --list
```

Or check individual values:

```
git config user.name
git config user.email
```

3.3 Configure Newline (Line Ending) Behavior

Different operating systems use different line endings:

- Linux / macOS: LF (\n)
- Windows: CRLF (\r\n)

To avoid noisy diffs and merge issues, configure Git to normalize line endings.

Recommended settings:

On macOS or Linux:

```
git config --global core.autocrlf input
```

On Windows:

```
git config --global core.autocrlf true
```

What this does:

- Ensures files are stored in the repository with consistent LF endings.
- Converts line endings appropriately for your local OS.
- Prevents "entire file changed" diffs caused only by newline differences.

3.4 Optional but recommended quality of life settings

Set default branch name to `main`:

```
git config --global init.defaultBranch main
```

Set a default editor (example: VS Code):

```
git config --global core.editor "code --wait"
```

3.5 What this does for you

Completing the above steps ensures

- Your commits are correctly attributed to you.
- Line endings are handled consistently across platforms.
- New repositories default to modern Git conventions.

4 Day-to-Day Workflow

To select a remote repository, execute the commands

```
git init
git remote add origin https://github.com/user/repo
```

which creates the local repo and adds a remote named origin pointing to the specified URL

Note: You only run the above two commands once per repository. After it's set, future pushes/pulls use `origin` automatically.

Now, for each task/feature/bugfix you implement, you will roughly follow this sequence:

1. Get Up-to-Date

```
git switch main
git pull
```

What this does:

- `git switch main`: moves you to the `main` branch locally.
- `git pull`: downloads and integrates the newest commits from GitHub into your local `main`.

2. Create (or resume) a branch for your work

There are two common cases:

- **New task:** create a new branch from the latest `main`.
- **Existing task:** you already have a branch from earlier and need to bring it up to date with `main`.

A) New task: create a new branch

```
git switch -c dev/feature-short-description
```

What this does:

- Creates a new branch off your current branch (`-c` means "create").
- Switches you to that new branch.

B) Existing task: resume an old branch and pull changes from `main`

1) Switch to your existing branch:

```
git switch dev/feature-short-description
```

2) Bring the latest `main` into your branch

Merge `main` into your branch

```
git pull origin main
```

What this does:

- Incorporates the newest commits from `main` into your feature branch.

Note: `dev/` is used in this example because it's a common naming convention. You can use anything you want.

3. Make changes and commit often

As you work, commit in small logical chunks.

```
git status
git add <files-you-changed>
git commit -m "Short, specific message describing the change"
```

What this does:

- `git status`: shows what changed, what is staged, and what is not.
- `git add`: stages files to be included in the next commit.
- `git commit`: records a snapshot of staged changes with a message.

4. Push your branch to gitHub

```
git push -u origin dev/feature-short-description
```

What this does:

- Uploads your branch to GitHub
- `-u` sets the upstream so future `git push` works without extra arguments.

It's basically saying "push my local branch to the remote called `origin`, and remember that relationship"

5. Open a PR

Open a PR on GitHub:

- **Base:** `main`
- **Compare:** your feature branch (`dev/feature-short-description`)

In the PR description, include:

- What you changed and why
- How to test it (exact steps/commands)
- Any screenshots/log output if relevant

6. Respond to any review feedback

If changes are requested, just keep working on the same branch:

```
// make edits
git add <files>
git commit -m "Address PR feedback: <short summary>"
git push
```

The PR updates automatically when you push.

7. Merge (maintainer does this)

After approval and any required checks pass, the maintainer merges the PR into `main`. A common approach for this is called the squash merge (to keep history clean).

5 Stashing (Temporarily Shelving Work)

Sometimes you need to **switch branches quickly** (like to pull updates, fix an urgent bug, or review another PR) but you have local edits that are not ready to commit. **Stashing** saves those edits aside temporarily and returns your working directory to a clean state.

5.1 General idea

A stash is a **temporary shelf** of your uncommitted changes (optionally including staged changes and/or untracked files) so you can safely change branches and come back later.

5.2 Common commands

```
// Save tracked changes (both staged + unstaged) onto a stash
git stash

// Save with a helpful label
git stash push -m "WIP: refactor parser"

// Include untracked files too (new files not yet added)
git stash -u

// Include EVERYTHING (including ignored files) -- rarely needed
git stash -a
```

5.3 Listing and inspecting stashes

```
// List stashes (most recent is stash@{0})
git stash list
```

```
// Show summary of what a stash contains
git stash show stash@{0}

// Show full patch/diff
git stash show -p stash@{0}
```

5.4 Restoring a stash

There are two common ways to restore:

- `pop`: apply the stash *and remove it* from the stash list.
- `apply`: apply the stash *but keep it* in the stash list (safer if you're not sure it will apply cleanly).

```
// Apply most recent stash and drop it from the stash list
git stash pop

// Apply most recent stash but keep it in the stash list
git stash apply

// Apply a specific stash
git stash apply stash@{2}
```

Note: Applying a stash can cause conflicts (just like a merge). If that happens, resolve conflicts, then `git add <files>` and continue working.

5.5 Dropping and clearing stashes

```
// Drop one stash (delete it)
git stash drop stash@{1}

// Delete all stashes (be careful!)
git stash clear
```

5.6 How stash is used

- If you need to switch branches and your changes are not ready: `git stash push -m "WIP: ..."`.
- Prefer `git stash apply` if you're worried about conflicts; once things look good, you can `git stash drop`.
- If your work **is** in a good state, prefer a real commit on your branch instead of stashing.

6 Undoing Undesired Changes

This section teaches how to recover safely from mistakes at three stages: *staged*, *committed*, and *pushed*. In general, prefer commands that do not rewrite history unless a maintainer explicitly

mentions to do so.

6.1 I staged a change I don't want (unstage)

Sometimes you accidentally stage the wrong file (or the right file but with too much).

Unstage a specific file (keep your edits):

```
git restore --staged <file>
```

Unstage everything (keep your edits):

```
git restore --staged .
```

If you want to discard the local edits too (be careful):

```
git restore <file>
```

Sanity check after doing this:

```
git status
git diff
git diff --staged
```

6.2 I committed something and I realize I want to undo that

First decide: do you want to **keep the changes** but redo the commit, or **throw away the changes** entirely?

A) Undo the commit but keep the changes (most common):

```
// Moves HEAD back 1 commit, but leaves your changes staged
git reset --soft HEAD~1
```

Now you can adjust what is staged, then recommit:

```
git restore --staged <file>   // if needed
git add <files>
git commit -m "Better message / corrected contents"
```

B) Undo the commit and unstage the changes (keep them as local edits):

```
// Leaves changes in your working directory (unstaged)
git reset HEAD~1
```

C) Undo the commit and discard the changes (dangerous):

```
// WARNING: loses work in that commit
git reset --hard HEAD~1
```

D) Only the commit message is wrong (no content changes):

```
git commit --amend
```

6.3 Big oops. I pushed something and that must be undone

Once commits are pushed to GitHub, assume other people might see them. The safest fix is usually to make a new commit that reverses the bad one:

```
// Find the bad commit hash
git log --oneline -n 10

// Make a new commit that reverses it
git revert <bad-commit-hash>

// Push the new "revert" commit
git push
```

If you need to undo a range of commits:

```
// Revert each commit in the range (creates multiple revert commits)
git revert <oldest-hash>^..<newest-hash>
git push
```

Note: History rewrite (almost always avoid): commands like `git reset --hard` followed by `git push --force` (or `--force-with-lease`) rewrite history and can break other people's work. Only do this if a maintainer explicitly asks you to, and only on your own feature branch.

7 Checking Out a Specific Commit (Commit Hash) and Detached HEAD

Sometimes you need to look at (or temporarily work from) an exact commit instead of a branch tip. This is common when debugging, reproducing a past result, or reviewing an older version.

7.1 Switch to a commit by hash (read-only / inspection)

1) Find the commit hash:

```
git log --oneline -n 20
```

2) Switch to that commit:

```
git switch --detach <commit-hash>
// older equivalent:
// git checkout <commit-hash>
```

At this point you are in a *detached HEAD* state: you're pointing at a commit directly, not a branch name.

3) Go back to a branch when done:

```
git switch main
```

7.2 Detached HEAD: keep working by creating a new branch (recommended)

If you want to make commits starting from that commit, create a branch right away so your new commits are anchored to a named branch (not floating).

```
// You're currently detached at <commit-hash>
git switch -c dev/from-<short-desc>

// now commit normally
git status
git add <files>
git commit -m "Continue work from <commit-hash>"
```

7.3 If you already made commits while detached

If you accidentally committed while detached, you can still save that work by creating a branch that points at your current commit.

```
// While still on the detached HEAD (after your commit(s))
git switch -c dev/recover-detached-work

// optional: push it so it's safely on GitHub
git push -u origin dev/recover-detached-work
```

Sanity checks (useful if you're unsure where you are):

```
git status
git log --oneline -n 10 --decorate
git branch --show-current
```

Note: If you lose track of a commit you made, `git reflog` can usually find it.

8 Merge Conflicts

8.1 What is a conflict?

A merge conflict happens when Git cannot automatically combine changes because the same part of the same file was modified in two different ways. Git stops and asks a human to decide what the final version should be.

Conflicts are normal and expected in collaborative work. They are *not* errors or signs that something went wrong.

8.2 When do conflicts occur?

Conflicts typically occur during:

- `git pull` (which fetches + merges by default)
- Merging a branch or a pull request
- Rebasing a branch onto a newer base
- Applying a stash (`git stash apply` or `pop`)

Common situations that cause conflicts:

- Two people edited the same lines of a file.
- One person deleted code that another person modified.
- A long-lived branch drifted far from `main`.

8.3 What does Git do during a conflict?

When a conflict occurs:

- Git pauses the operation.
- Files with conflicts are marked as unmerged.
- Git inserts conflict markers into the file.

Example conflict markers inside a file:

```
<<<<<<< HEAD
current branch version
=======
incoming branch version
>>>>>>> origin/main
```

8.4 Step-by-step to resolve a conflict

The goal with this is to decide what the final code should look like, remove markers, and tell Git you are done.

1. See What Is Conflicted

```
git status
```

Git will list files that need conflict resolution.

2. Open the Conflicted File(s)

Open each conflicted file in your editor and look for:

- `<<<<<<<`
- `=======`
- `>>>>>>>`

3. Decide the Correct Final Version

You may:

- Keep **your version**
- Keep **their version**
- **Combine both**
- Rewrite entirely (often the cleanest option)

Important: Remove *all* conflict markers before continuing.

4. Mark the Conflict as Resolved

After editing:

```
git add <conflicted-file>
```

Repeat for all conflicted files.

5. Finish the Operation

- If this was a merge or pull:

  ```
  git commit
  ```

- If this was a rebase:

```
git rebase --continue
```

8.5 If something feels wrong

You can safely stop and ask for help. Before doing so:

```
git status
git log --oneline -n 10
```

Paste the output and explain what you were trying to do.

8.6 Best practices to minimize conflicts

- Pull from `main` before starting new work.
- Keep branches short-lived and PRs small.
- Commit logically and frequently.
- Avoid editing large shared files unless necessary.

9 Bonus: Ignoring Filetypes That Were Already Committed

Key idea: `.gitignore` only prevents *new/untracked* files from being committed. If a file was already committed in a previous version, adding it to `.gitignore` will *not* stop Git from tracking it automatically.

9.1 Recommended Fix (Stop Tracking, Keep Files Locally)

Use this when you want the files to remain on your computer but be removed from the repository going forward.

1. **Add patterns to `.gitignore`** (examples):

```
// common examples
*.log
*.tmp
*.env
```

2. **Remove the files from the Git index (cached) without deleting them locally:**

```
// remove a specific file from tracking (keeps it on disk)
git rm --cached path/to/file.log
```

3. **Commit and push the change:**

```
git status
git commit -m "Stop tracking ignored files"
git push
```

What this does:

- The file(s) will be deleted from the repository in the next commit.
- The file(s) remain on your local machine.
- Future changes to those files will be ignored because of `.gitignore`.

9.2 If There Are Many Files Already Tracked

If you have a lot of already-tracked files that should now be ignored, you can clear the index and re-stage only what is not ignored.

```
// ensures .gitignore is applied to the entire repo
git rm -r --cached .
git add .
git status
git commit -m "Rebuild index to apply .gitignore"
git push
```

Note: This does *not* delete your working files, but it may stage many changes. Always check `git status` before committing.

9.3 Sensitive Data Warning (History Rewrite Needed)

If the already-committed files contain **secrets** (API keys, passwords, tokens), removing them with `git rm --cached` is *not enough* because the secret remains in Git history. In that case, notify the maintainer immediately and rotate the secret. Removing it from history requires a history rewrite (advanced and disruptive), and should be done by the maintainer.

A Command Quick Reference

This section is a quick lookup for common commands. Some of these aren't discussed in this writeup, but will come up as you use Git, so they are included here

A.1 Inspecting State

- `git status`: See changed files, staged files, and branch info.
- `git diff`: See unstaged changes.
- `git diff --staged`: See staged changes.
- `git log`: View commit history on the current branch.
- `git log -n 5`: Show the last 5 commits from `HEAD`.
- `git log --all`: Show commits reachable from all branches.
- `git log --oneline --decorate --graph --all -n 20`: Compact visual history of all branches.
- `git show <commit>`: Inspect a specific commit.

- `git blame <file>`: See who last changed each line of a file.

A.2 Branching and Switching

- `git branch`: List local branches.
- `git branch -a`: List local and remote branches.
- `git switch <branch>`: Switch branches.
- `git switch -c <new-branch>`: Create and switch to a new branch.
- `git branch -D <branch>`: Delete a local branch (force).
- `git branch --merged`: Show branches already merged.

A.3 Pulling and Pushing

- `git pull`: Get newest changes for the current branch.
- `git pull --rebase`: Pull while keeping a linear history.
- `git fetch`: Download new commits without merging them.
- `git push`: Upload commits on your current branch.
- `git push -u origin <branch>`: First push of a new branch (sets upstream).

A.4 Staging and Committing

- `git add <file>`: Stage one file.
- `git add .`: Stage all changes in the current directory tree.
- `git restore --staged <file>`: Unstage a file (keeps changes).
- `git commit -m "..."`: Commit staged changes with a message.
- `git commit --amend`: Modify the most recent commit.

A.5 Undoing and Recovering

- `git restore <file>`: Discard unstaged changes to a file.
- `git reset HEAD <file>`: Unstage a file (older alternative).
- `git reset --soft HEAD^`: Undo last commit, keep changes staged.
- `git reset --hard HEAD^`: Undo last commit and discard changes.
- `git reflog`: Recover lost commits or branches.

A.6 Cleaning and Maintenance

- `git clean -fdn`: Preview removal of untracked files and directories.
- `git clean -fd`: Delete untracked files and directories.
- `git clean -fxd`: Also remove ignored files (dangerous).
- `git gc`: Run garbage collection to optimize repository.

A.7 Remote Repositories

- `git remote -v`: List remotes and their URLs.
- `git remote show origin`: Inspect remote branches and tracking info.
- `git fetch --prune`: Remove references to deleted remote branches.

A.8 Stashing Work

- `git stash`: Save uncommitted changes temporarily.
- `git stash list`: Show saved stashes.
- `git stash pop`: Reapply and remove the latest stash.
- `git stash drop`: Delete a stash without applying it.

B Typical Naming Conventions

- Branch: `dev/feature-<short-name>` or `fix/<short-name>`
- Commit message: `Verb + object`, like `Add telemetry parsing`, `Fix null pointer in parser`
- PR title: similar to commit message but more descriptive if needed

C Simple Solo Project Workflow

This appendix section describes a simple Git workflow for a solo project where you are the only developer and pushing to `main` is totally fine. The goal is to keep history understandable, avoid accidental mistakes, and make it easy to recover if (or if you're like me, when) something goes wrong.

C.1 Model: One Branch (`main`) + Frequent Small Commits

In a solo repo, the simplest model is:

- You do your work directly on `main`.
- You commit early and often in small logical chunks.
- You push regularly so GitHub has a backup of your work.

C.2 Initial Setup (Done Once per Repo)

If you are creating a brand-new project:

Create a new repository locally, then connect to GitHub:

```
mkdir my-project
cd my-project
git init
git branch -M main

// Add your project files here (or create them)
git add .
git commit -m "Initial commit"

// Connect to your GitHub repo and push
git remote add origin https://github.com/user/repo.git
git push -u origin main
```

Or Clone an existing GitHub repository:

```
git clone https://github.com/user/repo.git
cd repo
```

The Core Workflow

Most of your time will be spent repeating this loop.

1. Start by syncing with GitHub

Even solo, you may switch machines or edit on GitHub occasionally, so pull first:

```
git switch main
git pull
```

2. Make your changes

Edit files as needed. Periodically check what changed:

```
git status
git diff
```

3. Stage the intended changes

Stage only what you want in the next snapshot.

Stage a few specific files:

```
git add file1 file2
```

Or stage everything in the current folder tree:

```
git add .
```

4. Commit (small + descriptive)

```
git commit -m "Implement <feature> / fix <bug> / update <doc>"
```

Helpful habit: prefer many small commits over one giant commit. It makes undoing mistakes much easier.

5. Push to GitHub

```
git push
```

C.3 Optional "Safety branch" for risky changes

If you are about to do something risky (big refactor, dependency upgrade), you can temporarily use a branch even in a solo repo:

```
git switch main
git pull
git switch -c safety/big-change

// work + commit as normal
git add .
git commit -m "WIP: big refactor checkpoint"
git push -u origin safety/big-change
```

When done, merge back locally, then push:

```
git switch main
git merge safety/big-change
git push
```

Cleanup:

```
git branch -d safety/big-change
git push origin --delete safety/big-change
```