Git Recovery Cheat Sheet - For Steph

1. See your history

git log --oneline --graph --all --decorate

alias git-time="git log --oneline --graph --all --decorate"

2. Explore a past version safely

git checkout <commit-hash>

git checkout -b debug-old-state # if you want to experiment

3. Undo your last commit, keep the code

git reset --soft HEAD~1 # Keeps code, removes last commit

4. Undo the last commit AND the code

git reset --hard HEAD~1 # Danger zone. Deletes code and commit

5. Undo local changes (before commit)

git checkout -- path/to/file # Undo changes to one file

git reset --hard # Undo all local changes

6. Stash work before switching or cleaning

git stash # Save work temporarily

git stash pop # Restore it later

7. Revert a specific commit

git revert <commit-hash> # Creates a new commit that undoes it

8. Clone an old version as a side experiment

git checkout abc123

git checkout -b experiment-old-logic

9. Delete a branch (after merge or experiment)

git branch -d my-feature-branch # Safe delete

git branch -D my-feature-branch # Force delete

10. Back up your current state (the lazy way)

git branch backup-before-i-break-it

Pro Tip: Add these aliases

alias git-undo='git reset --soft HEAD~1'

alias git-nuke='git reset --hard HEAD~1'

alias git-time='git log --oneline --graph --all --decorate'

alias git-panic='git stash && echo "[lifesaver emoji] Code stashed. You're safe."'

11. Understanding Pull and Push

Think of Git like a two-way street:

git pull (down Remote -> Local): Brings latest code from GitHub to your machine.

git push (up Local -> Remote): Sends your changes from your machine to GitHub.

git commit = Save (local)

git push = Upload (remote)

git pull = Sync (fetch + merge)

Examples:

```
git pull origin main  # Download and merge latest main branch
git push origin main  # Upload your commits to GitHub
```

Common Problems:

- Can't push? Run git pull first.
- Code messed up after pull? Stash changes or back up first.
- Wrong branch? Revert on GitHub or push to the right one.

12. Dealing with Merge Conflicts

Conflicts happen when Git can't auto-merge because the same lines of code were changed in two places.

Typical error:

```
CONFLICT (content): Merge conflict in app.js
Automatic merge failed; fix conflicts and then commit the result.
```

Git marks conflicts like this:

<<<<<<< HEAD

your version

=======

their version

>>>>>>> origin/main

To resolve:

1. Manually edit the file to keep what you want.

2. Save the file.

3. Run: git add <filename>

4. Commit the resolution: git commit -m "Resolve merge conflict"

Tips:

- Use git status to see conflicted files.

- VS Code shows conflicts visually.

- git mergetool can help (if configured).

Best Practices:

- git pull before starting work

- Keep commits small and focused

- Use feature branches to isolate work

Think of conflicts as code conversations - Git is just asking you to decide what story to keep.

13. Open vs Closed in Git and GitHub

These terms mostly apply to GitHub workflows, not files or commits directly.

Pull Requests:

- Open: Proposed changes under review, not yet merged.

- Closed: Either merged ([merged]) or discarded ([discarded]).

Issues:

- Open: Active discussion (bug, task, feature).

- Closed: Resolved or rejected.

Branches:

- Open: Still exists, possibly in progress.
- Closed: Typically means merged and deleted.

Files and Commits:

- These aren't considered open/closed.
- Commits exist permanently in the log unless you rewrite history.

Summary:

- Pull Requests -> Merged or Not
- Issues -> Resolved or Not
- Branches -> Still active or cleaned up
- Files/Commits -> Always tracked, never 'closed'