Git States and Workflow

Git States:

1. **Untracked**:
   Files not being tracked by Git, typically newly created files or those added to the working directory but not yet staged.
2. **Modified**:
   Files that have been edited but are not yet staged for a commit.
3. **Staged**:
   Files that are marked to be included in the next commit.
4. **Committed**:
   Changes that are saved to the local repository and are now part of the project's version history.

Git Workflow:

1. **Working Directory**:
   The space where files are created, edited, or deleted. Changes here are not tracked unless explicitly added to staging.
2. **Staging Area (Index)**:
   A temporary area where changes are listed to be included in the next commit.
3. **Repository**:
   - **Local Repository**: Stores the committed changes on the local machine.
   - **Remote Repository**: A shared version of the repository stored on a server like GitHub, GitLab, or Bitbucket.

Key Git Commands:

1. **Initialization and File Tracking**:
   - git init: Initializes a new Git repository.
   - git add <file>: Moves files from the working directory to the staging area.
   - git commit -m "<message>": Moves changes from the staging area to the local repository.
2. **Collaboration**:
   - git push: Sends local commits to the remote repository.
   - git pull: Fetches and integrates changes from the remote repository.
   - git status: Shows the current state of files in the working directory and staging area.
3. **Transitions and Debugging**:
   - git diff: Shows changes between the working directory and staging area or between commits.
   - git reset <file>: Removes files from the staging area without deleting changes in the working directory.
   - git checkout <branch>: Switches between branches or restores files to their last committed state.
   - git log: Displays the commit history.

Workflow Enhancements:

1. **Branching**:

- Used to isolate work for features, bug fixes, or experiments.
- Commands:
  - git branch <branch-name>: Creates a new branch.
  - git merge <branch>: Merges changes from a branch into the main branch.

2. **Rebasing**:
   - Integrates changes from one branch into another, maintaining a linear commit history.
   - Command: git rebase <branch-name>.
3. **Tagging**:
   - Marks specific commits, often for release versions.
   - Command: git tag <tag-name>.

Collaboration in Git:

1. **Conflict Handling**:
   - Resolve conflicts when merging or pulling changes from a remote repository.
2. **Excluding Files**:
   - Use .gitignore to exclude specific files or directories from tracking.

This comprehensive overview of **Git states and workflow** explains how to manage files, transitions, and advanced collaborative practices effectively.

eg: git fetch origin
his command fetches changes from the remote `origin`

eg: `git pull origin main`
his command fetches and merges changes from the remote `main` branch into your local `main`.

Git History Fixing Commits (3 marks)

Git provides several ways to fix or modify commit history, depending on the situation:

1. **Amending the Last Commit**:
   - Use `git commit --amend` to edit the message or add new changes to the most recent commit.
   - Useful for small corrections like typos in commit messages or overlooked file changes.
2. **Interactive Rebase**:
   - Use `git rebase -i <commit>` to edit, squash, or reorder commits in a branch.
   - Example: Squashing multiple commits into one or editing a specific commit message.
3. **Resetting Commits**:
   - Use `git reset` to move the branch pointer backward in history.
     - `--soft`: Keeps changes staged.
     - `--mixed`: Keeps changes in the working directory.
     - `--hard`: Discards all changes.
   - Useful to undo commits that should not be in the repository.

git commit --amend -m "Updated commit message"
git rebase -i HEAD~3//This rebase command allows editing the last 3 commits using the interactive editor.
git reset --soft HEAD~1 //Moves the last commit back to the staging area.
git reset --hard HEAD~2 //Deletes the last two commits entirely.

Rolling back Deployments and Zero-Downtime Releases

Rollback helps fix things when a deployment goes wrong, so users aren't affected, and debugging can happen later. Advanced methods like Blue-Green Deployments and Canary Releases make it possible to roll back without downtime but can be tricky if data changes or the system connects to many other systems. To make rollbacks work smoothly, always back up the system before releasing and practice the rollback process regularly to ensure it works.

Rollback by Redeploying the Previous Version

This is the simplest rollback method, where the previous working version is redeployed from scratch using an automated deployment process. It includes reconfiguring the environment to match the earlier version. Even without an automated rollback system, an automated deployment process ensures a smooth and predictable redeployment with lower risk.

Advantages:

- Easy and straightforward.
- Fixed time for redeployment.

Disadvantages:

- Can cause significant downtime during redeployment.
- If errors occur, the process might overwrite the old version with the new one, complicating recovery.

Zero-downtime
also known as hot deployments, allow users to switch seamlessly between application versions without delays. If something goes wrong, rollbacks to the previous version can also happen instantly. The key to this approach is decoupling the release process into independent components, allowing each step to be performed smoothly. Before upgrading the application, shared resources such as databases, services, or static files must be updated to match the new version. For web-based services and static resources, versioning is often handled by specifying the version in the resource's URL. This method ensures a smooth transition with minimal disruptions, but it requires careful planning and execution.

Blue-Green Deployment uses two identical environments, Blue and Green, to manage releases. The current production runs in the green environment, while new versions are deployed in the blue environment. Once tested and verified, traffic is switched to Blue, making it the new production environment. Rollback is simple by redirecting traffic back to Green. Challenges include database migration, often resolved by temporarily setting the application to read-only mode. This approach

requires sufficient resources to run both environments simultaneously and ensures seamless transitions with minimal downtime.

Canary Releasing

Canary releasing deploys a new application version to a small subset of production servers, allowing fast feedback without affecting most users. This reduces the risk of issues during release. Initially, the new version is deployed to servers without routing users to it, where smoke and capacity tests are conducted. Then, a small group of users, often power users, is routed to test the new version. Companies can also route different user groups to different versions for testing.

Key Benefits:

1. **Easy Rollback:** Users can quickly be routed back to the stable version if issues arise.
2. **Feature Usage Insights:** Allows companies to assess user engagement and revenue impact of new features and roll back if necessary.
3. **Capacity Testing:** Gradually increases the load on the application, measuring metrics like CPU, memory, and response time, ensuring it meets capacity requirements.

This approach enables safe releases with minimal risk and avoids the need for a large capacity testing environment.

Create a Repeatable, Reliable Release Process

Automate almost everything.
Use version control to manage build, deploy, test, and release processes.
Ensure deployment includes:
a) Environment provisioning (hardware, software, and services).
b) Installing the correct application version.
c) Configuring the application with required settings.