**1.git architecture**

Refer It defined notes

**2.How will u modify the git commit message**

git commit --amend -m "Updated commit message"

**3.Difference between git push and git pull**

 git push → Sends your local commits to the remote repository. (Local → Remote)

git push <remote\_name> <branch\_name>

git push origin main

 git pull → Fetches and merges remote changes into your local branch. (Remote → Local)

| **Concept** | **Explanation** |
| --- | --- |
| Push requires commits | You can only push committed changes. |
| Pull merges automatically | git pull = git fetch + git merge. |

**4.difference between git fetch and git pull**

 git fetch → Downloads commits from the remote repository but does not merge them. You update your local view of the remote branch. (Remote → Local, no merge)

git fetch <remote\_name> <branch\_name>

git fetch origin main

 git pull → Downloads and merges the remote changes into your current branch. (Fetch + Merge)

git pull <remote\_name> <branch\_name>

git pull origin main

**5.what is git cherry pick??**

git cherry-pick is used to copy a specific commit from one branch and apply it to another branch.

git checkout main

git cherry-pick <commit hash>

**6.what is git rebase??**

Refer it defined notes

git rebase <base\_branch>

git checkout feature/login

git rebase main

**7.Difference between git clone and git fork??**

* Git Clone → Makes a local copy of a repository. (Local)

git clone <repository\_url>

* Git Fork → Makes a remote copy of a repository in your account to contribute. (Remote)

**🔹 Workflow After Forking**

* + **Fork on GitHub**
* Click the **Fork** button on the repository page.
* This creates a copy in **your GitHub account**.
  + **Clone your fork locally**
* git clone https://github.com/yourusername/repo.git
* **Add the original repository as remote** (optional, to sync later)
* git remote add upstream https://github.com/originaluser/repo.git
  + **Sync changes from original repo** (upstream)
* git fetch upstream
* git checkout main
* git merge upstream/main
  + **Push changes to your fork**
* git push origin main

**8.how will you identify who has done previous commit**

Git log

git show <commit-hash>

git blame <file> shows **who last modified each line** in a file.

**9.what is git revert and git reset??**

Refer it defined notes

git revert <commit\_hash>

git reset [mode] <commit\_hash>

If no mode is specified, default is --mixed.

**10.what is purpose of amend cmd in git**

The purpose of git commit --amend is to modify the most recent commit.

**11.what is stash ?where will be in the stash**

git stash is a Git command used to temporarily save your uncommitted changes so you can work on something else without committing them.

OR

Need to move another branch to avoid false commits we do git stash

**1. Save changes to stash**

**git stash**

**➡ Saves your uncommitted changes (working directory + staging area).**

**2. Save with a message**

**git stash save "message"**

**➡ Helps identify stashes later.**

**3. List all stashes**

**git stash list**

**➡ Shows all saved stashes.**

**4. Apply stash without deleting**

**git stash apply**

**git stash apply stash@{0} # apply specific stash**

**➡ Brings back changes but keeps stash.**

**5. Apply stash and delete**

**git stash pop**

**➡ Brings back changes and removes stash.**

**6. Delete a stash**

**git stash drop stash@{0}**

**➡ Removes a specific stash.**

**7. Clear all stashes**

**git stash clear**

**➡ Removes all saved stashes.**

**12.How to delete a branch?**

git branch -d branch\_name

 Local: git branch -d branch\_name

 Remote: git push origin --delete branch\_name

**13.what is git squash??**

git squash is a Git technique used to combine multiple commits into a single commit.

**14.How will create branch?**

git branch branch\_name

**15.How do u check version of git**

git –version

2.41.0

**16.How do u Resolve merge conflict**

Refer IQ doc

**17.where do u get the commit history**

Git log

**18.what is git fetch?why do we need git fetch**

git fetch is a Git command used to download commits, files, and references from a remote repository to your local repository without merging them into your current branch.

Purpose / Why we need git fetch:

1. Update remote tracking branches
   * Keeps your local repo aware of new commits on the remote.
   * Example: origin/main gets updated with remote changes.
2. Safe way to see changes
   * Unlike git pull, git fetch does not modify your working branch, so you can review changes before merging.
3. Keeps history up to date
   * Helps prevent conflicts when you later merge or rebase.

**19.how can you convert dir into git dir**

Git init

**20.what is git tag??**

Git tag = a label or marker for a specific commit, usually used for releases.

Types of Tags

1. Lightweight tag – simple pointer to a commit:

git tag v1.0

1. Annotated tag – stores additional info like author, date, and message:

git tag -a v1.0 -m "Release version 1.0"

View tags

git tag

Push tags to remote

git push origin v1.0

# or push all tags

git push origin --tags

**21.what is git diff??**

**Refer IQ doc**

**git diff [options] [<commit>] [<commit>]**

**22.what is PR??**

**Refer IQ doc**

**23.what is bitcucket**

Bitbucket is a web-based Git repository hosting service (like GitHub or GitLab) that allows teams to store, manage, and collaborate on code.

Key Points:

1. Version Control: Supports Git and Mercurial repositories.
2. Collaboration: Provides pull requests, code reviews, and comments.
3. Integration: Works with CI/CD tools (like Jenkins, Bamboo) and project management tools (like Jira).
4. Private Repos: Offers free private repositories for small teams.

**24.what are the platforms that you have worked in git**

Common Git Platforms / Hosting Services:

1. GitHub – Popular for open-source and private repositories.
2. GitLab – Provides Git hosting + built-in CI/CD pipelines.
3. Bitbucket – Integrates well with Jira and supports private repos.
4. Azure DevOps Repos – Microsoft’s Git hosting with DevOps tools.
5. AWS CodeCommit – Cloud-based Git repositories on AWS.

**25.Need to know the status of git**

Git status

**26.what message would you see when u do file in staging area**

Changes to be committed:

**27.How to avoid merge conflict**

Keep branches updated, make small commits, communicate, and merge frequently to reduce merge conflicts.

**28.what is git clone??**

git clone is a Git command used to create a local copy of a remote repository.

git clone <repository-url>

**29.branch 1 and branch 2 want to merge from branch 2 to branch 1**

**1.Switch to the branch you want to merge into**

git checkout branch1

**2. Merge the other branch**

git merge branch2

**30.Difference between git and github**

| Feature | Git | GitHub |
| --- | --- | --- |
| Type | Version control system (VCS) | Web-based hosting platform for Git |
| Purpose | Track and manage code changes | Host repositories online and enable collaboration |
| Scope | Local and distributed | Remote / Cloud |
| Internet required | No | Yes |
| Main commands | git commit, git push, git pull, git merge | Use Git commands + web interface features (pull requests, issues) |
| Collaboration | Limited to sharing via remote repos | Facilitates collaboration, code reviews, and project management |
| Extra features | Version tracking, branching, merging | CI/CD, project boards, issues, social coding |

**31.if it is in stage what is the status??**

Changes to be commited

**32.what is Reflog?**

git reflog is a Git command that records all changes to the tips of branches and other references, including commits that are no longer visible in the normal branch history.

**33.How can u get to know all the files stashed**

Git stash list

**34.want to delete particular file in git stash**

In short:  
To remove a file from a stash → apply stash → remove file → re-stash → drop old stash.

This is the standard workflow since Git stash doesn’t support removing individual files directly.

**34.want to delete particular file in git stash**

Git does not provide a direct way to delete a single file from a stash, but you can do it with this workaround:

Steps to remove a specific file from a stash

1. Apply the stash without removing it

git stash apply stash@{0}

1. Remove the unwanted file from changes

git reset HEAD <file> # unstage the file if staged

git checkout -- <file> # discard changes to that file

1. Create a new stash with the remaining changes

git stash

1. Drop the old stash

git stash drop stash@{0}

Apply stash → remove the file → re-stash → drop old stash.

**35.what is git stash pop**

git stash pop

By default, it applies stash@{0} (the latest stash).

You can also apply a specific stash:

bash

Copy code

git stash pop stash@{1}

**36.what is git stash apply**

git stash apply is a Git command used to apply stashed changes to your working directory without removing the stash from the stash list.

**37.what is bare repo and non bare repo**

1️⃣ Non-Bare Repository

* Definition: A standard Git repository that contains:
  1. Working directory → the actual files you can edit.
  2. .git directory → stores Git history, branches, and objects.
* Purpose: Used for development. You can edit files, commit changes, and push.
* Example: Any repo you clone locally for coding.

Structure:

my-repo/

├── .git/

└── <project files>

2️⃣ Bare Repository

* Definition: A repository without a working directory, only contains Git history and metadata.
* Purpose: Used as a central/shared repository (like on GitHub or Bitbucket).
* You cannot directly edit files; you can only push/pull from it.

Structure:

my-repo.git/

├── branches

├── hooks

├── objects

├── refs

└── config

**38.expalin squash commands**

2️⃣ Steps to squash commits (using interactive rebase)

1. Start an interactive rebase

git rebase -i HEAD~n

* Replace n with the number of commits you want to squash (e.g., last 3 commits → HEAD~3).

1. Choose commits to squash

* In the editor that opens, you’ll see something like:

pick abc123 Commit 1

pick def456 Commit 2

pick ghi789 Commit 3

* Change pick to squash (or s) for commits you want to combine:

pick abc123 Commit 1

squash def456 Commit 2

squash ghi789 Commit 3

1. Edit commit message

* Git will open an editor to combine commit messages.
* Keep or edit the message as desired, then save and exit.

1. Finish rebase

* Git rewrites history, combining selected commits into one commit.

**39.how do u delete atag**

git tag -d tag\_name

**40.what cmd stage to local**

git commit -m "Your commit message"

**41.what cmd to workspace to staging**

git add <file>

**42.what cmd to central to workspace**

Git pull

**43.what cmd to central to local**

git fetch origin

**44.what cmd to local to central**

Git push

**45.How to avoid merge conflict**

Merge conflicts happen when two or more people (or branches) change the same part of a file and Git can’t decide which version to keep.

**1. Always pull before you start working**

git pull origin main

➡ Keeps your local branch up to date with the remote repository before making new changes.  
Prevents conflicts caused by working on outdated code.

**2. Work on separate branches**

* Don’t work directly on main or master.
* Create a new branch for each feature or bug fix:

git checkout -b feature/login-page

➡ This isolates your work and minimizes overlapping changes.

**3. Commit and push frequently**

git add .

git commit -m "Update login form"

git push origin feature/login-page

➡ Frequent commits make it easier to identify and resolve small conflicts early.

**4. Communicate with your team**

* Avoid working on the same file or same lines as someone else.
* Assign modules or files to each team member if possible.

**5. Use git fetch and rebase instead of merge (when appropriate)**

git fetch origin

git rebase origin/main

**46.can u give me the cmd to squash multiple commits to a single commit**

Git rebase -i

**47.Difference between Centralised and distributed version control??**

| Feature | Centralized Version Control (CVCS) | Distributed Version Control (DVCS) |
| --- | --- | --- |
| Definition | Has a single central server that stores all versions of code. | Every developer has a full copy (clone) of the repository. |
| Examples | Subversion (SVN), CVS, Perforce | Git, Mercurial, Bitbucket |
| Repository structure | One central repository; developers commit changes directly to it. | Each developer has a local repository + a remote (central) repository. |
| Network dependency | Requires network connection to commit or view history. | Works offline; commits and history are stored locally. |
| Single point of failure | Yes — if the central server goes down, no one can commit or retrieve history. | No — every developer has a full copy of the repository. |
| Speed | Slower (operations depend on server). | Faster (most operations are local). |
| Collaboration model | Centralized — all commits go to one place. | Distributed — developers can work independently and merge later. |
| Security | Code history is stored only on the server. | Code history is replicated on all developer machines (more backup). |
| Branching & merging | Complex and expensive operations. | Easy and efficient branching and merging. |

**48.what is the purpose of git**

The purpose of Git is to track changes in code (or any file) and help multiple people collaborate on the same project efficiently.

🔹 1. Version Control

Git keeps a complete history of changes to your files.  
You can:

* Go back to any previous version
* Compare changes between versions
* Undo mistakes easily

🧠 *Example:* If your code breaks after a change, you can revert to an older working version using Git.

🔹 2. Collaboration

Multiple developers can work on the same project at the same time without overwriting each other’s work.  
Each developer works in their own branch, and changes can later be merged.

🧠 *Example:* You work on a new feature while another developer fixes a bug — both changes can later be combined.

🔹 3. Backup and Synchronization

Your project can be stored both locally and on a remote server (like GitHub, GitLab, Bitbucket).  
This acts as a backup and allows synchronization between multiple machines.

🔹 4. Experimentation (Branching)

Git allows you to create branches to test new features or ideas without affecting the main codebase.

🔹 5. Audit and Accountability

Git records who made what change and when, which helps in code reviews, debugging, and accountability.

**49.why git commit is useful??**

git commit is used to record changes in the local repository with a message, helping track and manage project history effectively.

**50.what is the purpose of branching??**

The purpose of branching in Git is to create separate lines of development so that multiple features, bug fixes, or experiments can be worked on independently without affecting the main codebase. Branching allows developers to work in isolation and later merge their changes back into the main branch when ready.

**Short version (for quick interviews):**

Branching lets you work on new features or fixes separately without disturbing the main code, and merge them later when stable.

**51.suppose we have feature branch how do create new branch from new branch**

**52.Why git us used for??**

Git is used for version control — it helps developers track changes in code, collaborate efficiently, and manage different versions of a project.

Key points:

1. 🗂 Version Control: Keeps a history of all changes made to the code.
2. 🤝 Collaboration: Allows multiple developers to work on the same project simultaneously without overwriting each other's work.
3. 🌿 Branching & Merging: Developers can create branches to work on new features or bug fixes independently and later merge them.
4. 🧩 Backup & Recovery: Provides a way to recover previous versions of code easily.
5. ⚡ Distributed System: Every developer has a full copy of the repository, making it fast and reliable.

**53.Explain the basic steps to push to rempte repository**

🔹 Step-by-step process:

1. Initialize Git (if not already done)
2. git init
3. Add remote repository URL  
   (Link your local repo to the remote one)
4. git remote add origin <remote\_repository\_URL>

Example:

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

1. Check remote connection
2. git remote -v
3. Add files to staging area
4. git add .
5. Commit the changes
6. git commit -m "Your commit message"
7. Push to remote repository
8. git push origin main

*(Replace main with your branch name if different — e.g., master or develop)*

**54.How to create local repository**

Git init

**55.what are the file repositories in git init**

"When we run git init, Git creates a hidden .git folder containing all repository-related files like HEAD, config, refs, objects, hooks, and logs. This directory holds all version control information and makes the folder a Git repository."

| **File/Folder** | **Description** |
| --- | --- |
| **HEAD** | Points to the current branch reference (e.g., refs/heads/main). |
| **config** | Stores repository configuration details like username, remote URL, etc. |
| **description** | Used by GitWeb (optional); describes the repository. |
| **hooks/** | Contains sample scripts that trigger on specific Git actions (like commit, push). |
| **info/** | Contains global exclude patterns for files that should not be tracked. |
| **objects/** | Stores all commits, trees, and blob objects (the actual data). |
| **refs/** | Contains references to branches, tags, and remote repositories. |
| **index** | Tracks the staging area information. |
| **logs/** | Records history of branch updates and operations. |

**💡 Simple Explanation:**

When you run git init, Git creates a .git folder that stores all the internal data — commits, branches, and configurations — making your directory a Git repository.

**Example interview answer:**

"When we run git init, Git creates a hidden .git folder containing all repository-related files like HEAD, config, refs, objects, hooks, and logs. This directory holds all version control information and makes the folder a Git repository."

Would you like me to include a **diagram** showing how .git manages workspace → staging → repository?

Top of Form

Bottom of Form

**56.what is Versioning??**

Versioning means keeping track of changes made to files or code over time — each change is saved as a new version so you can review, compare, or restore previous versions whenever needed.

**57.why do we create a branch??**

Branches in Git are used to develop features, fix bugs, or experiment without affecting the main codebase.

🔹 Key Reasons to Create a Branch:

1. Isolate Work:  
   Work on a new feature or bug fix without disturbing the main or master branch.
2. Parallel Development:  
   Multiple developers can work on different branches simultaneously.
3. Safe Experimentation:  
   You can try new ideas or changes safely; if it fails, the main branch remains unaffected.
4. Organized Workflow:  
   Helps maintain a clean and structured code history (e.g., feature/login, bugfix/payment, hotfix/security).
5. Easy Merging:  
   Once the work is complete, the branch can be merged back into the main branch.

"We create a branch in Git to isolate development work, allowing multiple features or bug fixes to be developed in parallel without affecting the main code. It helps in safe experimentation and organized workflows."

**58.what is git cherry pick how do we do that.**

**59.what is git branching strategy**

**60.if I want to see last 5 commits**

git log -5

**61.while configuring creadentials how we do for username and user email**

**62.Branch A and branch B want to merge both if case merge conflict how you resolve**

**63.file is moved to staging area and how do u move to unstaging area**

Git restore –staged<filename

**64.what is git head?**

HEAD points to the latest commit in the current branch.

**65.when we do merging sometimes commitid created ,sometimes not created ??**

When you merge in Git, a new commit is created only in some cases.

Case 1: A new commit IS created — *Normal (Non–Fast-Forward) Merge*

When you merge two branches that have diverged, Git needs to combine their histories → it creates a new merge commit.

Example:

# You are on main branch

git checkout main

git merge feature

If both main and feature have different commits since they split,  
Git creates a new merge commit, like this:

A---B---C (main)

\

D---E (feature)

After merge:

A---B---C-------F (main)

\ /

D---E---/

Here, F = new merge commit.

Case 2: No new commit created — *Fast-Forward Merge*

If the branch you are merging has no new commits in the current branch (i.e., no divergence), Git just moves the branch pointer forward — no extra commit is needed.

Example:

A---B (main)

\

C---D (feature)

If main hasn’t moved since branching:

git checkout main

git merge feature

Now main just “fast-forwards” to D:

A---B---C---D (main, feature)

No merge commit created— Git simply updated the branch pointer.

**66.to remove ignored files want to cleanup even the ignored files**

Remove untracked files (not ignored)

git clean -f

* -f = force (required)
* Removes files not tracked by Git, but keeps ignored files.

Remove ignored files as well

git clean -fdX

* -d = remove untracked directories too
* -X = remove only ignored files

This removes files listed in .gitignore.

**67.what is git stash apply,after that if we run git stash where it ll be**

* Since git stash apply does not remove the stash, it still exists in the stash list.
* Running git stash again (without options) will create a new stash with your current working changes.

**68.Difference between git stash apply and git pop**

| Feature | git stash apply | git stash pop |
| --- | --- | --- |
| Applies changes | ✅ Yes | ✅ Yes |
| Removes stash from list | ❌ No | ✅ Yes |
| Use case | Apply changes but keep stash as backup | Apply changes and remove stash in one step |
| Command example | git stash apply stash@{0} | git stash pop stash@{0} |

**69.what is git –force??**

git --force (often used as git push --force) is a Git command option that forces an operation to happen even if it would normally be rejected. It’s most commonly used with git push when you want to overwrite remote changes.

Example scenario:

1. You commit locally:
2. git commit -m "Fix bug"
3. You try to push:
4. git push origin main
5. If someone else already pushed new commits to main, Git will reject your push to prevent losing their changes.
6. If you’re sure you want your changes to overwrite the remote branch, you use:

git push --force origin main

**70.Difference between repository and git branch.**

| Feature | Git Repository | Git Branch |
| --- | --- | --- |
| Definition | A storage space that holds your entire project, including all files, commit history, and branches. | A pointer to a specific series of commits within a repository. It represents an independent line of development. |
| Scope | Entire project including all branches, tags, commits, and configuration. | Only a subset of the repository’s commits—a single line of development. |
| Purpose | To store, manage, and track all changes for a project. | To allow parallel development, feature work, or experiments without affecting other branches. |
| Example | You clone a repository my-project from GitHub. | The repository has branches like main, dev, feature-login. You can switch between them. |
| Persistence | Exists on disk locally and/or remotely (e.g., GitHub). | Exists inside a repository, cannot exist independently. |
| Analogy | Repository = a library. | Branch = a book in the library representing one storyline. |

**71.what is remote repository??**

* A remote repository is a Git repository hosted on a remote server or a code hosting platform like GitHub, GitLab, or Bitbucket.
* It acts as a centralized location for collaborating changes from multiple developers, where multiple developers can push and pull changes to keep the repository in sync.

**72.what is local repository??**

* A local repository is a copy of a Git repository that resides on your local machine.
* It contains the complete copy of the repo repository, including all branches, commits, files etc.

**73.how I rename a branch in git?**

git branch -m new-branch-name

**74.where do u use pull request??**

A Pull Request (PR) is used in Git-based platforms like GitHub, GitLab, Bitbucket, etc., to propose changes from one branch to another and request that someone reviews and merges them.

**75.what is fetch head**

In Git, FETCH\_HEAD is a special reference that stores information about what has been fetched from a remote repository during the last git fetch.

Key Points About FETCH\_HEAD:

1. Temporary Reference
   * It’s not a branch; it’s a temporary pointer to the commits that were fetched.
   * It helps Git know what new commits exist on the remote without merging them automatically.
2. Used After git fetch
   * When you run:
   * git fetch origin

Git downloads commits from the remote and stores their information in .git/FETCH\_HEAD.

1. Helps in Merging or Comparing
   * After fetching, you can merge changes manually:
   * git merge FETCH\_HEAD
   * Or compare your local branch with the fetched commits:
   * git diff FETCH\_HEAD
2. Contains Multiple Entries
   * If you fetch multiple branches at once, FETCH\_HEAD stores all of them so Git knows what was updated.

**76.other than branch how do I switch to commit**

In Git, you can switch to a specific commit without using a branch by using the commit hash. This puts your repository in a detached HEAD state.

1. Checkout a specific commit

git checkout <commit-hash>

* Example:

git checkout a1b2c3d4

* This moves HEAD to that commit, but you’re not on any branch.
* Any new commits here won’t belong to a branch unless you create one.

2. Using git switch (newer command)

git switch --detach <commit-hash>

* --detach explicitly puts you in detached HEAD mode.

3. Optional: Create a branch from this commit

If you want to start working from that commit:

git checkout -b new-branch <commit-hash>

# or

git switch -c new-branch <commit-hash>

* This creates a branch starting at that commit.

**77.what is the benefit of using pull request??**

1. Code Review

* PRs allow teammates to review your code before it’s merged.
* Helps catch bugs, style issues, or logic errors early.

2. Collaboration

* Multiple developers can contribute to the same project safely.
* PRs provide a platform for discussions, suggestions, and improvements.

3. Controlled Merging

* Changes are not merged automatically; you control when and what gets merged.
* Helps maintain a stable main branch (like main or develop).

4. History & Documentation

* PRs create a record of why changes were made.
* Useful for audits, tracking features, or understanding project evolution.

5. Integration with CI/CD

* PRs can trigger automated tests or builds.
* Ensures only tested, validated code gets merged.

6. Encourages Best Practices

* Supports workflows like feature branching, Gitflow, or fork-based contributions.
* Promotes clean, modular, and well-documented code.

**78.how do u fetch all the branches**

1. Fetch all branches from the default remote (origin)

git fetch origin

* This fetches all branches from the remote, but does not merge them into your local branches.
* Updates your remote tracking branches like origin/branch-name.

2. Fetch all branches and prune deleted ones

git fetch --all --prune

* --all fetches from all remotes (if you have multiple remotes).
* --prune removes references to remote branches that have been deleted.

**79.what is git clean??**

git clean is a Git command used to remove untracked files or directories from your working directory.  
(Untracked means the files are not staged or committed — they are not part of Git history.)

🧩 Purpose:

It helps to clean up your working directory by deleting unwanted files that are not being tracked by Git.

⚙️ Common Commands:

| Command | Description |
| --- | --- |
| git clean -n | Shows which untracked files would be deleted (dry run). |
| git clean -f | Actually deletes untracked files. |
| git clean -fd | Deletes untracked files and directories. |
| git clean -fx | Deletes all untracked files, including those ignored by .gitignore. |

**80.in fast fwd what ll happen to log history**

In Fast-Forward Merge, the commit history (log history) remains linear — no new merge commit is created.

**81.how do u list local and remote branches?? How do u list both local and remote branches**

To list only local branches:

git branch

To list only remote branches:

git branch -r

To list both local and remote branches together:

git branch -a

| Command | Description |
| --- | --- |
| git branch | List local branches |
| git branch -r | List remote branches |
| git branch -a | List both local & remote branches |
| git branch -vv | Show tracking and last commit info |

**82.how do u list what are the remote urls’s present??**

Git remote -v

**83.PR Approval process?**

**84.what is git diff??**

**85.what is git reset head -n??**

**86.how many types of merge??**

**Fats and non fast fwd**

**87.How do u handle hidden files?**

.gitignore

**88.Add changes to the commit ,10 commits in you log I want to add some text in file,I want to add the latest commit**

**89.what are the files available in git tag??**

**90.How to see the status after the merge??**

Git show –stat

Git show

Git status

**91.what it ll display if we run git show**

Commits,date,author,files,index

**92.how to list merge commits**

Git log –merges --oneline

**93.what is push upstream??**

When you create a new local branch, Git doesn’t automatically know which remote branch it should push to or pull from.

So you use:

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

or simply:

git push -u origin <branch-name>

**94.How to combine last 3 commits by using squash**

git rebase -i HEAD ~3

**95.while doing cherrypick did u get any conflict.**

Yes, I’ve faced conflicts during git cherry-pick.  
It usually happens when the same lines of code have changed differently between the source and target branches.  
I resolve it by checking the conflicted files (git status), editing them to keep the correct version, adding the resolved files (git add), and then continuing with git cherry-pick --continue.  
If needed, I can abort the cherry-pick using git cherry-pick --abort.

**🧩 Example Scenario:**

Suppose:

* You have a commit abc123 in the feature branch that fixes a bug in app.py.
* You want to apply that commit to the main branch.

git checkout main

git cherry-pick abc123

Then you see:

error: could not apply abc123... Fix bug in app.py

hint: after resolving the conflicts, mark the corrected paths

hint: with 'git add <paths>' or 'git rm <paths>'

hint: and then run 'git cherry-pick --continue'

**⚙️ Steps to Resolve Cherry-pick Conflict:**

1. **Check conflicted files:**
2. git status
3. **Open the file with conflicts**, you’ll see conflict markers like:
4. <<<<<<< HEAD
5. print("Version from main branch")
6. =======
7. print("Version from feature branch")
8. >>>>>>> abc123
9. **Manually fix the file** — keep the correct code.
10. **Mark conflict as resolved:**
11. git add app.py
12. **Continue cherry-pick:**
13. git cherry-pick --continue
14. **If you want to cancel cherry-pick:**
15. git cherry-pick --abort

**✅ Final Answer (Interview Ready):**

Yes, I’ve faced conflicts during git cherry-pick.  
It usually happens when the same lines of code have changed differently between the source and target branches.  
I resolve it by checking the conflicted files (git status), editing them to keep the correct version, adding the resolved files (git add), and then continuing with git cherry-pick --continue.  
If needed, I can abort the cherry-pick using git cherry-pick --abort.

**96.when you are using hotfix branches??**

We use hotfix branches when we need to quickly fix a critical issue in production without waiting for the normal development or release cycle.

**97.create a couple of tags try to push it to remote repository??**

Git push –origin remote –tags

git ls-remote --tags origin =to list tags in remote

**98.just check want to edit the commit message with the same commit id??**

No — when you edit a commit message, Git actually creates a new commit with a new commit ID.  
You can change the message, but the commit hash (ID) changes, because the commit content (metadata) changes.

**99.what is best rebase or merge??**

Both are useful — it depends on the situation.  
Use merge for preserving history, and rebase for keeping history clean and linear.

**100.how to delete remote tags??**

git push origin --delete <tagname>

| **Action** | **Command** |
| --- | --- |
| Delete local tag | git tag -d <tagname> |
| Delete remote tag | git push origin --delete <tagname> |
| Verify remote tags | git ls-remote --tags origin |

**101.how to delete feature branch??**

Git checkout branch

git branch -d branch name