Git Notes for Hackathon 2026

Why Git Gud with Git?

Imagine this:

- You have a group of people all working on the same codebase.
- Someone changes a file that other people depend on.
- The project suddenly stops working, and nobody knows why.
- Stress rises, time is wasted, and progress stalls.

Git does not magically prevent bugs—but it **makes changes traceable and reversible**:

- You can quickly see **what changed**, **when**, and **by whom**.
- You can compare versions and isolate breaking changes.
- You can revert to a working state without panic.

In short: when a team uses Git well, “mystery breakage” becomes a solvable, calm problem.

Why You REALLY, REALLY Should Git Gud with Git

- **Jobs:** Most software and engineering teams use Git (or a Git-based workflow). Being comfortable with it is expected.
- **Resumes/portfolios:** A clean GitHub (or similar) project with a README and meaningful commits is a strong signal of competence—especially team projects that show collaboration habits.
- **Backups:** If your laptop dies but your work is pushed to a remote (GitHub/GitLab/etc.), you have not lost much.
- **Multiple computers:** Keep work synchronized across home/desktop/lab machines by pushing and pulling.
- **Rolling back:** If something breaks, Git makes it straightforward to return to a previous commit where things worked.
- **Root-cause debugging:** Tools like `git log`, `git diff`, and `git blame` help you find *which change introduced a behavior* and the context for it.
  (Use this for diagnosis and learning, not finger-pointing.)

What Is Git?

Git is a distributed version control system. It tracks your project history as a series of commits and supports branching/merging so people can work in parallel.

Key points:

- Git runs **locally** on your machine. You can commit even without internet.
- Teams usually add a **remote repository** (a shared copy) so everyone can push/pull changes.

Git vs GitHub (not the same thing!)

- **Git**: the version control software (open source, runs locally).
- **GitHub**: a hosting platform built around Git (remote repos + pull requests + issues + CI integrations).
- Other common Git hosting platforms:
  - **GitLab** (often chosen for “all-in-one” DevSecOps + strong self-hosting options)
  - **Bitbucket** (often chosen by teams already using Atlassian tools like Jira)

Core Git Terms (and what they actually mean)

Repository (“repo”)

- A repo is your project tracked by Git.
- You usually have:
  - a **local repo** on your machine
  - a **remote repo** on a server (e.g., GitHub), often named `origin`

Working directory → Staging area → Commit history

- **Working directory**: the files you are editing.
- **Staging area (index)**: the set of changes you’ve selected to include in the next commit.
- **Commit**: a saved snapshot of your staged changes, with a message and metadata.

Clone vs init (starting a repo)

- `git clone`: copy an existing remote repo onto your machine (history, branches, everything).
- `git init`: start a brand-new Git repo in a folder you already have locally.

Branches

- A branch is a **lightweight line of development** (technically, a movable pointer to a commit).
- Practically: branches let you work on a feature/fix without destabilizing `main`.
- Branches reduce chaos, but merges can still produce **conflicts** if changes overlap.

Merge

- **Merge** combines changes from one branch into another.
- If the same lines were edited differently, Git may require you to resolve a **merge conflict**.

Push / fetch / pull

- **Push**: send your local commits to the remote repository.

- **Fetch**: download updates from the remote *without changing your working files*. (It updates your remote-tracking branches like `origin/main`.)
- **Pull**: fetch + integrate into your current branch (often by merging; sometimes rebasing depending on configuration).
  Translation: `pull` can modify your branch, so it's more "active" than fetch.

Pull requests vs merge requests

- These are the *platform workflow* for merging branch changes into another branch.
- GitHub calls them **Pull Requests (PRs)**.
- GitLab calls them **Merge Requests (MRs)**.

Checkout / switch

- "Checkout" is the older term you'll still see.
- Newer Git uses:
  - `git switch` to change branches
  - `git restore` to restore files
    VS Code handles most of this through its UI.

Stash

- **Stash** temporarily shelves uncommitted changes so you can switch branches cleanly.
- Important correction: switching branches does **not** automatically "delete" your changes. Git usually prevents unsafe switches.
  Stash is simply a convenient "parking spot" when you need a clean working directory.
- You can later re-apply changes with `stash pop` or `stash apply`.

The Git Workflow (Windows + VS Code, team-friendly)

This is a simple workflow that works very well for student teams.

1) Create the remote repository

- Create a GitHub repository (public or private—choose based on course/team needs).
- Invite collaborators (or use a GitHub Organization for larger teams).

2) Protect `main` (strongly recommended for team projects)

Goal: nobody pushes directly to `main`; changes enter via PRs.

- GitHub: **Settings → Rules → Rulesets**
- Target `main`
- Enable: **Require a pull request before merging**
- Consider adding:

- **1 approval** (2 for larger teams)
- **Dismiss stale approvals**
- **Require conversation resolution**
- (If you have CI) **Require status checks**

3) Track work with Issues (optional, but excellent practice)

- Create Issues for features/bugs/tasks.
- Assign them, label them, discuss them.
- You can create a branch from an issue in GitHub **or** create a branch in VS Code and link the PR to the issue later.

4) Start work on a branch

- In VS Code (or terminal), create/switch to a new branch, e.g.:
  - `feature/login-ui`
  - `fix/crash-on-start`
- If you accidentally start coding on `main`, you have two common fixes:
  1. **Create a new branch** and keep your changes there, or
  2. **Stash**, switch to the right branch, then apply the stash.

5) Commit early and coherently

- Stage changes (VS Code Source Control panel).
- Commit with a meaningful message:
  - “Add input validation for …”
  - “Fix null pointer in …”
  - “Refactor parsing logic …”
- Push occasionally so your work is not trapped on one machine.

6) Open a Pull Request (PR)

- PRs are how changes get into `main`.
- You may open a PR early as a **Draft** to show progress, but you should **merge only when**:
  - the change is coherent and reviewed
  - tests/checks pass (if you have them)
  - it keeps `main` in a working state

7) Keep `main` healthy

- Treat `main` as “always runnable / always demo-able.”
- Before merging, it is often wise to sync your branch with the latest `main` (so you discover conflicts before your teammates do).