Core Concepts & Setup

1. What is Git?

- **Answer:** Git is a free, open-source **Distributed Version Control System (DVCS)**. It tracks changes to files, allows multiple developers to work on the same project simultaneously without interfering with each other's work, and enables you to revert to previous versions of the project if needed.

2. What's the difference between Git and GitHub?

- **Answer:** Git is the version control *tool* itself—the software that runs on your local machine. GitHub is a web-based *hosting service* for Git repositories. It provides a central place to store your Git repositories and offers collaboration features like pull requests, issue tracking, and project management tools. Other alternatives are GitLab and Bitbucket.

3. What is a repository (repo)?

- **Answer:** A Git repository is a storage location for your project, which tracks and saves the history of all changes made to the files. It contains all the project files and the revision history. There are two types: a **local repository** on your computer and a **remote repository** on a server (like GitHub).

4. Explain the basic Git workflow.

- **Answer:** The basic workflow involves these areas:
    1. **Working Directory:** You modify files in your working directory.
    2. **Staging Area (Index):** You selectively add the files you want to commit to the staging area (git add).
    3. **Local Repository:** You permanently commit the staged files to your local repository (git commit).
       (Optionally) **Remote Repository:** You push your commits from the local repo to a remote repo to share them (git push).

Basic Commands & Operations

5. What is git clone **used for?**

- **Answer:** git clone <url> is used to create a copy of an existing remote repository on your local machine. It downloads the entire project history and sets up a connection (called origin) to the remote repository.

6. What do the git add **and** git commit **commands do?**

- **Answer:**
    - git add <file> or git add .: Moves changes from the working directory to the staging area. It prepares a snapshot of the changes.
    - git commit -m "message": Takes the snapshot from the staging area and permanently saves it to the local repository. The message should be a concise description of the changes.

7. What is the purpose of git status**?**

- **Answer:** git status shows the current state of the working directory and the staging area. It displays which files are modified, which are staged (ready to be committed), and which are not being tracked by Git.

8. What does git log **show?**

- **Answer:** git log displays the commit history for the current branch. It shows the commit hash, author, date, and commit message for each commit.

9. How do you push changes to a remote repository?

- **Answer:** The command is git push <remote-name> <branch-name>. The most common usage is git push origin main, which pushes your local main branch commits to the origin remote.

10. How do you update your local repository with changes from the remote?

- **Answer:** You use git pull. This command is a combination of git fetch (which downloads the changes from the remote) and git merge (which merges those changes into your current local branch).

Branching & Merging

11. What is a branch in Git?

- **Answer:** A branch is a lightweight movable pointer to a commit. It allows you to create a separate line of development. You can work on new features or bug fixes in an isolated environment without affecting the main codebase (e.g., the main branch).

12. How do you create a new branch and switch to it?

- **Answer:** git checkout -b <branch-name>. This is a shortcut for two commands: git branch <branch-name> (to create it) and git checkout <branch-name> (to switch to it). In newer Git versions, you can also use git switch -c <branch-name>.

13. How do you merge a branch?

- **Answer:** First, switch to the branch you want to merge *into* (e.g., main). Then, run git merge <branch-to-merge>.

14. What is a merge conflict and how do you resolve it?

- **Answer:** A conflict occurs when Git cannot automatically merge changes because conflicting changes were made to the same part of the same file. Git will mark the file as conflicted. To resolve it, you must:
    1. Open the file and manually edit it to choose which changes to keep (look for the <<<<<<<, =======, and >>>>>>> markers).
    2. Remove the conflict markers.
    3. Stage the resolved file (git add <file>).
    4. Commit the changes (git commit).

15. What is the difference between git merge **and** git rebase**?**

- **Answer:** Both integrate changes from one branch into another.
    - **Merge:** Creates a new **merge commit** that ties together the histories of both branches. The branch history is preserved exactly as it happened.
    - **Rebase:** Moves the entire feature branch to begin *on the tip* of the target branch, effectively rewriting the project history by creating new commits for each original commit. It results in a cleaner, linear history.

Undoing Changes

16. How do you discard changes in your working directory?

- **Answer:** git checkout -- <file> discards changes for a specific file, reverting it to the state of the last commit. A more modern command is git restore <file>.

17. How do you unstage a file that you've git add**ed?**

- **Answer:** git reset HEAD <file> will unstage the file but keep the changes in your working directory. The modern equivalent is git restore --staged <file>.

18. What is the difference between git reset **and** git revert**?**

- **Answer:** This is a crucial difference.
    - git reset**:** Moves the **branch pointer backwards** to an older commit, effectively erasing commits from the current branch's history. **Use with caution on shared branches.**
    - git revert**:** Creates a **new commit** that undoes the changes made in a previous commit. This is a **safe** way to undo changes in a shared or public repository because it doesn't rewrite history.

Remote Repositories

19. What is origin**?**

- **Answer:** origin is the default conventional short name (alias) for the URL of the remote repository from which you cloned your local repository. It's a pointer to the upstream repository.

20. What is the command to view the remote repositories connected to your local repo?

- **Answer:** git remote -v. The -v (verbose) flag shows both the fetch and push URLs.

Advanced Basics

21. What is a .gitignore **file?**

- **Answer:** A .gitignore file is a text file that tells Git which files or directories to ignore in a project. You typically use it to avoid committing auto-generated files, local configuration files, dependency folders (like node_modules), or secrets (like API keys).

22. What is a commit hash (SHA)?

- **Answer:** A commit hash is a unique 40-character checksum (e.g., e0a9210d72b0e5a7e87d6d70a12e5b5a6a9a1b2c) generated by Git for each commit. It serves as a unique ID for that commit, based on its content, author, date, and previous commits. You can often use just the first 7 characters to reference it.

23. What is git fetch**?**

- **Answer:** git fetch downloads commits, files, and refs from a remote repository into your local repo. **It does not merge them into your current branch.** It lets you see

what others have been working on without integrating the changes immediately. git pull is essentially git fetch followed by git merge.

24. What is a "detached HEAD" state?

- **Answer:** A "detached HEAD" means your HEAD pointer (which points to the current snapshot) is pointing directly to a commit instead of a branch. This happens if you check out a specific commit, tag, or remote branch. You can look around and make experimental changes, but you should create a new branch if you want to save them.

25. What is a Pull Request (or Merge Request)?

- **Answer:** A Pull Request (PR) is a feature of collaboration platforms like GitHub and GitLab (where it's called a Merge Request). It's a **request** to merge changes from one branch into another. It initiates a discussion, code review, and automated testing *before* the code is actually merged. It is **not a core Git command** but a key part of the modern collaboration workflow.

Here are 25 intermediate-level Git interview questions and answers that delve deeper into Git's mechanics, branching strategies, and problem-solving scenarios.

Branching, Merging, and Rebasing

1. Explain the difference between a fast-forward merge and a three-way merge.

- **Answer:** A **fast-forward merge** happens when the target branch (e.g., main) has no new commits since the feature branch was created. Git simply moves the branch pointer forward along the line of commits. A **three-way merge** is necessary when both branches have diverged. Git creates a new **merge commit** that has two parents, combining the histories of both branches.

2. What is the purpose of git rebase --interactive **(interactive rebase)?**

- **Answer:** Interactive rebase allows you to rewrite history with precision. You can squash multiple commits into one, reorder commits, edit commit messages, split commits, or drop commits entirely. It's often used to clean up a feature branch's history before merging it.

3. When would you avoid using git rebase**?**

- **Answer:** You should avoid rebasing commits that have been pushed to a **public or shared repository**. Rebasing rewrites history, and if others have based their work on

the original commits, it will cause major synchronization conflicts and break their history.

4. What is a merge conflict and what are the steps to resolve it?

- **Answer:** A conflict occurs when Git cannot automatically reconcile changes from different branches (e.g., two people changed the same part of the same file). Steps to resolve:
    1. Identify conflicted files with git status.
    2. Open the files and manually edit them, removing the conflict markers (<<<<<<<, =======, >>>>>>>).
    3. After resolving, stage the fixed files with git add <file>.
    4. Complete the resolution by committing with git commit. Git will pre-fill a merge commit message.

5. Explain the concept of "Squash and Merge." What is its benefit?

- **Answer:** "Squash and Merge" is a strategy where all commits from a feature branch are combined ("squashed") into a single commit before being merged into the main branch (e.g., main or develop). The benefit is a much cleaner, more linear project history on the main branch, showing only complete features instead of every incremental "WIP" commit.

Advanced Operations & History

6. What is the difference between git cherry-pick **and** git rebase**?**

- **Answer:** Both apply commits from one branch to another.
    - git cherry-pick <commit-hash> applies the changes from a **specific commit** to your current branch. It's useful for grabbing a single bug fix.
    - git rebase <branch> moves the **entire current branch** to start on the tip of the target branch, replaying all its commits.

7. How would you find which commit introduced a specific bug? (e.g., a regression)

- **Answer:** Use git bisect. This command performs a binary search through your history to find the offending commit.
    1. Start with git bisect start.
    2. Mark a known bad commit: git bisect bad HEAD.

3. Mark a known good commit: git bisect good <old-commit-hash>.
4. Git will checkout a midpoint commit; you test it and mark it as good or bad.
5. Git repeats until it pinpoints the first bad commit.

8. What is the "reflog" and when would you use it?

- **Answer:** The Reference Log (reflog) is a safety net. It's a local history of all actions that changed the tips of branches and HEAD (e.g., checkouts, commits, merges, resets). You use it to recover lost commits or branches if you accidentally reset or delete them. Use git reflog to view it and then reset to a specific entry.

9. Explain the three main types of git reset **(**--soft**,** --mixed**,** --hard**).**

- **Answer:** They differ in how they affect the **Staging Index** and **Working Directory**.
    - git reset --soft <commit>: Moves the **branch pointer** back to the commit. Staging Index and Working Directory are **unchanged**. Great for fixing the last commit.
    - git reset --mixed <commit> (default): Moves the branch pointer and **resets the Staging Index** to match that commit. Working Directory is **unchanged**. This "unstages" changes.
    - git reset --hard <commit>: **DANGER**. Moves the branch pointer, resets the Staging Index, **and resets the Working Directory**. All changes since that commit are permanently lost. Use with extreme caution.

10. How does git stash **work and what are some use cases?**

- **Answer:** git stash temporarily shelves (or "stashes") changes in your working directory, giving you a clean state. Use cases:
    - Quickly context-switch to work on a urgent bug fix without committing half-done work.
    - Pulling latest changes from remote when you have local, uncommitted changes.
    - Use git stash pop to re-apply the stashed changes later.

Internal Mechanics & Configuration

11. Briefly describe what happens internally when you run git commit**.**

- **Answer:** Git:

1. Stores a **blob** for each file in the staging area.
2. Creates a **tree** object that records the structure of the repository (which blobs correspond to which filenames).
3. Creates a **commit** object that contains the author, message, a pointer to the tree, and a pointer to its parent commit(s).
4. Updates the current branch reference to point to this new commit object.

12. What are Git hooks? Can you name a few examples?

- **Answer:** Git hooks are scripts that run automatically before or after specific Git commands (like commit, push, receive). They are used to enforce policies, run tests, or trigger deployments.
    - **Examples:** pre-commit (run linters/tests before a commit is allowed), pre-push (run tests before pushing), post-receive (deploy code to a server after a push to remote).

13. How do you amend the most recent commit?

- **Answer:** git commit --amend. This allows you to:
    - Change the commit message.
    - Add forgotten files to the commit (stage them first, then run git commit --amend).
    - It replaces the previous commit with a new one, so avoid amending public commits.

14. What is the difference between git pull **and** git fetch**?**

- **Answer:** git fetch is the safe, "look before you leap" command. It only **downloads** new data from the remote (commits, branches, tags) and updates your remote-tracking branches (e.g., origin/main), but it does **not** merge them into your current branch. git pull = git fetch + git merge. It fetches and then immediately tries to merge the remote changes into your current branch.

15. What are remote-tracking branches (e.g., origin/main**)?**

- **Answer:** They are local pointers that represent the state of branches in the remote repository *the last time you communicated with it* (via fetch or pull). You cannot commit to them directly; they are Git's way of letting you see what happened on the remote without having to connect to it.

Workflow & Collaboration

16. Describe the Git Flow branching model.

- **Answer:** A robust model with several long-lived branches:
    - main: Represents production-ready code.
    - develop: Represents the latest delivered development changes for the next release.
    - **Supporting branches:** feature/* (branched from develop), release/* (branched from develop for final testing), hotfix/* (branched from main for urgent production fixes).

17. What is a Trunk-Based Development workflow?

- **Answer:** A simpler alternative to Git Flow where developers frequently integrate their code into a single main branch ("trunk"), often multiple times a day. It relies heavily on feature flags to hide incomplete features and requires strong CI/CD and testing practices. It minimizes merge hell and enables continuous delivery.

18. What is a "bare repository"?

- **Answer:** A repository without a **working directory**. It is essentially just the .git project directory. It's used almost exclusively as a central remote repository on a server (e.g., on GitHub) because there's no need to have a working copy checked out. You create one with git init --bare.

19. How would you change the URL of a remote repository?

- **Answer:** Use git remote set-url. For the default origin: git remote set-url origin <new-url>.

20. What is git blame **and what is it useful for?**

- **Answer:** git blame <file> shows for each line in a file: the last commit that modified it, the author, and the timestamp. It's useful for identifying who introduced a specific change or bug, and understanding the context behind a piece of code.

Scenario-Based Questions

21. You just ran git reset --hard **and realized you need the code you just lost. How do you get it back?**

- **Answer:** Use git reflog. Find the action in the reflog that was before the disastrous reset (it will have the old HEAD hash). Then run git reset --hard <hash-from-reflog> to restore your branch to that state.

22. How would you find and delete all branches that have already been merged into main**?**

- **Answer:**
    1. First, ensure your remote-tracking branches are up-to-date: git fetch --prune.
    2. List and delete merged branches: git branch --merged main | grep -v "main" | xargs git branch -d.
        - This lists branches merged into main, excludes main itself, and deletes them. Use -D instead of -d to force delete unmerged branches.

23. How do you resolve a situation where you've pushed a commit with sensitive data (password, key)?

- **Answer: Warning:** This rewrites history.
    1. **If possible, immediately revoke the sensitive data!**
    2. Use git filter-repo (preferred) or git filter-branch to rewrite history and remove the file from *every* commit.
    3. Force push the corrected history: git push --force. You must notify all team members, who will need to re-clone or rebase their work on top of the new history.

24. Your git pull **resulted in a lot of merge conflicts. How can you abort the merge?**

- **Answer:** Run git merge --abort. This will halt the merge process and try to reconstruct the pre-merge state.

25. How would you set up Git to ignore changes in file permissions (mode)?

- **Answer:** Git tracks file permissions by default. To tell it to ignore permission changes, use: git config core.fileMode false. This is a common setup on Windows/WSL systems where permissions can change frequently.

Here are 25 advanced Git interview questions and answers, focusing on internal architecture, complex problem-solving, intricate workflows, and deep command mastery.

Internal Architecture & Deep Mechanics

1. Describe the internal object model of Git: Blobs, Trees, Commits, and Tags.

- **Answer:**
    - **Blob:** A **B**inary **L**arge **OB**ject. Stores the compressed contents of a single file. It does not store the filename, just the data.
    - **Tree:** A directory-listing object. It contains pointers to **blobs** (files) and other **trees** (subdirectories), along with their associated names and file modes (e.g., executable).
    - **Commit:** A snapshot object. It points to a single **tree** (representing the top-level directory of the project at the time of commit), contains metadata (author, committer, message, timestamp), and points to one or more parent **commits**.
    - **Tag:** An immutable pointer to a specific **commit**. It's often used to mark release points (v1.0.0). An **annotated tag** is a full Git object with its own hash, storing the tagger, message, and date.

2. What is the Git "index" or "staging area" at a fundamental level?

- **Answer:** The index is a binary file (typically located at .git/index) that acts as a **cache**. It's a sorted list of path names, each with permissions, the SHA-1 of the associated blob (file content), and stage number (for conflict resolution). It represents the proposed next commit. git add hashes a file into a blob and adds its info to the index.

3. Explain the purpose of the git cat-file **and** git hash-object **plumbing commands.**

- **Answer:** These are "plumbing" commands used to inspect and manipulate raw Git objects.
    - git hash-object -w <file>: Takes a file, computes its SHA-1 hash, creates a **blob** from its content, and writes it into the object database (-w flag), returning the hash.
    - git cat-file -p <hash>: Pretty-prints the content of the Git object identified by the given hash. Essential for exploring the object database.

4. How does Git efficiently compute the differences between files and commits?

- **Answer:** Git does not store diffs. It stores **snapshots** (full content of files in trees and blobs). When you ask for a diff (e.g., git show or git diff), Git dynamically computes the difference by:

1. Finding the relevant trees and blobs for the commits being compared.
2. Comparing the file lists and contents between the two snapshots.
3. Generating the diff output on-the-fly. This is efficient because blob hashes act as content-addressable keys.

5. What is the purpose of git fsck **and when would you use it?**

- **Answer:** git fsck (file system check) is a data integrity tool. It verifies the connectivity and validity of all objects in the database, checking for **dangling** objects (objects unreachable from any branch, tag, or HEAD) and corrupted objects. It's used to diagnose repository corruption.

Advanced Operations & History Rewriting

6. Explain the difference between git merge --squash **and a regular merge.**

- **Answer:**
    - **Regular Merge (**git merge <branch>**):** Creates a **merge commit** with two parents, preserving the entire history of the feature branch. The branch topology is maintained.
    - **Squash Merge (**git merge --squash <branch>**):** Does **not** create a merge commit. It takes all the changes from the feature branch, stages them as a single set of changes on your current branch, and lets you create a **new, single-parent commit**. It results in a linear history but completely discards the feature branch's commit history.

7. When would you use git filter-repo **over** git filter-branch**?**

- **Answer: Always.** git filter-repo is a modern, third-party tool designed to replace the built-in but dangerous git filter-branch. It is drastically faster, safer (by default it doesn't run on dirty repos), and has a simpler API for common tasks like permanently removing a file (with sensitive data) from history or rewriting author information. The Git project itself recommends filter-repo in its documentation.

8. Describe a scenario where you would use git replace**.**

- **Answer:** git replace lets you surgically rewrite history by telling Git to *pretend* that one commit is actually another, without actually moving any references. A key use case is to **fix a broken commit deep in history**. You can create a new, corrected commit and then use git replace to tell Git to use the new commit instead of the old

one for all operations. This is a non-destructive way to fix history before making it permanent with git filter-repo.

9. How do you perform a binary search through history for a bug that introduced a performance regression (which can't be seen by a simple test pass/fail)?

- **Answer:** Use git bisect run. You write a script that:
    1. Builds the code at the current commit.
    2. Runs a performance benchmark.
    3. Exits with code 0 if the performance is "good" and 125 (or any code 1-127 except 125) if the performance is "bad".
       Then run git bisect start, git bisect good <good-hash>, git bisect bad <bad-hash>, and finally git bisect run ./your-perf-script.sh. Git will automatically find the commit that introduced the regression.

10. What is the "rebase --onto" command and provide a use case.

- **Answer:** git rebase --onto <newbase> <oldbase> <branch> allows for sophisticated rebasing. It takes the commits from <oldbase>..<branch> and replays them onto <newbase>.
    - **Use Case:** You have a feature branch (feature) branched from main. You then realize it depends on another unmerged feature branch (dependency). You can rebase *just your work* onto the tip of dependency:
      git rebase --onto dependency main feature
      This says: "Take the commits since main diverged (i.e., the commits unique to feature) and put them on top of dependency."

Complex Workflows & Collaboration

11. How does Git handle a "push" operation under the hood?

- **Answer:** The client (git push) tells the remote server about any new objects (commits, trees, blobs, tags) it has. The server integrates these objects into its repository. Then, it performs a **reference transaction**: it tries to update the branch reference on the remote (e.g., refs/heads/main) to point to the new commit, but only if it's a **fast-forward** update. If not (someone else pushed first), the push is rejected.

12. Explain how Git Hooks can be used to enforce a project policy (e.g., commit message format, tests passing).

- **Answer: Server-side hooks** (like pre-receive and update) are crucial for policy enforcement on a central repository.
    - A pre-receive hook gets a list of all refs being updated. It can reject the *entire push* if any commit fails a policy (e.g., a commit message doesn't match a regex pattern).
    - An update hook runs per ref being updated and can reject an update to a specific branch (e.g., reject any direct push to main, enforcing pull requests only).
    - A pre-commit hook (client-side) can check code style before a commit is even created.

13. What is a "diverged branch" and how can you recover from a situation where git pull **is rejected because of it?**

- **Answer:** A branch diverges when both your local branch and the remote branch have new, unique commits that the other doesn't have. git pull (which is fetch + merge) will be rejected if the merge would create a conflict.
    - **Solution 1 (Preserve History):** git pull --rebase. This will rewind your local commits, apply the remote commits, and then replay your local commits on top. This creates a linear history.
    - **Solution 2 (Explicit Merge):** Perform the merge manually: git fetch, git merge origin/your-branch, resolve any conflicts, then git push.

14. Describe how to use git worktree **and what problem it solves.**

- **Answer:** git worktree allows you to have **multiple working directories attached to the same repository**. It solves the problem of context switching. Instead of stashing your current work to check a different branch, you can:
  git worktree add ../hotfix-branch hotfix
  This creates a new directory ../hotfix-branch where you can work on the hotfix branch simultaneously, without disturbing your main working directory.

15. What are Git Submodules and what are their main drawbacks?

- **Answer:** Submodules allow you to keep a Git repository as a subdirectory of another Git repository. They pin a dependent project to a specific commit.
    - **Drawbacks:** They add complexity. Users must know to use git clone --recursive or git submodule update --init. They can be confusing to update (git

submodule update). They create a slight disconnect between the superproject and the subproject's latest code.

Deep Configuration & Optimization

16. What is the difference between git grep **and standard command-line** grep**?**

- **Answer:** git grep is significantly faster for searching through a Git repository because it only searches files that are tracked and present in the working tree. It can also search through specific commits or branches (e.g., git grep "foo" v2.1) and understands .gitignore patterns.

17. How would you configure Git to use a different merge strategy (e.g., "ours") for a specific file to avoid merge conflicts?

- **Answer:** You can define a custom merge driver in your .gitattributes file.
    1. First, define the driver in your Git config: git config merge.keepours.driver "true"
    2. Then, in .gitattributes, assign the driver to a file pattern: path/to/file.xml merge=keepours
       Now, during any merge involving file.xml, Git will always keep the "ours" version and ignore the "theirs" version.

18. Explain what git packfiles **are and their purpose.**

- **Answer:** Packfiles are Git's mechanism for efficient storage and network transfer. They combine many loose objects (individual files for each blob, tree, commit) into a single, compressed, deltafied packfile. Objects similar to each other are stored as deltas (differences), drastically reducing repository size. git gc (garbage collection) triggers packing.

19. What does git prune **do and when is it run automatically?**

- **Answer:** git prune is a plumbing command that removes **dangling objects** (objects that are no longer reachable from any branch, tag, or reference). It is run automatically by higher-level commands like git gc and git fetch --prune.

20. How can you significantly accelerate operations like git status **on a very large repository?**

- **Answer:** Enable the **untracked cache** and **fsmonitor** features.

- **Untracked Cache:** git config core.untrackedCache true. Caches information about untracked files, so status doesn't have to scan the entire working tree for them every time.
- **FSMonitor:** git config core.fsmonitor true (requires a compatible daemon). Uses a filesystem watcher daemon to instantly know which files have changed, eliminating the need for status to scan the filesystem stat(). This provides a massive speedup.

Expert Scenarios & Troubleshooting

21. Your repository has become very large and slow. How would you diagnose what is causing the bloat?

- **Answer:**
    1. Use git count-objects -vH to see the size and number of loose objects and packfiles.
    2. Use git rev-list --objects --all --disk-usage=human to see the total disk usage.
    3. Use git verify-pack -v .git/objects/pack/*.idx | sort -k 3 -n | tail -5 to find the **largest objects** in the packfile.
    4. Cross-reference the largest blob SHAs with git name-rev <sha> and git log --oneline --find-object=<sha> to find which commits introduced these large files.

22. How would you recover from a corrupted object?

- **Answer:**
    1. First, try git fsck to identify the corrupted object's hash.
    2. If the corruption is recent, you might have a copy of the object in a backup or another clone of the repository. You can copy the good object file (e.g., .git/objects/ab/cdef123...) from the backup into your corrupted repo.
    3. If the object is a commit and you can't recover it, you might need to use git log --reflog to find a previous good state and use git reset --hard to that state, effectively rewriting history to before the corruption.

23. Describe how to do a "reverse" of a git rebase -i **(squash) operation after you've already pushed.**

- **Answer:** You **cannot** do this safely without force-pushing and disrupting collaborators. The key is to **never rebase public history**. If you absolutely must, and you are prepared to force-push and have everyone re-clone, you can use git reflog to find the pre-rebase state (the old branch tip) and then git reset --hard to that point to undo the rebase locally before force-pushing.

24. What is git rerere **and how can it save you time?**

- **Answer:** git rerere (Reuse Recorded Resolution) is a hidden gem. When enabled (git config rerere.enabled true), it records how you resolved a particular merge conflict (the hunk conflict, not the file). The next time Git encounters the **exact same conflict** (e.g., during a rebase or another merge), it will automatically resolve it for you using the recorded resolution. It's a huge time-saver for long-lived branches that are rebased often.

25. How would you implement a deployment strategy where pushing to a specific branch (e.g., production**) automatically deploys to a server?**

- **Answer:** This is implemented using a **server-side hook** on the remote repository (e.g., on GitHub/GitLab or your own Git server). You would use a post-receive hook. This hook is triggered after the references have been updated. The script in the hook would:
    1. Check if the push was to the production branch.
    2. If so, it would git checkout -f production into a deployment directory on the server.
    3. Then, it could run any necessary deployment commands (restart services, run build scripts, etc.).
       Platforms like GitHub Actions or GitLab CI/CD provide a more robust and managed way to do this today.

- Here is a comprehensive list of Git commands, categorized by their function, with a short explanation for each.
- Getting Started & Setup

Command	Description
`git config --global user.name "Your Name"`	Sets your **username** for all repositories on your system.

Command	Description
`git config --global user.email "your@email.com"`	Sets your **email** for all repositories on your system.
`git init`	Creates a **new, empty Git repository** in the current directory.
`git clone <url>`	**Copies** an existing remote repository to your local machine.

- Basic Snapshotting (The Core Workflow)

Command	Description
`git status`	Shows the state of the **working directory** and **staging area**.
`git add <file>`	Adds a specific file's changes to the **staging area**.
`git add .`	Adds all **new and modified** files to the staging area.
`git add -A`	Adds **all** changes (new, modified, deleted) to the staging area.
`git commit -m "message"`	**Permanently records** the staged changes in the repository's history.
`git commit --amend`	**Modifies the most recent commit** (message or add forgotten files).

- Branching & Merging

Command	Description
`git branch`	Lists all **local branches**.
`git branch <branch-name>`	Creates a **new branch**.
`git branch -d <branch-name>`	**Deletes** a branch (safe, prevents loss of unmerged data).
`git branch -D <branch-name>`	**Force deletes** a branch (even if unmerged).

Command	Description
`git checkout <branch-name>`	**Switches** to an existing branch.
`git checkout -b <branch-name>`	Creates a **new branch and switches** to it (shortcut).
`git switch <branch-name>`	(Newer) Switches to an existing branch.
`git switch -c <branch-name>`	(Newer) Creates a new branch and switches to it.
`git merge <branch-name>`	**Combines** the specified branch's history into the current branch.
`git merge --squash <branch>`	Combines changes into a single, new commit on the current branch.
`git rebase <base-branch>`	**Reapplies** commits from the current branch onto the tip of another branch, creating a linear history.
`git rebase -i <commit>`	Starts an **interactive rebase**, allowing you to edit, squash, or reorder commits.

- Sharing & Updating

Command	Description
`git remote -v`	Lists all **remote connections** (with their URLs).
`git remote add <name> <url>`	Adds a new **remote connection** (like `origin`).
`git fetch <remote>`	**Downloads** commits and branches from a remote but **does not merge**.
`git pull <remote> <branch>`	**Fetches** from a remote and **merges** into the current branch (`fetch` + `merge`).
`git pull --rebase`	Fetches and then **rebases** current branch on top of remote changes.

Command	Description
`git push <remote> <branch>`	**Uploads** local commits to a remote repository.
`git push -u <remote> <branch>`	Pushes and sets the remote branch as the **upstream tracking branch**.
`git push --force-with-lease`	**Safely force-pushes** (overwrites remote history) only if no one else has pushed.

- Inspecting & Comparing

Command	Description
`git log`	Shows the **commit history**.
`git log --oneline --graph --all`	Shows a condensed, visual history of all branches.
`git diff`	Shows **unstaged changes** (differences in working directory).
`git diff --staged`	Shows **staged changes** (differences in the staging area).
`git diff <commit1> <commit2>`	Shows differences between two commits.
`git show <commit>`	Shows the changes and metadata of a specific commit.
`git blame <file>`	Shows who last modified each line of a file.

- Undoing Things

Command	Description
`git restore <file>`	(Newer) **Discards unstaged changes** in the working directory.
`git checkout -- <file>`	(Older) Discards unstaged changes in the working directory.
`git restore --staged <file>`	(Newer) **Unstages** a file but keeps the changes in the working directory.

Command	Description
`git reset HEAD <file>`	(Older) Unstages a file.
`git reset --soft <commit>`	Moves HEAD to a previous commit, **keeping changes staged**.
`git reset --mixed <commit>`	(Default) Moves HEAD and **unstages changes**, but keeps them in working dir.
`git reset --hard <commit>`	**DANGER:** Moves HEAD and **discards all changes** since that commit.
`git revert <commit>`	Creates a **new commit** that undoes the changes of a previous commit (safe for shared history).

- Stashing

Command	Description
`git stash`	**Temporarily shelves** all tracked, modified files.
`git stash -u`	Stashes including **untracked** files.
`git stash list`	Lists all stashed changesets.
`git stash pop`	**Reapplies** the most recently stashed changes and removes them from the stash list.
`git stash apply`	Reapplies stashed changes but **keeps** them in the stash list.
`git stash drop`	Discards the most recent stashed changeset.

- Advanced & Internal

Command	Description
`git tag <tagname>`	Creates a **lightweight tag** for the current commit.
`git tag -a v1.0 -m "msg"`	Creates an **annotated tag** (with message) for the current commit.

Command	Description
`git bisect start`	Uses **binary search** to find the commit that introduced a bug.
`git worktree add ../dir`	Allows you to have **multiple working trees** attached to one repository.
`git reflog`	Shows a log of where **HEAD** and branch references have been. Your safety net for mistakes.
`git grep "pattern"`	Searches for a string pattern in **tracked files** (very fast).

- Configuration & Help

Command	Description
`git config --list`	Lists all current Git configuration settings.
`git help <command>`	Opens the **manual page** for a specific Git command.
`git <command> -h`	Shows a quick **help summary** for a specific command.