334 Git / GitHub Commands (Code + Explanation)

1.
```
git init
```

Initializes a new Git repository in the current directory.

This creates a hidden `.git` folder to track changes.

2.
```
git clone <repo>
```

Clones an existing repository from a remote source (e.g., GitHub) to your local machine.

This includes all files, commit history, and branches.

3.
```
git status
```

Shows the current state of the working directory and staging area.

Helps you see which changes are staged, unstaged, or untracked.

4.
```
git add <file>
```

Adds a specific file to the staging area for the next commit.

This does not commit the file yet—it just marks it to be committed.

5.
```
git add .
```

Adds all new and changed files in the current directory to the staging area.

Useful for staging multiple files at once.

6.
```
git commit -m "message"
```

Records staged changes to the repository with a descriptive message.

Messages should be clear to explain the reason for the change.

7.
```
git commit --amend
```

Rewrites the latest commit, allowing you to modify the message or add changes.

This replaces the last commit in history.

> **Tip:** Amending commits rewrites history—avoid if you have already pushed to a shared branch.

8.
```
git log
```

Displays the commit history for the current branch.

Shows commit hashes, author info, dates, and messages.

9.
```
git log --oneline
```

Shows a simplified commit history with one commit per line.

Useful for a quick overview of the branch history.

10.
```
git log --graph --decorate --all
```

Displays the commit history as a graph with branch names and tags.

Helps visualize branching and merging.

11.
```
git reflog
```

Shows the history of all reference changes in your local repository.

Useful for recovering lost commits.

12.
```
git bisect
```

Performs a binary search through commits to find the one that introduced a bug.

Saves time in debugging large projects.

13.
```
git cherry-pick <commit>
```

Applies a specific commit from one branch to another.

Helps selectively merge changes without full branch merges.

14.
```
git stash
```

Temporarily saves changes without committing them.

Allows you to switch branches without losing progress.

15.
```
git stash pop
```

Applies the most recent stashed changes and removes them from the stash list.

Useful for resuming work quickly.

16.
```
git rebase -i HEAD~n
```

Starts an interactive rebase for the last `n` commits.

Lets you reorder, edit, or squash commits.

17.
```
git tag <tagname>
```

Creates a new tag pointing to the current commit.

Tags are often used for marking releases.

18.
```
git remote -v
```

Lists all remote connections and their URLs.

Helps verify where your repository pushes and pulls from.

19.
```
git fetch --all --prune
```

Fetches all branches from all remotes and removes deleted ones.

Keeps your local repo synchronized with the remote.

20.
```
git gc --prune=now --aggressive
```

Performs aggressive garbage collection to optimize repository size and performance.

Should be used occasionally for maintenance.

21.
```
git worktree add <path> <branch>
```

Creates a new working tree linked to an existing repository.

Useful for checking out multiple branches at once.

22.
```
git bundle create repo.bundle --all
```

Packages the repository into a single file for backup or transfer.

Can be unbundled later with `git clone`.

23.
```
git archive --format=zip HEAD > archive.zip
```

Creates a ZIP archive of the current branch.

Does not include Git history—just files.

24.

```
git blame <file>
```

Shows who last modified each line of a file and when.

Useful for tracking changes.

25.

```
git shortlog -s -n
```

Displays a summary of commits per author.

Helps analyze contributions.

26.

```
git config --global alias.co checkout
```

Creates a shortcut alias for a Git command.

Aliases save time on repetitive commands.

27.

```
git show <commit>
```

Displays details about a specific commit.

Includes changes, author, and date.

28. `git clean -fd`

Removes untracked files and directories.

Useful for cleaning up the working directory.

29. `git revert <commit>`

Creates a new commit that undoes the changes from a previous commit.

Does not rewrite history like `reset`.

30. `git reset --hard HEAD~1`

Resets the current branch to the previous commit, discarding changes.

Use with caution—it deletes uncommitted work.

31. `git push --force-with-lease`

Forces a push but only if the remote has not been updated by others.

Safer than `--force`.

32.
```
git pull --rebase
```

Pulls from the remote and applies changes on top of local commits.

Helps maintain a linear history.

33.
```
git diff --staged
```

Shows changes that are staged for the next commit.

Does not show unstaged changes.

34.
```
git diff <commit1> <commit2>
```

Shows differences between two commits.

Can be used to review specific changes.

35.
```
git describe --tags
```

Describes the current commit using the nearest tag.

Useful for versioning.

36.
```
git ls-tree HEAD
```

Lists the contents of a tree object (e.g., a commit).

Shows file modes, object types, and names.

37.
```
git fsck
```

Verifies the integrity of the Git database.

Reports any corruption.

38.
```
git notes add -m "Note" <commit>
```

Adds a note to a specific commit.

Notes are stored separately from commit messages.

39.
```
git submodule update --init --recursive
```

Initializes and updates all submodules in the repository.

Ensures all nested dependencies are pulled.

40.
```
git mergetool
```

Opens the configured merge tool to help resolve merge conflicts.

Useful when dealing with complex file differences.

41.
```
git rerere
```

Enables reuse recorded resolution for merge conflicts.

Saves previous conflict resolutions to speed up future merges.

42.
```
git range-diff <base> <head>
```

Shows differences between two commit ranges.

Helps compare two series of commits.

43.
```
git filter-branch --tree-filter 'command'
```

Rewrites commit history applying a shell command to each tree.

Can be used to remove sensitive files from history.

44. `git replace <object> <replacement>`

Replaces one Git object with another.

Useful for altering commits without rewriting history.

45. `git verify-commit <commit>`

Checks the GPG signature of a commit.

Ensures the commit came from a trusted source.

46. `git verify-tag <tag>`

Checks the GPG signature of a tag.

Confirms authenticity of tagged releases.

47. `git show-ref`

Displays references and their commit hashes.

Useful for debugging refs in a repository.

48.
```
git ls-files --stage
```

Shows staged files with object details.

Helps inspect the index state.

49.
```
git rev-parse HEAD
```

Outputs the hash of the current commit.

Used in scripts to get exact commit IDs.

50.
```
git commit --signoff
```

Adds a "Signed-off-by" line to the commit message.

Commonly used in open-source contribution workflows.

51.
```
git bundle verify repo.bundle
```

Verifies the integrity of a Git bundle file.

Ensures it can be safely cloned or unbundled.

52.
```
git format-patch origin/main
```

Generates patch files for commits not in the target branch.

Used for sending changes via email or review systems.

53.
```
git am <patch-file>
```

Applies a patch file to the current branch.

Often used after `git format-patch`.

54.
```
git grep "pattern"
```

Searches for a text pattern across files in the repository.

Works on any commit or branch.

55.
```
git whatchanged
```

Shows commit history with file changes.

Similar to `git log` but focused on diffs.

56.
```
git log --stat
```

Displays commit history with file change statistics.

Shows lines added and deleted per file.

57.
```
git branch --contains <commit>
```

Lists branches that contain a specific commit.

Helps identify where a commit exists.

58.
```
git rev-list --count HEAD
```

Counts the total number of commits in the current branch.

Useful for versioning and stats.

59.
```
git config --global core.autocrlf true
```

Automatically converts line endings between Windows and Unix formats.

Prevents inconsistent line endings in projects.

60.

```
git cherry <upstream> <head>
```

Shows commits in HEAD that are not in the upstream branch.

Helps identify unique changes before merging.

61.

```
git gc --auto
```

Runs Git’s automatic garbage collection when needed.

Optimizes repository storage without aggressive cleanup.

62.

```
git show-branch
```

Displays branches and their commits in a compact form.

Useful for comparing branch differences.

63.

```
git fsck --full
```

Performs a full integrity check of the Git object database.

Detects any corruption in the repository.

64.
```
git archive --format=tar HEAD | gzip > archive.tar.gz
```

Creates a tar.gz archive of the current branch.

Only includes files, not Git history.

65.
```
git for-each-ref
```

Iterates over all references in the repository.

Useful for custom scripts and automation.

66.
```
git commit --allow-empty -m "Message"
```

Creates a commit without any file changes.

Often used for triggering CI/CD pipelines.

67.
```
git diff --name-status
```

Shows names and status (added, modified, deleted) of changed files.

Provides a quick change summary.

68.
```
git merge --no-ff <branch>
```

Merges a branch without fast-forwarding.

Keeps a merge commit for clear history.

69.
```
git merge --abort
```

Aborts the current merge process and resets to pre-merge state.

Used when merge conflicts can’t be resolved.

70.
```
git rebase --abort
```

Aborts an ongoing rebase and restores the branch to its previous state.

Useful when rebase causes unexpected issues.

71.
```
git diff --check
```

Identifies whitespace errors in changes.

Helps maintain clean coding style.

72. `git commit --date="YYYY-MM-DD" -m "Message"`

Creates a commit with a custom date.

Can be used for backdating commits.

73. `git push origin --delete <branch>`

Deletes a branch on the remote repository.

Used for cleaning up merged or unused branches.

74. `git tag -d <tag>`

Deletes a local tag.

To delete on remote, use `git push origin --delete tag <tag>`.

75. `git show <tag>`

Displays details of the specified tag.

Includes commit, author, and message info.

76.
```
git branch -r
```

Lists all remote branches.

Helps identify available remote branch names.

77.
```
git branch -a
```

Lists both local and remote branches.

Useful for complete branch overview.

78.
```
git log --since="2 weeks ago"
```

Shows commits made within the last two weeks.

Supports many time formats like "yesterday" or specific dates.

79.
```
git log --author="Name"
```

Displays commits made by a specific author.

Helps filter history by contributor.

80.
```
git stash list
```

Lists all the stashed changes in your repository.

Useful to see saved work that is not yet applied.

81.
```
git stash show
```

Displays a summary of the changes in the most recent stash.

Helps quickly review what is saved.

82.
```
git stash drop <stash@{n}>
```

Deletes a specific stash entry from the stash list.

Frees space and cleans old stashes.

83.
```
git stash clear
```

Removes all stash entries.

Use with caution as this deletes all saved stashes.

84.
```
git branch -d <branch>
```

Deletes a local branch that has been merged.

Prevents accidental deletion of unmerged branches.

85.
```
git branch -D <branch>
```

Forcibly deletes a local branch regardless of merge status.

Use carefully to avoid losing unmerged work.

86.
```
git checkout -b <new-branch>
```

Creates and switches to a new branch in one step.

Saves time creating and checking out branches separately.

87.
```
git checkout <branch>
```

Switches to an existing branch.

Updates working directory to the branch’s latest commit.

88.
```
git switch <branch>
```

Modern command to switch branches.

Preferred over checkout for branch switching only.

89.
```
git switch -c <new-branch>
```

Creates and switches to a new branch.

Equivalent to `git checkout -b`.

90.
```
git reset <file>
```

Unstages a file that was added to the staging area.

Does not delete changes in the working directory.

91.
```
git reset --soft <commit>
```

Moves HEAD to a previous commit, keeps staged and working directory changes.

Useful for undoing commits but keeping changes ready to commit.

92.
```
git reset --mixed <commit>
```

Moves HEAD and resets staging area to a commit, keeps working directory intact.

Default reset behavior, unstages changes.

93.
```
git reset --hard <commit>
```

Moves HEAD to a commit and resets staging and working directory.

All changes after the commit are lost.

94.
```
git reflog expire --expire=now --all
```

Expires reflog entries immediately.

Used for cleaning reflog history.

95.
```
git reflog delete <ref>@{<n>}
```

Deletes a specific reflog entry.

Useful for removing unwanted reflog history.

96.
```
git bisect start
```

Begins a bisect session to find a problematic commit.

Requires marking good and bad commits next.

97.
```
git bisect good <commit>
```

Marks a commit as good during bisecting.

Helps narrow down the problematic commit.

98.
```
git bisect bad <commit>
```

Marks a commit as bad during bisecting.

Identifies where the bug or issue started.

99.
```
git bisect reset
```

Ends the bisect session and returns to the original branch.

Clears the bisect state.

100.

```
git config --list
```

Lists all Git configuration variables.

Shows system, global, and local settings.

101.

```
git config user.name "Your Name"
```

Sets the username for commits.

Important for commit author info.

102.

```
git config user.email "you@example.com"
```

Sets the email for commits.

Used in commit metadata.

103.

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

Sets VS Code as the default editor for Git.

Use any editor with the appropriate command.

104.

```
git submodule init
```

Initializes submodules recorded in the repository.

Prepares submodules for fetching.

105.

```
git submodule update
```

Checks out the specific versions of submodules.

Synchronizes submodules to recorded commits.

106.

```
git submodule sync
```

Updates URLs of submodules if they have changed.

Ensures remote URLs are up-to-date.

107.

```
git cherry-pick --continue
```

Continues cherry-pick after resolving conflicts.

Completes the cherry-pick operation.

108.

```
git cherry-pick --abort
```

Aborts the current cherry-pick operation.

Restores the branch to pre-cherry-pick state.

109.

```
git merge --squash <branch>
```

Squashes changes from a branch into a single commit.

Does not create a merge commit automatically.

110.

```
git blame -L <start>,<end> <file>
```

Shows blame information for a range of lines in a file.

Helps track changes in specific sections.

111.

```
git log --follow <file>
```

Shows commit history including renames for a file.

Tracks the file across name changes.

112.
```
git archive --prefix=folder/ -o archive.zip HEAD
```

Creates a zip archive with files inside a folder prefix.

Useful for packaging with directory structure.

113.
```
git shortlog -sn
```

Summarizes commit counts per author in short format.

Shows contribution statistics.

114.
```
git rev-parse --abbrev-ref HEAD
```

Shows the current branch name.

Used in scripting and status checks.

115.
```
git remote show origin
```

Shows detailed info about the remote repository named origin.

Includes fetch/push URLs and branch tracking.

116.

```
git reflog show HEAD
```

Displays reflog entries for HEAD.

Useful to recover lost commits or check recent HEAD moves.

117.

```
git checkout -- <file>
```

Discards changes in the working directory for a file.

Resets file to last committed state.

118.

```
git tag -a <tag> -m "Message"
```

Creates an annotated tag with a message.

Annotated tags include metadata like author and date.

119.

```
git tag -l
```

Lists all tags in the repository.

Useful for checking available release points.

120.

```
git push origin <branch>
```

Pushes local branch changes to the remote repository.

Synchronizes your commits with others.

121.

```
git push origin --tags
```

Pushes all local tags to the remote repository.

Makes tags available to collaborators.

122.

```
git fetch origin
```

Fetches changes from the remote without merging.

Updates remote tracking branches only.

123.

```
git pull --ff-only
```

Pulls remote changes only if fast-forward is possible.

Prevents merge commits during pull.

124.

```
git clean -n
```

Shows which untracked files would be removed without deleting.

Useful preview before cleaning.

125.

```
git clean -f
```

Removes untracked files from the working directory.

Use with caution to avoid deleting needed files.

126.

```
git clean -fd
```

Removes untracked files and directories.

Useful for deep cleaning.

127.

```
git diff --cached
```

Shows changes staged for the next commit.

Synonym for `git diff --staged`.

128.

```
git merge --strategy=ours <branch>
```

Merges a branch but keeps current branch’s changes on conflicts.

Useful to ignore changes from the merged branch.

129.

```
git merge --strategy=theirs <branch>
```

Opposite of 'ours' strategy, prefers incoming branch changes.

Used to accept remote changes in conflicts.

130.

```
git rebase -i --autosquash
```

Interactive rebase that automatically moves fixup commits.

Simplifies cleaning commit history.

131.

```
git show --stat <commit>
```

Shows commit changes with a summary of file modifications.

Useful for quick review of a commit.

132.

```
git log --pretty=format:"%h - %an, %ar : %s"
```

Displays a custom formatted log.

Shows commit hash, author, relative date, and message.

133.

```
git log --patch-with-stat
```

Shows commit patches along with file change statistics.

Combines diff and summary in log.

134.

```
git clean -xdf
```

Removes untracked files, directories, and ignored files.

Use with caution for full cleanup.

135.

```
git update-index --assume-unchanged <file>
```

Tells Git to temporarily ignore changes to a file.

Useful for local config files you don’t want to commit.

136.

```
git update-index --no-assume-unchanged <file>
```

Reverses the assume-unchanged flag on a file.

Resumes normal tracking of changes.

137.

```
git reflog expire --expire-unreachable=now --all
```

Expires unreachable reflog entries immediately.

Useful for aggressive reflog cleanup.

138.

```
git submodule foreach <command>
```

Runs a shell command in each checked out submodule.

Helps automate tasks across submodules.

139.

```
git rev-list --all --count
```

Counts all commits in all branches.

Useful for repository statistics.

140.

```
git prune
```

Removes unreachable objects from the object database.

Used for cleaning and reducing repo size.

141.

```
git blame -p <file>
```

Shows detailed blame information with commit metadata.

Useful for deep analysis of file changes.

142.

```
git notes show <commit>
```

Displays notes attached to a specific commit.

Helps add metadata or comments to commits.

143.

```
git worktree prune
```

Removes stale working trees that no longer exist.

Maintains worktree configurations clean.

144.

```
git worktree list
```

Lists all linked working trees.

Shows branches checked out in multiple working directories.

145.

```
git stash branch <branch>
```

Creates a new branch from the stash and applies stashed changes.

Useful for resuming stashed work in a new branch.

146.

```
git blame --reverse <file>
```

Shows blame starting from the first commit (oldest first).

Helps understand file evolution chronologically.

147.

```
git cat-file -p <object>
```

Displays the content of a Git object.

Useful for debugging low-level Git data.

148.

```
git diff --color-words
```

Shows changes with word-level highlighting.

Improves readability of diffs.

149.

```
git cherry-pick -x <commit>
```

Cherry-picks a commit and appends a reference to the original commit.

Helps trace back the source of cherry-picked changes.

150.

```
git push --set-upstream origin <branch>
```

Sets upstream tracking for a new branch during push.

Makes future pushes and pulls easier.

151.

```
git remote add <name> <url>
```

Adds a new remote repository.

Used to track multiple remotes.

152.

```
git remote remove <name>
```

Removes a remote repository.

Cleans up unused remote references.

153.

```
git ls-remote
```

Lists references in a remote repository.

Useful to see branches and tags on remotes.

154.

```
git mergetool --tool-help
```

Lists all available merge tools configured.

Helps pick the best tool for resolving conflicts.

155.

```
git commit --verbose
```

Shows diff in commit message editor.

Helps review changes while writing commit messages.

156.

```
git show --name-only
```

Displays the commit information with just the filenames changed.

Useful for quick overview of files affected.

157.

```
git rev-parse --show-toplevel
```

Shows the absolute path of the top-level directory of the repository.

Useful in scripts to find repo root.

158.

```
git update-ref -d <ref>
```

Deletes a reference (branch, tag, etc.).

Used for low-level ref management.

159.

```
git cherry-pick --edit <commit>
```

Starts cherry-pick and opens commit message editor.

Allows modifying the commit message before applying.

160.

```
git filter-repo --path <file> --invert-paths
```

Removes a file from entire repository history.

Modern alternative to `filter-branch`

```
git worktree add -b <branch> <path> <start-point>
```

Creates and checks out a new branch in a new working tree.

Useful for working on multiple branches simultaneously.

161.

```
git commit --fixup <commit>
```

Creates a commit intended to fix a previous commit.

Works with autosquash during interactive rebase.

162.

```
git commit --squash <commit>
```

Prepares a commit to be squashed into a specified commit.

Used with interactive rebase and autosquash.

163.

```
git rebase --autosquash
```

Automatically moves fixup and squash commits to be combined.

Makes cleaning up commit history easier.

164.

```
git remote set-url <name> <newurl>
```

Changes the URL of a remote repository.

Useful if remote repository location changes.

165.

```
git fetch --prune
```

Fetches changes and removes remote-tracking references that no longer exist.

Keeps local references clean and up to date.

166.

```
git log --cherry-mark
```

Marks equivalent commits in different branches.

Helps identify duplicate commits during comparisons.

167.

```
git log --left-right <branch1>...<branch2>
```

Shows commits unique to each branch with directional markers.

Useful for comparing branches.

168.

```
git revert --no-commit <commit>
```

Reverts a commit but does not create a commit automatically.

Allows batching multiple reverts before committing.

169.

```
git replace --delete <object>
```

Deletes a previously created replacement object.

Restores original object behavior.

170.

```
git reflog expire --expire-unreachable=now --all
```

Expires all unreachable reflog entries immediately.

Useful for aggressive reflog cleanup and pruning.

171.

```
git push --mirror
```

Pushes all refs (branches, tags) to the remote repository.

Useful for mirroring repositories exactly.

172.

```
git show-branch --topics
```

Shows topic branches with their recent commits.

Helps overview active development branches.

173.

```
git archive --remote=<repo> HEAD | tar -x
```

Creates an archive from a remote repository without cloning.

Useful for downloading source without full clone.

174.

```
git config --get-regexp <pattern>
```

Lists config entries matching a regex pattern.

Helpful to find specific config settings.

175.

```
git log --since="yesterday" --until="today"
```

Shows commits made in the last day.

Supports flexible date ranges for log filtering.

176.
```
git shortlog --email
```

Shows commit counts per author with email addresses.

Useful for detailed contributor stats.

177.
```
git update-index --skip-worktree <file>
```

Tells Git to avoid changes in a file (used mainly in sparse checkouts).

Different from assume-unchanged, affects working directory.

178.
```
git ls-files -o --exclude-standard
```

Lists untracked files excluding those ignored by .gitignore.

Useful to see which files are truly untracked.

179.
```
git stash push -m "message"
```

Saves your local modifications to a new stash with a message.

Helps keep track of multiple stashes.

180.

```
git stash apply <stash@{n}>
```

Applies a specific stash without removing it from the stash list.

Allows reusing stash multiple times.

181.

```
git blame --show-name <file>
```

Shows commit and author along with filename for each line.

Helpful when blaming merged files.

182.

```
git rev-list --max-count=10 HEAD
```

Lists the last 10 commits from HEAD.

Useful for limiting log output.

183.

```
git log --grep="fix bug"
```

Searches commit messages matching a pattern.

Helps find commits by keywords.

184.

```
git describe --all
```

Describes the current commit with all refs pointing to it.

Shows tags, branches, or HEAD info.

185.

```
git filter-repo --path <file> --invert-paths
```

Removes a file from all history using modern tool.

Replaces deprecated filter-branch for rewriting history.

186.

```
git log --pretty=format:"%h %s" --graph --decorate --all
```

Shows a colorful graph of all commits with short hashes and messages.

Great for visualizing repository history.

187.

```
git notes edit <commit>
```

Opens editor to add or modify notes on a commit.

Useful to attach metadata or comments.

188.

```
git worktree move <path> <new-path>
```

Moves a linked worktree to a new location.

Keeps branch checkout intact.

189.

```
git reflog show --date=iso
```

Shows reflog with ISO formatted dates.

Improves readability of reflog timestamps.

190.

```
git log --numstat
```

Shows the number of added and deleted lines per file in commits.

Helpful for analyzing code churn.

191.

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

Sets default branch name for new repositories.

Replaces default 'master' with 'main' or preferred name.

192.

```
git reflog expire --expire=1.week.ago --all
```

Expires reflog entries older than one week.

Useful for cleaning reflog gradually.

193.

```
git merge --no-commit <branch>
```

Merges a branch but pauses before committing.

Allows manual changes or conflict resolution first.

194.

```
git bisect visualize
```

Launches a GUI to help visualize the bisect process.

Requires a configured GUI tool.

195.

```
git symbolic-ref HEAD
```

Shows the full reference of the current branch.

Outputs something like refs/heads/main.

196.

```
git stash drop <stash@{n}>
```

Removes a stash entry by its index.

Use to clean specific saved changes.

197.

```
git push origin --force-with-lease
```

Force pushes only if remote hasn't changed.

Safer alternative to plain --force.

198.

```
git ls-tree -r HEAD
```

Lists all files in the current commit recursively.

Shows object types and modes.

199.

```
git rev-parse --show-prefix
```

Displays the path prefix relative to the repository root.

Useful for scripting inside subdirectories.

200.

```
git config --local core.hooksPath <path>
```

Sets custom directory for Git hooks.

Overrides default .git/hooks directory.

201.

```
git submodule absorbgitdirs
```

Makes submodules store their .git directory inside the superproject.

Useful for simpler submodule management.

202.

```
git tag --contains <commit>
```

Lists tags that include a given commit.

Helps find releases containing a fix or feature.

203.

```
git checkout --detach <commit>
```

Checks out a commit in detached HEAD state.

Used for testing or inspecting history without branch.

204.

```
git merge --quiet <branch>
```

Merges a branch without outputting messages.

Useful in scripts to reduce noise.

205.

```
git blame --show-email <file>
```

Shows commit author emails alongside lines.

Helpful for contacting contributors.

206.

```
git revert -n <commit>
```

Reverts a commit without committing immediately.

Allows batching multiple reverts.

207.

```
git ls-remote --tags origin
```

Lists remote tags from the origin repository.

Useful for verifying remote tags.

208.

```
git remote prune origin
```

Deletes stale remote tracking branches for origin.

Keeps local references synchronized.

209.

```
git fetch --depth=1
```

Performs a shallow fetch with only the latest commit.

Speeds up cloning and fetching large repos.

210.

```
git worktree remove <path>
```

Removes a linked working tree.

Does not delete the files on disk.

211.

```
git update-ref refs/heads/<branch> <commit>
```

Updates a branch ref to point to a specific commit.

Used for low-level ref manipulation.

212.

```
git reset --merge
```

Resets the index and updates files in the working tree, preserving unmerged entries.

Useful to abort merges safely.

213.

```
git rebase --onto <newbase> <upstream> <branch>
```

Rebases a branch onto a new base, starting after upstream.

Powerful tool for complex history rewriting.

214.

```
git log --reverse
```

Shows commits in reverse chronological order (oldest first).

Helps read history from beginning.

215.

```
git config --unset <key>
```

Removes a Git config variable.

Use to delete unwanted settings.

216.

```
git push --dry-run
```

Simulates a push without actually transferring data.

Useful to check what would be pushed.

217.

```
git describe --tags --abbrev=0
```

Shows the most recent tag reachable from HEAD.

Useful for version numbering.

218.

```
git config core.fileMode false
```

Disables file mode tracking.

Useful on filesystems where execute permissions are unreliable.

219.

```
git submodule summary
```

Shows a summary of changes in submodules.

Quick overview of submodule statuses.

220.

```
git log --date=short
```

Displays commit dates in YYYY-MM-DD format.

Cleaner date output for logs.

221.

```
git worktree lock <path>
```

Locks a worktree to prevent removal.

Useful to protect important worktrees.

222.

```
git worktree unlock <path>
```

Unlocks a previously locked worktree.

Allows safe removal or modification.

223.

```
git bundle verify <file>
```

Verifies a Git bundle file's integrity and refs.

Ensures bundles can be safely used.

224.

```
git config alias.st status
```

Creates a shortcut alias `git st` for `git status`.

Improves command line efficiency.

225.

```
git log --skip=5
```

Skips the first 5 commits in the log output.

Useful for paging through history.

226.

```
git shortlog -e
```

Shows author emails in the shortlog summary.

Helps identify contributors uniquely.

227.

```
git commit --allow-empty-message -m ""
```

Creates a commit with an empty message.

Generally discouraged but sometimes needed.

228.

```
git merge-file <base> <ours> <theirs>
```

Manually merges two files using a base version.

Useful for low-level merge conflict resolution.

229.

```
git blame --show-number <file>
```

Shows line numbers along with blame output.

Helps correlate lines and commits.

230.

```
git log --first-parent
```

Shows only commits from the first parent during merges.

Useful for linear history views.

231.

```
git rev-parse --verify HEAD
```

Checks if HEAD exists and outputs its commit hash.

Useful in scripting for repository state checks.

232.

```
git bisect log
```

Shows the commands run during a bisect session.

Helps track bisect steps.

233.

```
git ls-files -c
```

Lists cached files in the index.

Helps understand staged content.

234.

```
git prune-packed
```

Removes objects that are already packed.

Helps cleanup after repacking.

235.

```
git reflog expire --expire=now --rewrite
```

Expires reflog entries with rewriting history.

Used during history cleanup.

236.

```
git merge --ff
```

Performs a fast-forward merge if possible.

Does not create a merge commit.

237.

```
git rebase --exec <command>
```

Runs a command after each commit during rebase.

Useful for automated tests or formatting.

238.

```
git remote update
```

Fetches updates for all remotes.

Keeps all remote tracking branches updated.

239.

```
git commit --no-verify
```

Skips pre-commit and commit-msg hooks.

Useful to bypass hooks temporarily.

240.

```
git fetch --all
```

Fetches updates from all remotes.

Ensures all remotes are synced.

241.

```
git show --pretty=raw <commit>
```

Shows commit data in raw format.

Includes parent commits and tree information.

242.

```
git reflog delete --rewrite HEAD@{1}
```

Deletes a specific reflog entry with rewriting.

Used to prune history selectively.

243.

```
git log --pretty=oneline --abbrev-commit
```

Shows commits in a compact single line format.

Good for quick browsing of history.

244.

```
git blame --incremental <file>
```

Outputs blame data in machine-readable format.

Useful for scripting and tooling.

245.

```
git filter-repo --path <file> --invert-paths
```

Removes a file or directory from the entire repository history.

Modern, faster alternative to git filter-branch for history rewriting.

246.

```
git rebase --interactive --autosquash HEAD~10
```

Interactively rebase last 10 commits, auto-moving fixup/squash commits.

Streamlines cleaning commit history.

247.

```
git worktree add --detach <path> <commit>
```

Creates a detached HEAD working tree at a specific commit.

Useful for testing or temporary changes without creating branches.

248.
```
git sparse-checkout init --cone
```

Initializes sparse checkout in cone mode for easier path filtering.

Improves performance and usability for large repos.

249.
```
git sparse-checkout set <dir1> <dir2>
```

Selectively checks out specified directories only.

Reduces disk usage and improves speed on large projects.

250.
```
git push origin refs/heads/<branch>:refs/heads/<branch> --force-with-lease=remote_branch
```

Force push but verify remote branch has not changed since last fetch.

Prevents overwriting others' work unintentionally.

251.
```
git checkout -b <new-branch> --track origin/<remote-branch>
```

Creates a new local branch tracking a remote branch.

Simplifies working on remote branches locally.

252.
```
git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)%Creset <%an>' --abbrev-commit --all
```

Fancy colorful graphical log of all commits with authors and relative dates.

Great for visualizing complex branching histories.

253.
```
git reflog expire --expire=now --all && git gc --prune=now --aggressive
```

Immediately expire all reflogs and aggressively garbage collect unused objects.

Frees space and cleans up repository thoroughly.

254.
```
git cherry-pick -x <commit>
```

Applies a commit from another branch, appending the original commit hash in message.

Keeps track of cherry-picked commits for traceability.

255.
```
git reset --soft HEAD~1
```

Moves HEAD back one commit but keeps changes staged.

Allows editing last commit message or contents.

256.

```
git rebase --exec "npm test"
```

Runs specified command after each commit during rebase.

Great for automated test checking during history rewriting.

257.

```
git merge --no-ff --no-commit <branch>
```

Performs merge creating a merge commit but pauses before committing.

Allows manual review or conflict resolution.

258.

```
git commit --fixup <commit-hash>
```

Creates a fixup commit targeted to a specific earlier commit.

Works with autosquash to cleanly fix past commits.

259.

```
git submodule update --remote --merge
```

Updates submodules to the latest remote commits and merges any local changes.

Keeps submodules current while preserving local modifications.

260.

```
git stash branch <branch>
```

Creates a new branch and applies the most recent stash to it.

Convenient for continuing work saved in stash.

261.

```
git push --force-with-lease=origin/<branch>
```

Force push changes only if remote branch matches local expectation.

Reduces risk of overwriting remote updates.

262.

```
git config --global credential.helper 'cache --timeout=3600'
```

Caches Git credentials for an hour.

Improves usability by reducing repeated authentication prompts.

263.

```
git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```

Displays a concise and colorful graph of commits with dates and authors.

Great for readable history inspection.

264.

```
git diff --color-words
```

Shows diffs highlighting changed words instead of whole lines.

Helps pinpoint exact changes in lines.

265.

```
git push origin --delete <branch>
```

Deletes a remote branch from the origin repository.

Useful for cleaning up merged or stale branches.

266.

```
git remote add upstream <original-repo-url>
```

Adds an upstream remote to track the original repository.

Common in fork workflows to keep updated.

267.

```
git fetch upstream
```

Fetches branches and commits from the upstream remote.

Used to keep fork in sync with original repo.

268.
```
git merge upstream/main
```

Merges upstream's main branch into the current branch.

Keeps fork updated with latest changes.

269.
```
gh pr checkout <pr-number>
```

Using GitHub CLI: Checks out a pull request locally for review.

Simplifies PR testing and development.

270.
```
gh pr create --title "Fix issue" --body "Detailed explanation" --base main --head feature-branch
```

Using GitHub CLI: Creates a new pull request from a branch.

Streamlines PR creation from command line.

271.
```
gh pr merge <pr-number> --squash --delete-branch
```

Using GitHub CLI: Merges a PR by squashing commits and deletes branch.

Helps keep history clean and branches tidy.

272.
```
git config --global alias.prune-branches '!git branch --merged | grep -v "*" | xargs -n 1 git branch -d'
```

Creates an alias to delete all local branches already merged.

Quick cleanup of stale branches.

273.
```
git ls-files --ignored --exclude-standard -o
```

Lists all ignored and untracked files in working directory.

Useful for finding files excluded by .gitignore.

274.
```
git reset --keep HEAD~1
```

Resets HEAD to previous commit, keeping local changes if possible.

Less destructive than --hard reset.

275.
```
git reflog expire --expire=30.days.ago --all
```

Expires reflog entries older than 30 days.

Helps manage reflog size and cleanup.

276.

```
git bundle create repo.bundle --branches --tags
```

Creates a bundle including all branches and tags.

Useful for offline backups or transfers.

277.

```
git gc --auto
```

Runs garbage collection automatically when necessary.

Keeps repo clean without manual intervention.

278.

```
git merge --strategy=ours <branch>
```

Merges a branch but keeps current branch content.

Useful for ignoring changes from a branch.

279.

```
git rev-list --count HEAD
```

Counts total number of commits reachable from HEAD.

Helps track repository growth.

280.

```
git clean -xdf
```

Force removes all untracked files and directories, including ignored ones.

Use with caution to avoid data loss.

281.

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

Sets VSCode as the default Git editor.

Improves workflow for VSCode users.

282.

```
git revert --no-edit <commit>
```

Reverts a commit without opening an editor for the message.

Speeds up undoing commits in scripts.

283.

```
git commit --no-gpg-sign
```

Skips GPG signing for the commit.

Overrides global or local commit signing config.

284.

```
git notes merge -s cat notes1 notes2
```

Merges two sets of notes using a specified strategy.

Useful for combining metadata on commits.

285.

```
git bundle verify repo.bundle
```

Checks integrity and references in a bundle file.

Ensures bundle is ready for cloning or fetching.

286.

```
git fetch origin +refs/pull/*/head:refs/remotes/origin/pr/*
```

Fetches all GitHub pull request branches locally.

Enables local testing of PRs without GitHub CLI.

287.

```
git sparse-checkout disable
```

Disables sparse checkout and restores full working directory.

Useful to revert partial checkout mode.

288.

```
git blame -L 20,40 <file>
```

Shows blame only for lines 20 through 40.

Helps narrow blame analysis to specific code sections.

289.

```
git worktree prune
```

Removes stale worktree administrative data for deleted directories.

Keeps repository worktree metadata clean.

290.

```
git config --global alias.co checkout
```

Creates alias 'co' for 'checkout' command.

Saves typing for frequent operations.

291.

```
git ls-files --stage
```

Shows staged files with their mode, blob SHA, and stage number.

Useful for low-level index inspection.

292.

```
git cat-file -p <object>
```

Shows the content of a Git object (blob, tree, commit).

Useful for debugging low-level Git data.

293.

```
git cherry <upstream> <branch>
```

Shows commits in branch not merged upstream.

Helps identify unique changes.

294.

```
git fetch --force
```

Forces fetching even if non-fast-forward updates happen.

Useful for overwriting local refs.

295.

```
git branch --sort=-committerdate
```

Lists branches sorted by last commit date (most recent first).

Helps find active branches quickly.

296.

```
git merge --abort
```

Aborts a conflicted merge and resets to pre-merge state.

Useful to safely cancel merges.

297.

```
git config --global user.signingkey <key-id>
```

Sets GPG key ID for signing commits globally.

Essential for verified commits.

298.

```
git rebase --onto <newbase> <oldbase> <branch>
```

Rebases branch starting after oldbase onto newbase.

Powerful for history rewriting and branch relocation.

299.

```
git push origin HEAD:refs/for/<branch>
```

Pushes changes for code review in Gerrit.

Used in Gerrit workflows.

300.

```
git update-index --assume-unchanged <file>
```

Tells Git to ignore local changes to a file.

Useful to temporarily avoid committing local modifications.

301.

```
git blame --ignore-rev <commit> <file>
```

Ignores changes from specified commit during blame.

Helps skip bulk refactoring commits.

302.

```
git commit --date="2025-08-08 20:00"
```

Overrides commit date with a specified timestamp.

Useful for backdating commits.

303.

```
git log --pretty=format:"%an - %s" --author="John Doe"
```

Shows commits by a specific author with author name and message.

Filters history for individual contributions.

304.

```
git reset --mixed HEAD~1
```

Moves HEAD back one commit and unstages changes.

Preserves working directory changes.

305.

```
git diff --stat <commit1> <commit2>
```

Shows summary of changes (files changed, insertions, deletions) between commits.

Useful for quick overview of differences.

306.

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

Enables automatic conversion of line endings for cross-platform consistency.

Prevents line-ending issues on Windows/Linux/Mac.

307.

```
git pull --rebase=interactive
```

Pulls remote changes and rebases interactively.

Allows editing and squashing commits during pull.

308.

```
git merge --no-verify <branch>
```

Skips pre-merge hooks during merge.

Useful in automated or scripted merges.

309.

```
git gc --prune=now
```

Performs garbage collection removing all unreachable objects immediately.

Frees disk space promptly.

310.

```
git show -s --format=%B <commit>
```

Shows only the commit message body for a commit.

Useful for scripting or logs.

311.

```
git archive --prefix=<folder>/ -o archive.zip HEAD
```

Creates a zip archive of current HEAD, adding a folder prefix to files.

Useful for packaged releases.

312.

```
git stash push -p
```

Interactively selects changes to stash.

Allows partial stashing of modifications.

313.

```
git ls-remote --heads origin
```

Lists all branches available in remote repository.

Useful for checking remote refs.

314.

```
git branch --merged
```

Lists branches that have been merged into the current branch.

Helps identify safe branches for deletion.

315.

```
git cherry-pick --continue
```

Continues cherry-pick after resolving conflicts.

Required step in manual conflict resolution.

316.

```
git rebase --continue
```

Continues rebase after conflicts are resolved.

Essential for multi-step rebases.

317.

```
git bisect reset
```

Ends bisect session and returns to original HEAD.

Used after bug is found or bisect aborted.

318.

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

Sets default branch name to 'main' for new repositories.

Overrides legacy 'master' default.

319.

```
git reset --hard @{u}
```

Resets current branch to its upstream state, discarding local changes.

Useful for syncing local with remote exactly.

320.

```
git remote rename origin upstream
```

Renames a remote from origin to upstream.

Useful in fork workflows for clarity.

321.

```
git tag -a v1.2.3 -m "Release 1.2.3"
```

Creates an annotated tag with message.

Preferred for marking releases with metadata.

322.

```
git show-ref --tags
```

Lists all tags and their commit hashes.

Useful for verifying tags.

323.

```
git worktree list
```

Lists all linked worktrees for the repository.

Helps manage multiple checkouts.

324.

```
git stash drop stash@{2}
```

Deletes a specific stash entry by reference.

Keeps stash list clean.

325.

```
git remote set-url origin <new-url>
```

Changes the URL of the remote named origin.

Useful when repository moves or changes remote server.

326.

```
git apply <patch-file>
```

Applies a patch file to the working directory.

Used to apply changes from external diffs.

327.

```
git rebase --skip
```

Skips the current patch during an interactive rebase.

Useful to ignore problematic commits.

328.
```
git log --grep="fix bug"
```

Searches commit messages containing "fix bug".

Helps find relevant bug fix commits.

329.
```
git log --author="Jane Doe" --since="2 weeks ago"
```

Filters commits by author and date.

Useful for recent work tracking.

330.
```
git fetch origin --prune
```

Fetches updates and removes stale remote-tracking branches.

Keeps local repo clean and in sync.

331.
```
git reflog show --date=iso
```

Shows reflog with ISO date format.

Improves readability of reflog timestamps.

332.

```
git bundle unbundle repo.bundle
```

Extracts objects and refs from a bundle file into the repo.

Used for offline repo transfer.

333.

```
git remote show origin
```

Shows detailed info about the origin remote.

Includes fetch/push URLs and tracking branches.

334.

```
git show --stat --oneline <commit>
```

Shows a summary stat and one-line commit message.

Quick overview of changes in a commit.