Github Interview Questions - Pankaj Dhapade

Absolutely, Pankaj. Here's a clean, professional set of answers for the first 15 Git interview questions, tailored to sound like a 3-year experienced DevOps engineer who understands Git in production scenarios.

---

1. What are Git and GitHub?

Answer:
Git is a distributed version control system (VCS) used to track changes in source code during software development. It allows multiple developers to collaborate efficiently by maintaining a history of every change made in a codebase. Git runs locally and supports offline commits and branching.

GitHub is a cloud-based hosting service for Git repositories. It provides collaboration features such as pull requests, issue tracking, GitHub Actions (CI/CD), and project boards. GitHub is often used to manage and share code in a team or open-source environment.

---

2. What is the difference between Git and GitHub?

Answer:

- **Git** is a command-line tool used to manage versions of code on a developer's local machine.
- **GitHub** is a web-based platform built around Git that allows teams to host, collaborate, review, and deploy code from a centralized remote repository.

In simple terms:

> *Git is the engine; GitHub is the garage where we keep the engine and collaborate with others.*

---

3. What is a Git repository? Name some popular Git hosting services.

Answer:
A Git repository is a directory that contains your project's files and the entire version history tracked by Git. It can be:

- **Local repository:** On your system.
- **Remote repository:** On platforms for collaboration.

Popular Git hosting services:

- GitHub
- GitLab
- Bitbucket

- AWS CodeCommit
- Azure Repos

---

4. What is a version control system? Mention its types.

Answer:
A **Version Control System (VCS)** is a tool that helps developers manage and track changes in code over time. It ensures collaboration, traceability, and rollback capabilities.

Types:

1. **Centralized VCS** (e.g., SVN, CVS) – One central server, all clients pull/push from there.
2. **Distributed VCS** (e.g., Git, Mercurial) – Every developer has a full copy of the repo, enabling offline work and better redundancy.

---

5. What are the main differences between Git and SVN?

Answer:

Feature	Git (Distributed)	SVN (Centralized)
Repository Type	Distributed	Centralized
Offline Access	Fully supported	Limited
Branching	Lightweight and fast	Heavy and slow
Speed	Faster	Slower
Merge Conflicts	Handled better with built-in strategies	More manual effort
Storage	Full clone of history	Only current version

In real-world DevOps, Git is the preferred choice due to its performance and collaboration advantages.

---

6. What are the advantages of using Git?

Answer:

- **Distributed development:** Every user has a full history.
- **Efficient branching/merging:** Git encourages feature-based workflows.
- **Offline capability:** Full functionality even without internet.
- **Integrity:** Every commit is checksummed using SHA-1.
- **Lightweight and fast.**
- **Integration-friendly:** Works with CI/CD tools like Jenkins, GitHub Actions, and Terraform.

---

7. What language is used in Git?

Answer:
Git is written in **C** for performance reasons. It also uses shell scripts, Perl, and Tcl for some of its helper commands and portability layers.

---

8. What are the advantages of Git over SVN?

Answer:

- **Offline operations:** Work without a central server.
- **Faster commits and diffs.**
- **Branching:** Cheaper, quicker, and more practical.
- **Data integrity:** Uses SHA-1 checksums.
- **Smaller footprint:** Stores data efficiently using packfiles.
- **Flexible workflows:** Supports feature, GitFlow, trunk-based models.

---

9. What is Git Bash?

Answer:
Git Bash is a command-line environment for Windows that provides a Unix-like interface for using Git. It includes Git CLI and essential Unix commands (e.g., ssh, cat, grep), making it easier for Windows users to work in environments similar to Linux/Unix systems.

---

10. What is the meaning of “Index” or “Staging Area” in Git?

Answer:
The **Staging Area (Index)** is a temporary holding area where changes are placed before they are committed. You use `git add` to move changes to the staging area, and `git commit` to save them to the repository history. This provides control over what changes are committed.

---

11. What is tagging in Git?

Answer:
A **Git tag** is a reference to a specific commit. Tags are used to mark important points like version releases (`v1.0`, `v2.3.1`). Unlike branches, tags do not move—they point to a fixed commit.

Example:

```
git tag v1.0
git push origin v1.0
```

---

12. What is forking in Git?

Answer:
Forking is creating a personal copy of someone else's GitHub repository into your GitHub account. It allows developers to experiment independently without affecting the original codebase. It's commonly used in open-source contributions.

---

13. What is the use of a Git clone?

Answer:
`git clone` creates a local copy of a remote repository, including all files, branches, and history.

Example:

```
git clone https://github.com/user/project.git
```

In DevOps workflows, this is often the first step when onboarding a new project into a CI/CD pipeline.

---

14. What is the function of 'git config'?

Answer:
`git config` is used to set configuration variables for Git, such as username, email, editor, merge tool, etc.

Example:

```
git config --global user.name "Pankaj DevOps"
git config --global user.email "pankaj@example.com"
```

It supports three levels:

- System (`--system`)
- Global (`--global`)
- Local (per repository)

---

15. What is the process for creating a repository in Git?

Answer:
Creating a local repository:

```
mkdir myproject
cd myproject
git init
```

Steps in practice:

1. Initialize with `git init`

2. Create `.gitignore` to exclude files
3. Add files: `git add .`
4. Commit: `git commit -m "Initial commit"`

To link it with a remote:

```
git remote add origin https://github.com/user/myproject.git
git push -u origin main
```

Let me know when you're ready for questions **16 to 30**, and I'll continue in the same format — clear, practical, and aligned with real-world DevOps experience.

Perfect. Here are the next set of answers: **Git Interview Questions 16 to 30**, written with the perspective of a DevOps engineer with 3 years of production experience.

---

16. What is cherry-pick in Git?

Answer:
`git cherry-pick` is used to apply the changes from a specific commit on one branch to another branch without merging the entire branch.

Example use case in DevOps:
You fixed a critical bug in `dev`, and now want to apply just that fix to `main`:

```
git checkout main
git cherry-pick <commit-hash>
```

It helps avoid unnecessary code when you don't want to merge full feature branches.

---

17. What is origin in Git?

Answer:
`origin` is a default alias for the remote repository from which a project was cloned or initialized.

For example, after cloning:

```
git clone https://github.com/user/project.git
```

Git automatically sets the remote repository to `origin`.

You can verify this with:

```
git remote -v
```

In CI/CD or team environments, you often push or pull using `origin`.

---

18. What is the git push command?

Answer:
`git push` uploads your local commits to the remote repository (like GitHub).

Basic usage:

```
git push origin main
```

This is essential in a DevOps context where CI/CD pipelines trigger on remote changes (e.g., in GitHub Actions or Jenkins).

---

19. What is the git pull command?

Answer:
`git pull` is a combination of `git fetch` and `git merge`. It downloads the latest changes from the remote branch and merges them into your local branch.

```
git pull origin main
```

It's commonly used by developers and DevOps engineers to sync with the latest codebase.

---

20. What is the difference between git fetch and git pull?

Feature	`git fetch`	`git pull`
Action	Downloads changes only	Downloads and merges
Safer?	Yes – does not modify working directory	Less safe – auto merges
Use case	When you want to review changes before applying	When you're ready to merge immediately

In production, it's recommended to use `fetch` and review before merge to avoid unexpected breakage.

---

21. Explain git checkout in Git.

Answer:
`git checkout` is used to:

- Switch branches
- Restore files
- Detach HEAD to point to a specific commit

Example:

```
git checkout feature/login
```

Or restore a file:

```
git checkout main -- file.txt
```

In practice, it’s widely used to jump between branches during troubleshooting, feature work, or testing.

---

22. What does git rebase do?

Answer:
`git rebase` is used to apply changes from one branch on top of another, creating a linear commit history.

```
git checkout feature
git rebase main
```

Why it's important in DevOps:
Rebasing keeps history clean and avoids unnecessary merge commits, which is critical in large CI/CD workflows.

---

23. What is the difference between git rebase and git merge?

Feature	`git merge`	`git rebase`
History	Preserves complete history	Rewrites history
Merge commits	Creates a merge commit	No merge commit
Use case	Team collaboration with traceability	Clean history, personal branches

In production, merge is safer for collaborative work. Rebase is best for local cleanup before merging.

---

24. What is revert in Git?

Answer:
`git revert` creates a new commit that undoes the changes of a previous commit, without altering the commit history.

Example:

```
git revert <commit-id>
```

It’s commonly used in production to roll back a faulty change without rewriting Git history (safe for shared branches like `main`).

---

25. What is the difference between resetting and reverting?

Feature	`git reset`	`git revert`
History	Rewrites history	Preserves history
Risk	Dangerous on shared branches	Safe on all branches
Purpose	Move HEAD/branch pointer	Undo changes via new commit

In a team or CI/CD pipeline, **revert** is preferred. **Reset** is suitable only for local or isolated branches.

---

26. What is the difference between ‘git remote’ and ‘git clone’?

- `git clone` creates a full local copy of a remote repository and auto-adds `origin` as a remote.
- `git remote` is used to **view**, **add**, or **manage** remote repositories in an existing local repo.

Example:

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

Use `git remote` when linking an existing project to a remote repo.

---

27. What is Git stash?

Answer:
`git stash` temporarily saves your uncommitted changes so you can work on something else (like switching branches) without losing progress.

Example:

```
git stash
git checkout main
```

Later:

```
git stash apply
```

In real use, this is handy when an urgent hotfix comes and your work is in an unstable state.

---

28. What is Git stash drop?

Answer:
`git stash drop` deletes a specific stash entry from the stash list.

Example:

```
git stash list
git stash drop stash@{0}
```

This is done to clean up unused stashes and avoid memory bloat.

---

29. Why do we need branching in Git?

Answer:
Branching allows developers to work independently on features, bug fixes, or experiments without affecting the main codebase.

Benefits in DevOps:

- Enables parallel development
- Supports CI/CD feature environments
- Enables GitOps, version control of infrastructure

Typical branching strategies:

- Feature branches
- Hotfix branches
- GitFlow or trunk-based development

---

30. What is HEAD in Git, and how many HEADs can be created in a repository?

Answer:
HEAD is a reference to the current commit (typically the tip of the current branch).

Each Git repository has only **one active HEAD** at a time. It can point to:

- A branch (e.g., `main`)
- A specific commit (detached HEAD)

In CI/CD, understanding HEAD is crucial when checking out commits or tags for deployment.

Let me know when you're ready for questions **31 to 45**, and I'll continue the series with clean, practical, and interview-ready answers.

Great, Pankaj. Let's continue with the next set of Git interview questions — **questions 31 to 45**, answered in a structured, DevOps-aware, and professional tone.

---

31. What is the regular way for branching in Git?

Answer:
The typical way to create and work on branches in Git is:

```
git checkout -b feature/login
```

This creates and switches to a new branch named `feature/login`. A common Git branching workflow in DevOps includes:

- `main` or `master` for production-ready code
- `dev` or `develop` for integration/testing
- Feature branches (`feature/*`)
- Hotfix branches (`hotfix/*`)
- Release branches (`release/*`)

---

32. State a way to create a new branch in Git.

Answer:
You can create a new branch using:

```
git branch feature/ui-redesign
```

To switch to that branch:

```
git checkout feature/ui-redesign
```

Or both together (preferred):

```
git checkout -b feature/ui-redesign
```

This is standard in CI/CD pipelines where developers isolate features or hotfixes before merging.

---

33. How do you define a 'conflict' in Git?

Answer:
A **Git conflict** occurs when:

- Two branches have changes in the same part of the file
- Git cannot automatically merge them

Example: If `main` and `feature1` both modify line 12 of `app.py`, Git will raise a conflict when merging.

It's common during:

- `git merge`
- `git rebase`
- Cherry-picking

---

34. How to resolve a conflict in Git?

Answer:

1. Run a merge/rebase that causes a conflict:

```
git merge feature1
```

2. Git marks the file with conflict markers:

```
<<<<<<< HEAD
your change
=======
their change
>>>>>>> feature1
```

3. Manually edit and resolve.
4. Mark as resolved:

```
git add filename
```

5. Continue:

```
git commit   # or git rebase --continue
```

In teams, it's critical to resolve conflicts early and test thoroughly before pushing.

---

35. What is Git, and what is it used for?

Answer:
Git is a distributed version control system used to track changes in code across time and collaborate with teams. It enables:

- Version history
- Code rollback
- Branching and merging
- Conflict resolution
- Integrations with CI/CD, automation, and DevOps pipelines

It is the backbone of modern software delivery practices.

---

36. What is the difference between Git and GitHub?

Answer:

Feature	Git	GitHub
Type	CLI tool	Web-based hosting
Scope	Local version control	Remote collaboration

Feature	Git	GitHub
Internet required	No	Yes
Use case	Code tracking	Code sharing, pull requests, CI/CD
Maintained by	Open-source	Microsoft

They work together: Git manages code, GitHub helps collaborate and deploy.

---

37. How do you initialize a new Git repository?

Answer:
To initialize a Git repository locally:

```
mkdir myapp
cd myapp
git init
```

This creates a `.git` directory and starts tracking changes.

In projects, this is often followed by:

- Creating `.gitignore`
- Adding initial code
- First commit:

```
git add .
git commit -m "Initial commit"
```

---

38. What is the purpose of the `.gitignore` file?

Answer:
`.gitignore` tells Git which files/folders to **exclude from tracking**.

Common use cases:

- IDE config files (`.vscode/`, `.idea/`)
- Build artifacts (`/dist`, `/target`)
- Secrets (`.env`, `*.pem`)

Example:

```
*.log
.env
__pycache__/
```

This is critical in DevOps to avoid leaking sensitive files into version control.

---

39. How do you commit changes in Git?

Answer:
Standard commit flow:

```
git add .
git commit -m "Describe the change"
```

- `git add` stages the changes.
- `git commit` saves them in Git history.

Follow **conventional commit messages** in production:

```
git commit -m "fix(api): handle 500 error from backend"
```

---

40. What is a branch in Git, and how do you create a new one?

Answer:
A **branch** is an independent line of development. It allows developers to work on new features or fixes without affecting the main codebase.

To create a new branch:

```
git checkout -b feature/add-login
```

Then push it:

```
git push origin feature/add-login
```

Branches are key to feature isolation, testing, and structured CI/CD.

---

41. How do you merge two branches in Git?

Answer:

1. Switch to the branch you want to merge into (usually `main`):

```
git checkout main
```

2. Merge another branch (e.g., `feature/ui`):

```
git merge feature/ui
```

3. Resolve any conflicts, then commit if needed.

This is part of the DevOps workflow before deploying to staging or production.

42. What is the difference between `git pull` and `git fetch`?

Answer:

Command	Description	When to Use
`git pull`	Fetches + auto-merges	Quick sync
`git fetch`	Fetches only, no merge	Safe sync & review

In production workflows, it's safer to use:

```
git fetch
git log origin/main
```

Then merge manually if needed.

---

43. How do you resolve conflicts in a Git merge?

Answer:

1. Attempt merge:

```
git merge feature1
```

2. Git shows conflicted files.
3. Open the file and resolve marked conflict sections.
4. Mark file as resolved:

```
git add conflicted_file.py
```

5. Finish merge:

```
git commit -m "Resolved merge conflict"
```

Always test before pushing after conflict resolution.

---

44. What is the purpose of `git remote add origin`?

Answer:
It links your local Git repo to a remote (usually on GitHub, GitLab, etc.).

Example:

```
git remote add origin https://github.com/pankaj-devops/myapp.git
```

Then push:

```
git push -u origin main
```

This command is typically used after `git init` in a new repo to connect it with the remote.

---

45. What is the difference between `git reset` and `git revert`?

Feature	`git reset`	`git revert`
Purpose	Move HEAD backwards	Undo changes via new commit
History	Rewrites history	Preserves history
Use on shared branches	No (unsafe)	Yes (safe)
Commit ID needed?	Yes	Yes

> In team environments, **use** `revert`. Reset is only safe locally.

Let me know when you're ready for questions **46 to 64**, and I'll continue with clean, real-world-focused answers suitable for interviews and on-the-job readiness.

Absolutely, Pankaj. Let's finish the final set of **Git Interview Questions 46 to 64**, with detailed, DevOps-aware, production-friendly answers. These answers are suitable for 3-year experience-level interview conversations.

---

46. How do you delete a branch in Git?

Answer:

- **Local branch:**

```
git branch -d feature/login
```

Use `-D` to force delete without checking for merge:

```
git branch -D feature/login
```

- **Remote branch:**

```
git push origin --delete feature/login
```

In CI/CD environments, cleaning up unused branches helps maintain clarity and reduce risk of stale deployments.

---

47. What is a Git tag, and how do you create one?

Answer:
A **tag** marks a specific point in history, often used for releases.

- **Create lightweight tag:**

```
git tag v1.0
```

- **Create annotated tag:**

```
git tag -a v1.0 -m "First release"
```

- **Push tag:**

```
git push origin v1.0
```

Tags are commonly used in production deployments and version-controlled infrastructure rollouts.

---

48. How do you squash multiple commits into one?

Answer:
Use interactive rebase:

```
git rebase -i HEAD~3
```

Then change:

```
pick 1st-commit
squash 2nd-commit
squash 3rd-commit
```

Squashing is used in production workflows to clean up commit history before merging, especially in PRs.

---

49. What is a Git submodule, and how do you add one?

Answer:
A **Git submodule** allows one Git repository to include another Git repo as a subdirectory.

Add one:

```
git submodule add https://github.com/example/library.git lib/
```

Use when:

- You want to track another repo as part of your project.
- Common in mono-repo or shared-library setups.

Caution: Managing submodules requires discipline in CI/CD workflows.

---

50. How do you cherry-pick a commit from one branch to another?

Answer:

```
git checkout main
git cherry-pick <commit-hash>
```

Use when:

- You want to apply a specific fix/feature from a different branch without merging everything.

This is common in hotfixes or selective backporting in DevOps pipelines.

---

51. What is the difference between `git stash` and `git reset`?

Feature	`git stash`	`git reset`
Use Case	Temporarily save changes	Undo commits or staging
Impact	Keeps history safe	Can alter history
Safer?	Yes	Risky if not used carefully

In real-world CI/CD, `stash` is safe to switch tasks. `reset` is usually for cleanup before pushing.

---

52. How do you create a new Git repository from an existing one?

Answer:
If you want a **clean, fresh repo**:

```
git clone --bare https://github.com/old/repo.git
cd repo.git
git push --mirror https://github.com/new/repo.git
```

Or copy `.git`:

```
cp -r old_project/.git new_project/
cd new_project
git reset --hard
```

This technique is useful for forking internal repositories or splitting monorepos.

---

53. What is GitFlow, and how does it differ from GitHub Flow?

Feature	GitFlow	GitHub Flow
Branches	`feature`, `develop`, `release`, `hotfix`, `master`	`main`, `feature`, PR-based
Complexity	High	Simple

Feature	GitFlow	GitHub Flow
Suitable For	Big teams, versioned releases	CI/CD, startups, modern agile
Usage	Used in enterprise DevOps	Used in cloud-native teams

DevOps teams now prefer GitHub Flow or trunk-based development for faster releases.

---

54. How do you configure Git to use a proxy server?

Answer:

```
git config --global http.proxy http://proxyuser:password@proxy.example.com:8080
git config --global https.proxy https://proxyuser:password@proxy.example.com:8080
```

This is essential in corporate environments where outbound connections are restricted.

---

55. How do you find the SHA of a commit in Git?

Answer:

```
git log
```

You'll see:

```
commit d4e7f3bfa45e1cd3e3f2...
Author: ...
Date: ...
Message
```

You can also use:

```
git rev-parse HEAD
```

SHA is used in cherry-pick, revert, reset, etc.

---

56. What is the purpose of `git clean`?

Answer:
`git clean` removes **untracked files** and directories.

Preview:

```
git clean -n
```

Execute:

```
git clean -f -d
```

This is helpful before a build or deployment to ensure a clean working tree.

---

57. How do you create an empty Git repository?

Answer:

```
mkdir newrepo
cd newrepo
git init
```

Now the folder becomes a Git repo. Useful when starting a project or initializing infrastructure as code.

---

58. What is the difference between `git diff` and `git status`?

Command	Purpose
`git diff`	Shows actual line-by-line changes
`git status`	Shows file-level summary (modified, staged, etc.)

Used in DevOps to audit changes before commits or during incident fixes.

---

59. How do you configure Git to ignore file permissions?

Answer:

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

This is especially useful on Windows/Linux cross-environments where file mode changes can clutter diffs.

---

60. What is a Git hook, and how do you use one?

Answer:
A **Git hook** is a script that runs automatically at specific Git events.

Example: pre-commit hook to check linting.
Path:

```
.git/hooks/pre-commit
```

Use case:

- Block bad commits
- Enforce formatting (e.g., Black, ESLint)
- Integrate secret scanning

Used heavily in secure DevOps pipelines.

---

61. How do you change the author of a commit in Git?

Answer:

```
git commit --amend --author="New Name <new@example.com>"
```

Or for multiple commits:

```
git rebase -i HEAD~3
# then edit commit and change author
```

Important for compliance in enterprise repos.

---

62. What is the purpose of `git bisect`?

Answer:
`git bisect` helps find the commit that introduced a bug by binary search.

```
git bisect start
git bisect bad
git bisect good <commit>
```

This is helpful in incident debugging or CI/CD regressions.

---

63. How do you create a Git repository from a folder?

Answer:
If you have an existing folder:

```
cd myfolder
git init
git add .
git commit -m "Initial import"
```

To link with remote:

```
git remote add origin <url>
git push -u origin main
```

---

64. What is the difference between `git rebase` and `git merge`?

Feature	`git rebase`	`git merge`
History	Rewrites	Preserves

Feature	`git rebase`	`git merge`
Merge commit	No	Yes
Linear history	Yes	No
Usage	Clean up feature branches	Combine team work

Best Practice: Use `rebase` on local branches before merge, use `merge` for shared/public branches.

✅ **Done!**

You now have a complete, interview-ready, DevOps-level understanding of all 64 Git interview questions.

Would you like me to format this into a **PDF cheat sheet** or a **Notion-ready document** for easy revision?

Also, we can move on to **realistic Git-based scenarios** used in production (e.g., rollback via Git, GitOps workflows, GitHub Actions triggers, secure Git workflows) if you'd like.