Git Questions for DevOps Engineers

1. What is Git and why is it used?

- **Answer:** Git is a **distributed version control system** used to track code changes, collaborate with teams, and manage multiple versions of code efficiently.

---

2. Difference between Git and GitHub?

- **Answer:**
    - **Git** → version control system for tracking changes locally.
    - **GitHub** → cloud-based platform for hosting Git repositories and collaboration.

---

3. What is a Git branch and why is it used?

- **Answer:** Branches allow **parallel development** without affecting the main code. Example:

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

---

4. How do you merge a branch into main/master?

- **Answer:**

```
git checkout main
git merge feature-login
```

- Resolve conflicts if any arise.

---

5. What is Git rebase and when to use it?

- **Answer:** Rebase **moves your branch changes on top of another branch** to keep a clean commit history.

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

---

6. How do you resolve merge conflicts?

- **Answer:**
    1. Git will mark conflicts in files.
    2. Edit the file to keep desired changes.
    3. Stage and commit: `git add <file>`

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

---

7. How do you see commit history?

- **Answer:**

```
git log                  # detailed commit history
git log --oneline --graph  # simple visual graph
git log -p               # shows changes in each commit
```

---

8. How do you undo a commit?

- **Answer:**

```
git reset --soft HEAD~1    # undo last commit, keep changes staged
git reset --hard HEAD~1    # undo last commit, discard changes
```

---

9. How do you clone a repository?

- **Answer:**

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

---

10. How do you stash changes temporarily?

- **Answer:**

  ```
  git stash          # stash current changes
  git stash pop      # apply stashed changes
  git stash list     # view stashes
  ```

---

11. How do you check the status of your repository?

- **Answer:**

  ```
  git status       # shows staged/unstaged changes
  git diff         # shows changes not staged
  git diff --staged # shows changes staged for commit
  ```

---

12. How do you pull latest changes from remote?

- **Answer:**

  ```
  git pull origin main
  ```

- Pulls latest commits from the remote repository and merges with local branch.

---

13. How do you push changes to remote repository?

- **Answer:**

  ```
  git push origin main
  ```

- Push commits from local branch to the remote repository.

---

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

- **Answer:**
  - `git fetch` → downloads commits from remote but **does not merge**.
  - `git pull` → downloads and **merges commits** automatically.

---

15. How do you create a tag in Git?

- **Answer:**

  ```
  git tag v1.0        # lightweight tag
  git tag -a v1.0 -m "Version 1.0"  # annotated tag
  git push origin v1.0  # push tag to remote
  ```

==

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

- **Answer:**
  - `git merge` → combines branches, may create a merge commit.
  - `git rebase` → moves branch changes on top of another branch, keeps a linear history.

---

17. What is the difference between `git reset`, `git revert`, and `git checkout`?

- **Answer:**
  - `git reset` → undo commits locally (can remove history).
  - `git revert` → creates a new commit to undo changes safely.
  - `git checkout` → switch branches or restore files.

---

18. What are Git hooks?

- **Answer:** Scripts triggered by Git actions (commit, push, merge). Commonly used for:
  - Code linting before commit (`pre-commit`)
  - Running tests (`pre-push`)

        - Enforcing commit message format

---

19. Explain Gitflow workflow.

- **Answer:** Gitflow is a **branching strategy**:
    - `main` → production
    - `develop` → integration branch
    - `feature/*` → new features
    - `release/*` → release preparation
    - `hotfix/*` → urgent fixes in production

---

20. How do you squash commits? Why?

- **Answer:** Squashing combines multiple commits into one for **cleaner history**:

```
git rebase -i HEAD~3
# choose 'squash' for commits to merge
```

---

21. How do you resolve detached HEAD state?

- **Answer:**

```
git checkout main           # switch back to branch
git branch new-branch       # save changes if needed
```

- Detached HEAD occurs when checking out a commit directly.

---

22. What is a fork and how is it different from a clone?

- **Answer:**
    - **Fork** → creates a personal copy on GitHub to contribute.
    - **Clone** → local copy of a repository for working on your machine.

---

23. How do you check differences between branches?

- **Answer:**

```
git diff main..feature-login
git log main..feature-login --oneline
```

---

24. How do you remove a file from Git but keep it locally?

- **Answer:**

```
git rm --cached filename
git commit -m "Remove file from repo but keep locally"
```

---

25. How do you undo a pushed commit?

- **Answer:**

```
git revert <commit_id>    # safe, creates a new commit
git reset --hard <commit_id>   # force reset (careful!)
git push origin main --force   # force push if reset
```

---

26. How do you cherry-pick a commit from another branch?

- **Answer:**

```
git checkout main
git cherry-pick <commit_id>
```

- Useful to bring specific commits without merging the entire branch.

---

27. How do you handle large files in Git?

- **Answer:** Use **Git LFS (Large File Storage)**:

```
git lfs install
git lfs track "*.zip"
git add .gitattributes
git commit -m "Track large files"
```

---

28. What is a fast-forward merge?

- **Answer:** A merge without a merge commit, happens when the branch is **directly ahead of the target**:

```
git checkout main
git merge feature-branch  # no extra commit if no divergence
```

---

29. How do you configure Git globally and locally?

- **Answer:**

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config user.name "Local Name"   # local repo config
```

---

30. How do you rollback a file to a previous commit?

- **Answer:**

  ```
  git checkout <commit_id> -- filename
  git commit -m "Rollback file to previous version"
  ```

31. How do you handle rewriting commit history in a shared repository?

Answer:

- Rewriting history (via `git rebase`, `git commit --amend`, `git reset`) should be avoided on shared branches like `main`.
- If necessary on feature branches:

  ```
  git rebase -i HEAD~3
  git push --force-with-lease
  ```

- Use `--force-with-lease` (safer than `--force`) to avoid overwriting others' work.

---

32. How do you bisect commits to find a bug?

Answer:
Use `git bisect` for binary search between known good and bad commits:

```
git bisect start
git bisect bad HEAD
git bisect good <commit_id>
# Git checks out commits until bug is isolated
git bisect reset
```

This speeds up debugging large histories.

---

33. How do you sign commits and why?

Answer:

- Signing ensures authenticity and trust in commits.
- Configure GPG key:

  ```
  git config --global user.signingkey <key-id>
  git commit -S -m "Signed commit"
  ```

- Required in secure environments and open-source projects.

---

34. What is a shallow clone and when would you use it?

Answer:

- A **shallow clone** limits history depth for faster and lighter checkout:

```
git clone --depth 1 https://github.com/user/repo.git
```

- Useful in CI/CD pipelines where only the latest snapshot is needed.

---

35. How do you handle Git submodules?

Answer:

- Submodules embed another Git repo inside your repo.
- Workflow:

```
git submodule add <repo-url>
git submodule update --init --recursive
```

- Common in monorepos and when managing dependencies as separate repos.

---

36. How do you recover a deleted branch?

Answer:

- Use reflog to find the last commit:

```
git reflog
git checkout -b branch_name <commit_id>
```

- Git rarely loses data immediately; reflog helps restore.

---

37. What is Git reflog and how is it useful?

Answer:

- `git reflog` tracks branch movements and HEAD changes.
- Example:

```
git reflog
```

- Used to recover lost commits, undo resets, and restore deleted branches.

---

38. How do you perform an interactive rebase and why?

Answer:

- Interactive rebase lets you reorder, squash, or edit commits:

```
git rebase -i HEAD~5
```

- Choose options (`pick`, `squash`, `edit`) to rewrite history.
- Useful for cleaning messy commits before merging to main.

---

39. What’s the difference between `git reset --soft`, `--mixed`, and `--hard`?

Answer:

- `--soft`: Moves HEAD, keeps changes staged.
- `--mixed` (default): Moves HEAD, unstages changes but keeps them in working directory.
- `--hard`: Moves HEAD and deletes changes permanently.

---

40. How do you optimize a large Git repository?

Answer:

- Remove large files → Use Git LFS.
- Clean old objects:

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

- Use shallow clones in pipelines.
- Split history with `git filter-repo` (replacement for `git filter-branch`).

41. How do you recover a commit after a hard reset?

Answer:

- Use **reflog** to find commit ID:

  ```
  git reflog
  git checkout -b recovery <commit_id>
  ```

- This works because Git keeps references even after resets.

---

42. How do you handle binary files in Git efficiently?

Answer:

- Use **Git LFS (Large File Storage)**:

  ```
  git lfs install
  git lfs track "*.iso"
  ```

- Prevents repo bloat from large binaries.

---

43. What’s the difference between `git fetch --prune` and `git gc`?

Answer:

- `git fetch --prune`: Removes references to remote branches deleted on the server.

- `git gc`: Garbage collection; cleans unnecessary objects and optimizes repo.

---

44. How do you squash commits across multiple branches?

Answer:

- Use `git rebase -i <base-branch>` to squash commits into one.
- Or, when merging:

  ```
  git merge --squash feature-branch
  ```

- Keeps main history clean.

---

45. How do you handle diverged branches in Git?

Answer:

- If both branches have unique commits:

  ```
  git pull --rebase
  ```

- If you need a merge commit:

  ```
  git merge origin/main
  ```

- Always check `git status` before resolving.

---

46. What is a bare repository and when do you use it?

Answer:

- Bare repo: Has no working directory, only `.git` objects.
- Used as a **remote repository** for collaboration.

  ```
  git init --bare repo.git
  ```

---

47. How do you migrate a repository while preserving commit history?

Answer:

- Clone with `--mirror`:

  ```
  git clone --mirror old-repo.git
  cd old-repo.git
  git push --mirror new-repo.git
  ```

- Preserves **branches, tags, and history**.

---

48. How do you enforce branch policies with Git hooks?

Answer:

- Example: Prevent commits directly to `main`:

  ```
  # in .git/hooks/pre-commit
  if [ "$(git rev-parse --abbrev-ref HEAD)" = "main" ]; then
    echo "Direct commits to main are not allowed."
    exit 1
  fi
  ```

- Teams often integrate with **pre-push hooks** or **CI checks**.

---

49. How do you split a Git repository into multiple smaller repos?

Answer:

- Use `git filter-repo` (modern replacement for `git filter-branch`):

  ```
  git filter-repo --path src/module1/ --path-rename src/module1/:
  ```

- Useful in breaking monolith repos into microservice repos.

---

50. How do you combine multiple repositories into a monorepo?

Answer:

- Use `git subtree` or `git filter-repo`:

  ```
  git remote add repo2 <url>
  git fetch repo2
  git subtree add --prefix=repo2/ repo2 main
  ```

- Keeps full commit history for each imported repo.

---

51. How do you deal with “dangling commits”?

Answer:

- Dangling commits = commits not reachable from any branch.
- Find with:

  ```
  git fsck --lost-found
  ```

- Recover with:

  ```
  git checkout <dangling_commit_id>
  ```

---

52. How do you detect and resolve performance issues in very large repositories?

Answer:

- Use shallow clones in CI/CD.
- Run garbage collection:

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

- Split repo if needed (monorepo → microrepo).
- Use partial clone:

```
git clone --filter=blob:none <repo>
```

---

53. How do you clean sensitive data (like passwords) from Git history?

Answer:

- Use `git filter-repo`:

```
git filter-repo --replace-text passwords.txt
```

- Then force push:

```
git push --force --all
```

- Rotate credentials because old clones may still have them.

---

54. How do you enforce commit message standards?

Answer:

- Use a `commit-msg` hook:

```
# .git/hooks/commit-msg
if ! grep -qE "^(feat|fix|chore|docs|style|refactor|test):" "$1"; then
  echo "Commit message must follow Conventional Commits."
  exit 1
fi
```

- Or enforce via **CI/CD pipeline checks**.

---

55. How do you rollback a merge commit?

Answer:

- If you want to undo a merge:

```
git revert -m 1 <merge_commit_id>
```

- `-m 1` → keeps parent branch (usually main).

---

56. How do you sync a fork with the upstream repo?

Answer:

```
git remote add upstream https://github.com/original/repo.git
git fetch upstream
git checkout main
git merge upstream/main
git push origin main
```

---

57. What is Git worktree and why is it useful?

Answer:

- Git worktree allows multiple working directories for the same repo.
- Example:

  ```
  git worktree add ../feature-branch feature-branch
  ```

- Useful for checking out multiple branches at once without cloning again.

---

58. How do you debug “detached HEAD” issues?

Answer:

- Detached HEAD happens when you checkout a commit instead of a branch.
- Fix:

  ```
  git checkout -b new-branch
  ```

- Or switch back to an existing branch:

  ```
  git checkout main
  ```

---

59. How do you force Git to track only specific folders in a repo?

Answer:

- Use `.gitignore` to exclude unnecessary files.
- Or use **sparse checkout**:

  ```
  git sparse-checkout init
  git sparse-checkout set folder1/ folder2/
  ```

- Saves disk space and checkout time.

---

60. How do you perform a `git push` for only tags and not commits?

Answer:

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

- Pushes all tags without pushing commits.

61. How do you enforce code reviews before merging into `main`?

Answer:

- Use **branch protection rules** (e.g., in GitHub/GitLab/Bitbucket).
- Require PR approval, status checks (CI/CD pipelines), and signed commits.
- Prevents direct pushes into protected branches.

---

62. How do you integrate Git with CI/CD pipelines?

Answer:

- CI/CD tools (GitHub Actions, GitLab CI, Jenkins, CircleCI) trigger workflows on Git events:
    - `push` → build/test pipeline.
    - `pull_request` → PR validation pipeline.
    - `tag` → release pipeline.

---

63. What is GitOps and how does Git fit into it?

Answer:

- GitOps = using Git as the **single source of truth** for infrastructure & app configuration.
- Tools like ArgoCD / Flux continuously sync cluster state from Git repos.
- All deployments are Git-driven via commits & PRs.

---

64. How do you automate versioning in Git for releases?

Answer:

- Use **semantic versioning** tags with scripts or GitHub Actions:

  ```
  git tag v1.2.0
  git push origin v1.2.0
  ```

- CI/CD pipelines can auto-generate release notes & Docker image tags.

---

65. How do you handle hotfixes in GitFlow?

Answer:

- Hotfix branch → created from `main`, merged back into both `main` and `develop`.

  ```
  git checkout main
  git checkout -b hotfix/login-bug
  ```

- Ensures production and future releases both get the fix.

---

66. How do you maintain long-lived branches in large teams?

Answer:

- Use **rebase or regular merges** from `main` to keep up-to-date.
- Avoid stale branches by deleting old feature branches post-merge.
- Use automation to close inactive PRs.

---

67. How do you prevent secrets from being committed into Git?

Answer:

- Use **pre-commit hooks** with tools like `git-secrets` or `trufflehog`.
- Scan repos with CI pipelines.
- Enforce secret detection in PR checks.

---

68. How do you rollback a failed deployment in a GitOps setup?

Answer:

- Rollback = `git revert` the faulty commit → GitOps tool syncs cluster back to last good state.
- Example:

  ```
  git revert <commit_id>
  git push origin main
  ```

---

69. What is trunk-based development and how is it different from GitFlow?

Answer:

- **Trunk-based**: Small, frequent commits to `main`, feature toggles used.
- **GitFlow**: Heavy branching (feature, release, hotfix).
- Trunk-based is common in **CI/CD, DevOps, microservices**.

---

70. How do you deal with a repo growing too large over time?

Answer:

- Split repo into smaller services (microrepos).
- Use **Git LFS** for large files.
- Use `git filter-repo` to purge old/unwanted history.
- Consider **monorepo tooling** (Nx, Bazel) if repo consolidation is intentional.

---

71. How do you enforce a linear history in `main`?

Answer:

- Enforce **rebase strategy** for merging PRs.
- In GitHub: enable “Rebase and merge” only, disable merge commits.
- Keeps commit history clean & linear.

---

72. How do you set up a Git mirror for disaster recovery?

Answer:

- Use `--mirror` cloning:

  ```
  git clone --mirror https://github.com/org/repo.git
  git push --mirror backup-repo.git
  ```

- Automate sync with cron/CI jobs.

---

73. How do you sync a monorepo to multiple microservices?

Answer:

- Use **Git subtree** or CI jobs to split folders into service repos.
- Example with subtree split:

  ```
  git subtree split --prefix=service1 -b service1-branch
  ```

---

74. How do you apply GitOps in multi-environment deployments (dev, staging, prod)?

Answer:

- Use **separate branches/repos** for environments.
  - `main` → production

    - `staging` → staging
    - `dev` → development
- PRs promote changes between environments.

---

75. How do you enforce commit conventions across teams?

Answer:

- Adopt **Conventional Commits** (feat, fix, chore).
- Use a `commit-msg` hook or lint tool (`commitlint`).
- CI/CD fails if commit messages don't follow standard.

---

76. How do you detect unused branches automatically?

Answer:

- Use GitHub API or Git CLI to check stale branches:

  ```
  git branch -r --merged
  ```

- Automation removes branches merged into `main`.

---

77. How do you manage multiple remotes in Git?

Answer:

- Example: One repo on GitHub, another on GitLab:

  ```
  git remote add github <url>
  git remote add gitlab <url>
  git push github main
  git push gitlab main
  ```

- Useful for **multi-cloud redundancy**.

---

78. How do you configure Git in CI/CD pipelines for automation?

Answer:

- Use bot users with SSH keys or tokens.
- Example:

  ```
  git config user.name "CI Bot"
  git config user.email "ci-bot@company.com"
  ```

- Needed for pipelines that commit version bumps or changelogs.

---

79. How do you manage secrets in `.gitconfig` or Git credentials?

Answer:

- Use **credential helpers**:

  ```
  git config --global credential.helper store
  git config --global credential.helper cache
  ```

- In DevOps: store tokens in **vaults** (AWS Secrets Manager, HashiCorp Vault, K8s Secrets).

---

80. How do you ensure reproducibility in builds from Git?

Answer:

- Pin builds to **specific commits** or **tags** (not branches).
- Example in CI:

  ```
  git checkout <commit_id>
  ```

- Ensures identical builds even if `main` moves forward.

Extreme/Scenario-Based Git FAQs

81. How do you debug a pipeline failure caused by incorrect Git shallow clones?

Answer:

- CI pipelines often use shallow clones (`--depth=1`) to save time.
- Some tasks (changelog generation, semantic release) need full history.
- Fix:

  ```
  git fetch --unshallow
  ```

---

82. How do you handle Git conflicts in CI/CD automated merges?

Answer:

- Use **rebase + auto-merge strategies** in pipelines.
- Example with GitHub Actions:

  ```
  git merge origin/main --strategy-option theirs
  ```

- Or fail the pipeline and require manual intervention.

---

83. How do you implement GitOps drift detection?

Answer:

- Tools like **ArgoCD/Flux** compare cluster state vs Git state.
- If drift is found (manual cluster changes), the tool alerts or auto-reverts.
- DevOps workflow = Git is always the source of truth.

---

84. How do you enforce signed commits in GitHub/GitLab?

Answer:

- Enable “Require signed commits” in branch protection.
- Developers must configure GPG/SSH signing.
- Prevents impersonation attacks in open-source/enterprise projects.

---

85. How do you handle multiple `.gitignore` files?

Answer:

- Repo can have multiple `.gitignore` files (per folder).
- Git merges them during evaluation.
- Useful for mono-repos where each service defines its own ignores.

---

86. How do you rebase safely in a team environment?

Answer:

- Rule: Only rebase **feature branches**, never `main`/shared branches.
- Always `git pull --rebase` instead of merge to keep clean history.
- If conflicts → resolve locally before pushing.

---

87. How do you enforce Git branch naming conventions?

Answer:

- Use server-side hooks or CI jobs.
- Example regex rule for branches:
  - `feature/*`
  - `bugfix/*`
  - `release/*`

- In GitLab/GitHub, enforce with **protected branch rules**.

---

88. How do you detect and clean large files accidentally pushed to Git?

Answer:

- Detect:

  ```
  git rev-list --objects --all | sort -k 2 > allfiles.txt
  ```

- Clean:

  ```
  git filter-repo --path filename --invert-paths
  ```

- Then force push updated history.

---

89. How do you roll back to a specific tag in production?

Answer:

- Checkout the tag:

  ```
  git checkout tags/v1.0 -b rollback-v1.0
  ```

- Deploy from rollback branch.
- GitOps: PR the rollback commit into `main`.

---

90. How do you ensure deterministic builds in CI/CD using Git?

Answer:

- Always use commit SHA instead of branch name.
- Example:

  ```
  git checkout <commit_sha>
  ```

- Store commit hash in build artifacts for traceability.

---

91. How do you maintain Git in an enterprise with thousands of developers?

Answer:

- Use **monorepo tooling** (Nx, Bazel) or **polyrepo strategy**.
- Enforce **branch protections, CI checks, PR reviews**.
- Scale repos with **shallow/partial clones**.

---

92. How do you resolve "history has diverged" errors?

Answer:

- Happens when remote and local branches have different histories.
- Fix:

  ```
  git pull --rebase
  # or if forced
  git push --force-with-lease
  ```

- Avoid with rebase-only workflows.

---

93. How do you implement Git-based feature toggles?

Answer:

- Use `feature/*` branches + merge when toggle is ready.
- OR commit feature toggles in code but control with config flags.
- GitOps → environment branch determines which feature is active.

---

94. How do you manage Git workflows for microservices in CI/CD?

Answer:

- Option 1: Separate repos per service (polyrepo).
- Option 2: Monorepo + CI filters (build/deploy only changed services).
- Example in GitHub Actions:

  ```
  on:
    push:
      paths:
        - "service-a/**"
  ```

---

95. How do you handle force pushes in team environments?

Answer:

- Use `git push --force-with-lease` (safer than `--force`).
- Set branch protections in GitHub/GitLab to prevent force pushes on `main`.
- Only allow on feature branches.

---

96. How do you mirror Git branches across multiple repos?

Answer:

- Use `git push` with multiple remotes:

  ```
  git push github main
  git push gitlab main
  ```

- Or automate with CI/CD sync jobs.

---

97. How do you integrate Git tags with release automation?

Answer:

- Create annotated tags:

  ```
  git tag -a v2.0 -m "Release 2.0"
  git push origin v2.0
  ```

- CI/CD triggers on tags to publish Docker images/packages.

---

98. How do you implement GitOps for multiple Kubernetes clusters?

Answer:

- Use **branch-per-cluster** or **folder-per-cluster**.
- Example structure:

  ```
  /clusters/prod/
  /clusters/staging/
  /clusters/dev/
  ```

- ArgoCD/Flux syncs each cluster from its folder/branch.

---

99. How do you enforce “no direct commits to main” in Git?

Answer:

- Enable branch protection rules.
- Use server-side hook:

  ```
  # .git/hooks/pre-receive
  if [ "$branch" = "refs/heads/main" ]; then
    echo "Direct commits to main not allowed."
    exit 1
  fi
  ```

---

100. How do you manage Git history size in a long-lived project?

Answer:

- Archive old branches & tags.

- Run regular GC:

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

- Use shallow clones in CI/CD.
- Consider repo splitting with `git filter-repo`.