**1 \* Git branching strategies**

A Git branching strategy defines how we manage code changes using branches.  
The main ones are:

* Git Flow, used for large projects with main, develop, feature, release, and hotfix branches.
* GitHub Flow, for continuous deployment using just main and short-lived feature branches.
* GitLab Flow, a mix of both, supporting multiple environments like dev, staging, and prod.
* Trunk-Based Development, where developers commit small, frequent changes directly to main.  
  I prefer GitLab Flow, as it’s simple, structured, and fits well with CI/CD pipelines.

**2 \* From which branch u release to prod**

Production releases always come from the main branch — it contains stable, tested code ready for deployment.”

* **Development (Dev) environment:**  
  We usually deploy from the develop branch or feature branches.  
  This is where developers test new features and integrate code.
* **Staging (Pre-prod) environment:**  
  We release from the staging or release branch.  
  This branch is used for UAT (User Acceptance Testing) and final validation before production.
* **Production (Prod) environment:**The main (or master) branch is released to production.  
  It contains stable, tested, and approved code ready for live deployment.

**3 \* How u maintain branches for different features**

We maintain separate branches for each feature to keep development isolated and organized.  
Whenever a new feature starts, we create a branch from the develop branch, named clearly — for example,  
feature/login-page or feature/add-to-cart.

feature/<feature-name>  
For example:

* feature/login-page
* feature/payment-gateway
* feature/profile-update

**5 \* PR Approval process ?**

“A Pull Request (PR) is a way to ask team members to review your code before it is merged.  
You create a PR from your feature branch to the main or develop branch.  
Reviewers check the code, suggest changes if needed, and once approved, the code is merged.  
This helps keep the code clean, reviewed, and error-free.”

**9 \* Which type branching strategy you are using in your project**

In my current project, we are using GitLab Flow.  
We have a main branch for production-ready code, a develop branch for ongoing development, and separate feature branches for each new feature.  
Once a feature is complete and tested, it is merged into develop, then promoted to staging for testing, and finally merged into main for production deployment.  
This strategy allows us to keep the code organized, reduce conflicts, and maintain stable releases.

**18\* About version control tool**

A version control tool is used to track and manage changes to code or files over time.  
It allows multiple developers to collaborate, maintain a history of changes, and revert to previous versions if needed.

Types:

* Centralized VCS (CVCS): like SVN — all files are stored on a central server.
* Distributed VCS (DVCS): like Git — every developer has a full copy of the repository locally.

Git is the most popular DVCS because it supports branching, merging, and distributed collaboration efficiently.

* Dependencies or requirements
* Contribution instructions and contact info

**33\* Git webhooks**

“Git webhooks are automated notifications that trigger actions in external services, like CI/CD pipelines, whenever events like push or PR happen in a repository.”

**Git webhooks** are HTTP callbacks that allow Git repositories to **notify external services** when certain events happen.  
They are commonly used in **CI/CD pipelines** to trigger actions automatically.

**Example use cases:**

* Trigger a **Jenkins pipeline** when code is pushed to a branch.
* Notify a **chat application** (Slack, Teams) when a PR is created or merged.
* Automatically deploy code to a **staging or production server** after a merge.

**How it works:**

1. You configure a webhook URL in the Git repository.
2. When an event occurs (like push or pull request), Git sends a **payload** to the URL.
3. The external service processes the payload and performs actions.

**34\* Where do you fix bug , vl u create a new branch and later how you manage with next release**

1. Where to fix a bug:

Critical or production bugs are usually fixed in a hotfix branch, which is created from the main branch.  
Non-critical bugs in development can be fixed in the develop branch or a feature/bugfix branch.

1. Creating a branch:

I create a separate branch for each bug:

git checkout -b hotfix/<bug-name> main # for production bugs

git checkout -b bugfix/<bug-name> develop # for development bugs

1. Managing with next release:

* The bugfix branch is merged back into main (for production fixes) and develop (so the fix is included in future releases).
* This ensures the bug is fixed immediately in production and also available in the next release**.**

**37\* how many repo you have in github**

“I currently have [number] repositories in my GitHub account.  
These include personal projects, practice repositories, and contributions to collaborative projects.  
I regularly maintain them with proper commits, branches, and README files to keep the projects organized and up-to-date.”

**38\* explain all types of branches in your project**

 Main branch (main or master)

* Contains production-ready, stable code.
* Only tested and approved changes are merged here.

 Develop branch (develop)

* Used for ongoing development.
* All feature branches are merged here first before staging or production.

 Feature branches (feature/<feature-name>)

* Created from develop for new features or enhancements.
* Example: feature/login-page.
* Merged back into develop after completion and code review.

 Release / Staging branch (release/<version>)

* Used for testing and final verification before production deployment.
* Example: release/v1.2.

 Hotfix branch (hotfix/<issue>)

* Created from main to quickly fix critical production bugs.
* Merged back into both main and develop.

**8.How git fork works**

**🔹 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

**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.

**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.

**27.How to avoid merge conflict**

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

**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 |

**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.

**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

**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.

**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.

**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.
6. **55.what are the file repositories in git init**
7. "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. |

**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."

**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.

**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.

**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.

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

A Pull Request is used to review and approve code changes before merging them into the main branch. It allows team members to discuss, review, and suggest improvements, ensuring code quality and avoiding bugs. Pull requests also help maintain collaboration, version control, and prevent unauthorized or untested code from being merged.

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.