GIT

Theoretical Part:

> The **theoretical part** covers how the GIT version control system works in general.

▼ 🗃️ Version Control Systems (VCS):

▼ Introduction:

Definition:

Version Control System (VCS) is a software system that records changes to a file or set of files over time, enabling software developers to work together collaboratively and maintain a complete history of their work.

Features VCS provide:

- Collaboration — allows for simultaneous work between multiple developers on a single project and a seamless integration of modifications.
- Version Control — maintains a track of all changes performed, enabling rollbacks and development history.
- Conflict Resolution — recognizes and assists in resolving conflicts resulting from simultaneous editing.
- Backup and Security — Maintains an audit trace, manages access, and functions as a safe backup system.
- Transparency — Gives a clear picture of the modifications made, by whom, and when.

▼ Types of VCSs:

Intro:

There are four main types of VCS:

1. **Manual Version Control (copying files)**
2. **Local Version Control Systems**
3. **Centralized Version Control Systems**
4. **Distributes Version Control Systems**

1. Manual Version Control (copying files)

Common technique — files are frequently copied to new directories, perhaps with timestamps added for organization.

Pros:

- Still some version control.

Cons:

- Hard to manage manually.
- Easy to lose track of version.

2. Local Version Control:

A complete version control system set up locally.

Pros:

- Database — maintains a history of file modifications.
- Revision control — enables to keep track of particular file versions and toll them back when necessary. One such tool is the widely used *revision control system (RCS)*, which records changes as *patch sets* (differences between files).
- Patch sets — there are saved on disk and, by applying appropriate patches one after another, enable the reconstruction of any previous version of a file.

Cons:

- Single point of failure as all project version are stored locally on one machine.
- Has issues with collaboration.

3. Centralized Version Control:

All project files are stored on a remote centralized server. Developer workstations connected to the server to retrieve files for editing are called clients.

Pros:

- Improved collaboration.
- Simple administration.

Cons:

- Single point of failure.

4. Distributes Version Control:

DVCSs distribute the project's history instead of relying on a single server.

Pros:

- Decentralized backup — every clone is a full backup. Recovery from server failures is easy by copying any clone back to the server.

- Offline work — developers can work without the need to connect to the internet with a local copy.
- Multiple remote repositories for collaboration — DVCSs support working with several remote repositories.

Cons:

- Complex in administration.

▼ Git VCS:

Intro:

Git is a **Distributed Version Control System (DVCS)** that monitors modifications made to files in different versions.

Features of GIT DVCS:

- Version Control — tracks changes in files made over time.
- Distributed — there are multiple copies of the same project.
- Branching — branches are important feature of Git that are lightweight and easy to create, manage, delete. These branches help in isolating the work while adding new features of fixing bugs. Developers can work on different features simultaneously without impacting the main codebase.
- Merging — changes from one branch to another branch can be merged into the main branch, using tools of Git.
- Commiting — the changes are made in the code is saved as commits in Git, which as nothing but snapshots of the project at given time. Commits are identified by unique hash. And have metadata that contains information about who, when, and what changed.
- External vendors — GitHub, GitLab, GitBucket are platforms that host Git repositories.

▼ 😺 Git Specifically:

▼ Git Terminology:

1. Git Repository:

A repository (or repo) is the fundamental unit of Git where all project files and their revision history are stored. There are two types of repositories:

- **Local repo:** This is the version of the repository on local machine. It includes the working directory, staging area, and the `.git` directory containing the actual version history.
- **Remote repo:** This is the version of the repository hosted on a server or cloud server like GitHub. Remote repositories allow multiple developers to collaborate on the same project.

2. Working Directory:

The working directory is where the actual work is performed. All the files the Git tracks are stored in a working directory.

- Git keeps an eye on these files and gets them ready for commits.
- In essence, it acts as the working area where modifications are performed prior to being completed and committed to the Git repository.

3. Staging Area (Index):

The staging are (or index) is a space where you prepare a snapshot of the made changed before committing them into the repository. When `git add <file_name>` is used, changes are moved to the staging area. This allows to control which changes are included in the next commit.

4. Commit:

A commit is a snapshot of the project at a specific point in time. it records changes to the repository and includes a commit messages that describe the changes made. Each commit is identified by a unique hash (SHA-1 checksum). The command `git commit` creates a new commit form the changes in the staging area.

5. Branch:

A branch is a separate development patch that departs from the main project.

- it permits working independently on project features or versions.
- Creating, renaming, listing, and deleting branches are important operations.
- Branches allow for efficient version control and collaboration by allowing them to be pulled or pushed remote repositories and then merged into the main project.
- Branches are managed with commands like `git branch`, `git checkout`, and `git merge`.

6. Merge:

Merging is the process of integrating changes from one branch into another. When two branches are merges Git combines the changes from both branches. The command `git merge` is used to perform this operation. Merges can be straightforward or may require resolving conflicts if changes overlap.

7. Rebase:

Rebasing is an alternative to merging that integrates changes from one branch into another by replaying commits. The `git rebase` command applies from one branch onto another, creating a linear history. Rebasing can simplify the commit history but must be used carefully, especially with the shared branches.

8. Commit History:

The commit history is a chronological list of all commits made in a repository. Each commit in the history identified by a hash and includes metadata like author, commit message, date. The `git log` command is used to display the commit history.

9. Tag:

A tag is a reference to a specific commit that is used to mark important points in the repository history, such as releases and milestones. Tags can be lightweight (simple pointers) or annotated (contain additional metadata). The command `git tag` is used to create and manage tags.

10. Remote:

A remote is a version of the repository hosted on a server or cloud service. It allows collaboration with others by providing a centralized location for sharing changes. Common commands for working with remotes include `git remote`, `git fetch`, `git pull`, and `git push`.

11. Push:

To upload changes from a local repository to a repose repository, use the `git push` command.

- Commits from the local branch can be moved to a designated branch to the remote repository by using the `git push` command.
- Usually, the command takes as inputs the name of the branch and the repository.
- Pushing should be done carefully since it has the potential to overwrite changes made in the remote repository. (Push a commits from a new branch, then merge them using the remote repository provider's interface).

12. Pull:

To fetch and integrate all changes from a remote repository into the local repository using Git, use the `git pull` command.

- Updates are retrieved from the remote server and merged into the current working branch using the `git pull` command.
- A pull request is also how members are informed when changes in a feature branch are prepared for review.
- The code is review and merged into the main branch after the developers sends a pull request via a remote service like GitHub.

13. Fetch:

The `git fetch` command retrieves updates from the remote repository without merging them into the local branch. It downloads new commits, branches, and tags but does not alter the working directory. Fetching is useful for retrieving changes before integrating them.

- It can be used with arguments, like `git fetch --all` to fetch updates form all branches of a repository and updates remote-tracking branches.
- Before choosing to merge updates into the local repository, one can use this command to view changes from the remote repository.

14. Conflict:

A conflict occurs when Git cannot automatically merge changes form different branches because they overlap or contradict each other. Conflicts require manual resolution by editing the affected files and then staging the resolved changes. Conflicts are indicated in files with conflict markers and can be resolved using tools or editors.

15. HEAD:

The symbolic reference HEAD in Git refers to the active branch or commit.

- It is the most recent commit in the branch that is currently being checked out.
- By using commands such as `git checkout` to switch branches, HEAD is updated to point to the new branch's tip, which reflects the branch's most recent commit.

16. Checkout:

Git allows to navigate between multiple branches or versions within a repository by using the `git checkout` command.

- To switch to a particular branch the is currently being checked out. `git checkout <branch-name>`.
- This enables to switch between branches, for example, from a stable branch to a beta and back again.

17. Clone:

A repository copy can be made to a local machine by using the `git clone` command.

- The complete repository, including files and history, can be copied to a local computer by running the `git clone <repo>` command, where `<repo>` is the repository address.

18. Fork:

A clone of a repository hosted on a remote server is called a Git Fork. It let's make changes to the repository without having an impact on the original project.

- Usually, developers fork a repository so they could independently check for and fix updates.
- It's possible to suggest to merge adjustments back into the original repository by submitting a pull request after making updates, such as bug fixes.
- This process maintains the integrity of the primary project while facilitating contributions and bug fixes.

19. Status:

The `git status` command in Git gives a summary of the local repository's current state.

- Shows details about untracked files, staged changes, and the staging area.

20. DIFF:

The `git diff` command is a flexible tool used to show differences between files or commits.

- It can outline differences between the index and HEAD, between the working directory and the staging area.

21. Reset:

The `git reset` command is used to undo changes by adjusting the state of the repository.

- The working directory, staging area (index), HEAD can be reset.

There are three types of resets:

1. Sort Reset: Maintains changes in the working directory and index by moving HEAD to a prior commit.
2. Mixed Reset: Maintains changes in the working directory while moving HEAD and updating the index.
3. Hard Reset: Deletes the modifications and resets the HEAD, the index, and the working directory to the prior commit.

22. Stash:

The `git stash` command is used to temporary store uncommitted changes in the working directory.

- This allows developers to switch branches without committing to finished work.
- Applying the stored modifications back to the working directory using the `git stash pop` command allows to pick up where the job was left.
- This is useful for managing work-in-progress without cluttering the commit history.

23. Revert:

The `git revert` command is used to create a fresh commit that reverses the changes from the previous commit.

- It's similar to the undo command that is adds a new commit that undoes the changes made in the specified commit, rather than deleting history.
- This method successfully reverts modifications while preserving the project's history.

25. Submodule:

A submodule is a repository embedded within another repository. It allows to include and manage external repositories as a part of the project. Submodules are manages using commands like `git submodule add`, `git submodule update`, and `git submodule sync`.

26. Blame:

The `git blame` command shows who last modifies each line of a file. it's useful for tracking down changes and understanding the history of specific lines. Blame outputs includes commit hashes, author names, and dates of each line.

27. Cherry-pick:

The `git cherry-pick` command allows to apply particular commits from one branch to another without merging the branch as a whole.

- To choose and apply specific changes from one branch, such as a beta branch to another, such as the main branch use the `git cherry-pick commit-id` command.
- This helps to apply specific modifications or correct errors without affecting other commits made in the branch.

28. Index:

The staging area, also called Git index. It's a bridge between the repository and the working directory.

29. Master:

The "master" or "main" branch in Git as the main branch of the repository.

- The main branch must maintain reliable, production-ready code.

30. Origin:

In Git, "origin" is a shorthand alias for the URL of the remote repository from which the project was cloned.

- It simplifies referring to he remote repository by providing a local alias instead of using the full URL.
- This makes it easier to perform operations like fetching, pushing, and pulling changes from the remote repository.

31. .gitignore:

The `.gitignore` file indicates which files or directories Git should not add to the staging region.

- It does not impact files that Git is currently tracking; it only applies to untracked files.
- This help help to prevent secret files to leave the working directory, and be committed and pushed to the remote repo.

32. Squash:

Squash in Git describes the process of combining several commits in to a single commit.

- Usually, the interactive rebase command is used for this.
- Squashing help to organize similar changes into a single, more manageable branch before combining them into the main branch.
- This procedure involves combining changes from many commits into a single file that is subsequently added to the index.

33. git rm:

The `git rm` command is used to delete files from the working directory, repository, and Git index.

- It ensures that tracked files are erased from the working directory as well as the staging area.

▼ Git workflows:

Intro:

Different team use Git in different ways, but there are a few established patterns are likely to be encountered.

▼ Standard team workflow:

1. You pull the latest main: `git checkout main && git pull`;
2. Create a feature branch: `git checkout -b feature/add-user-auth`;
3. Make changes and commit them.
4. Push your branch: `git push -u origin feature/add-user-auth`;
5. Open a pull request for review.
6. After approval, merge into main (usually through Git hosting platform).

▼ Standard open-source workflow:

1. Fork the repository to your GitHub account.
2. Clone the forked repository: `git clone <url>`;
3. Setup upstream. Setting up the upstream remote connects the local repository to the original repository:

   `git remote add upstream <original-repository>`;
4. Create a feature branch: `git checkout -b feature/xyz`;
5. Make changes.
6. Commit changes.
7. Push to the remote repo: `git push origin feature/xyz`;
8. Create pull request to the original repository.

▼ Standard solo-project workflow:

1. Clone — The first step is cloning a remote repository to the local machine. It creates a local copy of the project's files and history.
2. Branch — After having clone of the repository, create a new branch to work on a specific feature or task. branching is isolating the local changes form the main codebase until it can be merged.
3. Work — ...

4. Commit — As changes are made, periodically commit them to the local repository. Each commit is represented as a snapshot of the project at a particular time.
5. Pull — To incorporate the changes made by other developers, you can pull the latest changes from the remote repository.
6. Merge — Once work is completed and tested, merge changes into the main branch. this integrates your changes with the rest of the project.
7. Push — Push changes to the remote repository.

▼ Git Lifecycle:

Intro:

The Git lifecycle involves several steps that represent the state of project's files and how they move from being a just part of a working directory to becoming a part of version-controlled history.

The primary states of the lifecycle include:

```
|→Working Directory—→
|——→Staging Area (Inedx)—→
|----——→Local Repository (Git Directory—→
|--------——→Remote Repository.
```

1. Working Directory:

The working directory is where all the project's files are kept at the beginning of the Git lifecycle. It's a local setting in which the code is edited.

Purpose:

- This directory server for editing, and adding files.

Untracked vs, Tracked files:

- The working directory contains files that can be staged, edited, and untracked.
- Files are initially untracked unless added to the staging area using the `git add <file_name>` command.

2. Staging Area (index):

The **stating area** servers as a sandbox for grouping, adding, and arranging the files that need to be committed in order to track their versions.

- This directory acts as a bridge between the working directory and the Git directory.
- The process of adding, grouping, and arranging files for a commit is called indexing.
- The snapshot of the files that are included in the next commit and stored in the staging are is often referred as an index.

Purpose:

- Group, add, arrange files for a specific commit.

3. Git Directory:

The **Git** directory stores all version history, metadata, and configuration data internally in the Git directory.

- The Git directory includes: configurations files, indexing (staging area), the complete history of commits, and more.

Purpose:

- Keeping all repository's object database, metadata in one place.

Under the Hood:

The Git directory is the actual `.git` directory in the working directory. With the most important one being the `object` folder.

▼ The `.git` directory:

Intro:

The `.git` directory is the Git directory of the repository. It contains all the information about the repository such as:

Contents of the `.git` directory:

1. The `objecs/` folder: contains blob, tree, commit, and tag objects. Basically, a physical Content-Addressable Storage (CAS) filesystem.
2. The `branches/` folder:
3. The `info/` folder:
4. The `refs/` folder: Refs are pointers to a specific commits, these files are rearranged into tags, heads, among other subdirectories:
    a. `refs/heads`: This subdirectory contains references to the latest commit of each branch.
    b. `refs/remotes`:
    c. `refs/tags`: This subdirectory contains references to specific commits associated with tags,
5. The `hooks/` folder: contains hook scripts.
6. The `HEAD` file: Contains the reference to the HEAD branch.
7. The `config` file: Contains the configuration details about the repo.
8. The `description` file: contains the description of the repository (README.md)

▼ Branches:

Intro:

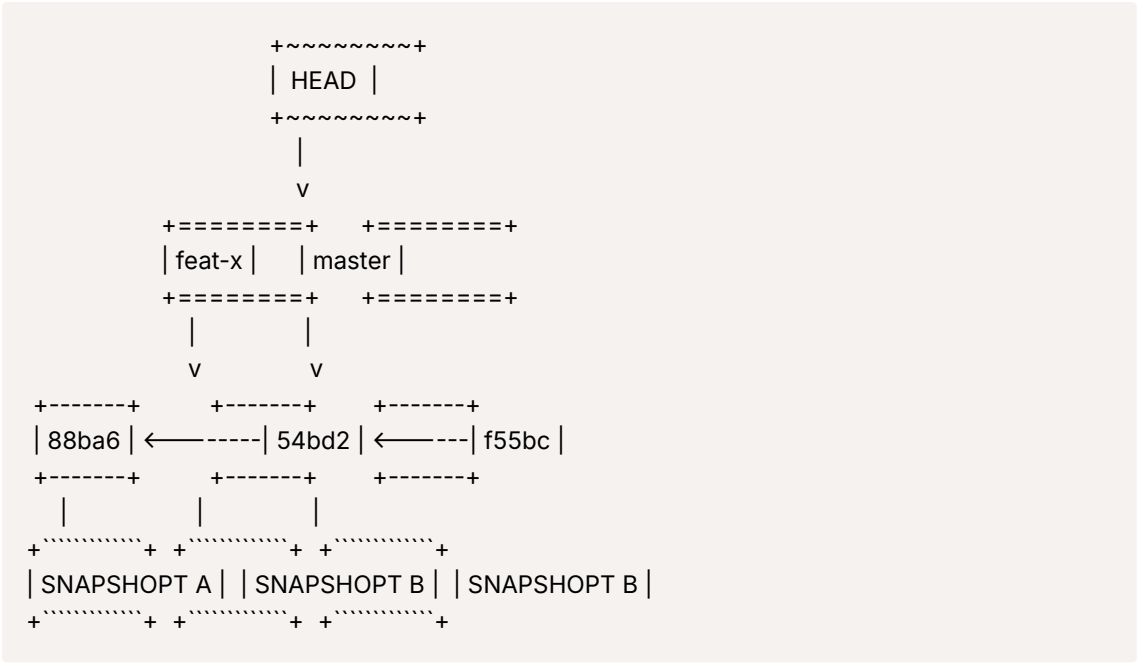
A **branch** is simply a pointer to a specific commit in the Git history, representing an independent line of development within the project.

To switch between branched move the HEAD pointer to a specific branch using the `git switch <branch-name>` (newer), or `git checkout <branch-name>` commands.

How it work under the hood:

In Git, every commit takes a picture of the code as it stands at that particular moment. This snapshot is kept in the form of a commit object, which has details about author, message, and parent commit referenced to it. While regular commits have only one parent, merges (combining branches) have several parents.

Schema:



The “master” branch pointer automatically refreshes to point to the most recent commit.

HEAD

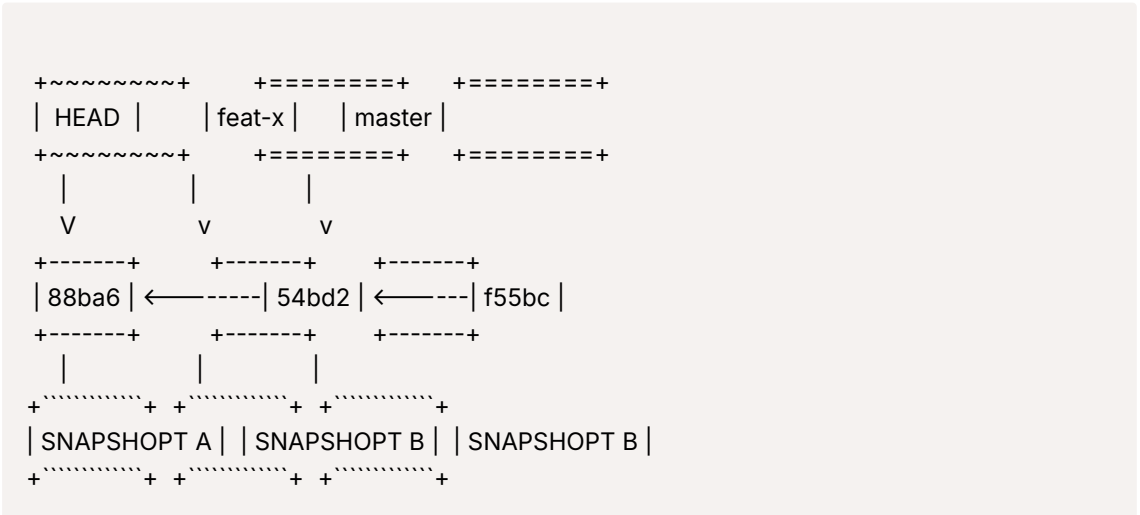
The Git uses a specific pointer called HEAD to keep track of the currently active branch, and HEAD is a reference to the most recent commit on the branch that is being worked on.

Essentially, HEAD keeps track of the commit that is being worked on and adjusts when branch switching or examining various commits.

HEAD is automatically updated to the tip of the new branch whenever the `git checkout` command is executed. As a result, HEAD becomes a dynamic reference that aid to monitor the repository’s current working context.

Detached HEAD

Be default the HEAD pointer must point to a specific branch. But, the HEAD may become detached while examining a remote branch, a tag, or an older commit. Instead of referring to the branch in this case, HEAD points straight to a particular commit. Unless a new branch is made from the detached HEAD, any new commits made in this state will not be connected with any branch and will be lost switching back to a branch.



Branch tracking

By default, local and remote branches are non connected, thus they are stored and managed independently. However, in real life local and remote branches have relationship. By establishing the relationship between a local branch and a remote branch Git knows which remote branch to push and pull from automatically. It simplifies the workflow by letting to use `git push` and `git pull` commands without specifying the remote branch names every time.

▼ Tags:

Intro:

Unlike branches, which can move over time as new commits are added, **tags** are immutable always point to the same commit.

Purpose of tags is to highlight significant events in project's history.

Types of Tags:

- Lightweight tags: Like bookmarks, these are just direct pointers to a particular commit. They do not store extra information, like messages or logger details.

  Lightweight tags are used for temporary or quick tagging tasks.
- Annotated tags: More complex bookmarks. They store extra metadata such as the tagger's name, the date, and a message. Annotated tags are preferred type when tagging releases because they are more descriptive.

▼ Stash:

Intro:

The `git stash` command is used to temporary save uncommitted changes (both staged and unstaged) in the working directory.

Git stashing helps to transition between activities or put off unfinished modifications for the moment without stacking up commit history.

Stashing features:

- Saving work temporary — both staged and unstaged changes can be saved temporary.
- Multiple stashes — multiple set of changes can be saved and reapplied later in any order.
- Stash list — the stashed changed can be applied later using **apply** or **pop** options.
- Stash specific changes — only a selected part of changes can be stashed.
- Older stashed can be retrieved using the reflog notation `stash@{1}`, `stash@{2.hours.ago}`, etc. with the most recent stash kept in `ref/stash`.
- Additionally, stashed can be accessed via index `stash@<n>`, where n is the stash number.

▼ Patch:

Intro:

A Git patch is plain-text file that represents the difference (diff) between two sets of files or commits. It contains code changes, along with metadata about the change (like commit message, author, and date), in format that Git can easily understand and apply to another repository or branch.

Patches are older way to share code, particularly useful in environments where developers don't have a direct push access to a repository or for contribution exchange via email.

Types of patches:

1. With metadata: contains metadata about the changes.
2. Simple patch: simply outlines the differences between two files.

🪛 Git Tools:

Git Command-Line Interface (CLI):

Intro:

Git comes with pre-built command-line interface. Which serves a purpose to interact with the Git's DVCS and remote repositories.

Git GUIs:

Intro:

Git supports several GUI clients that help to visualize work with the Git.

List of Git GUI clients:

1. GitHub Desktop.
2. SourceTree.
3. GitKraken.
4. Fork.

Git Hosting Platforms:

Intro:

Hosting platforms provide a centralized location for Git repositories and often include additional collaboration and project management features.

List of Git hosting platforms:

- GitHub — The most popular platform.
- GitLab — Provides Git repository hosting alongside with CI/CD pipelines, issue tracking and robust DevOps features.
- Bitbucket — Owned by Atlassian, integrates seamlessly with Atlassian's products like Jira.
- Azure Repos — Part of Microsoft's Azure DevOps suite.

Git Integrations & Extensions:

Intro:

Git integrates with various tools and services to enhance its capabilities.

List of Services:

1. Jira — integrates with Jira workflows, issue tracking, etc.
2. Slack — Slack integration can notify teams about Git activities such as commits, pull requests, etc.

3. VSCode — Perform Git operations within the code editor and view diff changes in the codebase.
4. Git Hooks — Git hooks are scripts that run at various points in the Git workflow, such as before committing and after committing. They can enforce code standards, run tests, or automate other tasks.

🪝 Git Hooks:

Client-side hooks:

Intro:

Git hooks are scripts that are executed automatically on an event. There are different types of event including both client-side and server-side.

Note that hooks can be bypassed during development: `git commit --no-verify`;

Set up:

1. Change directory to: `cd .git/hooks`;
2. Create a pre-commit hook script: `touch pre-commit`;
3. Make the script executable: `chmod +x pre-commit`;
4. Write the hook script (example):

```
#!/bin/bash

# This line tells the system to use bash to interpret \
# this script
# It's called a "shebang" and must be the very first
# line

echo "Running pre-commit code style checks..."

# Get a list of all Python files that are staged for \
# commit
# The git diff command shows us what's changed, and we \
# filter for Python files
PYTHON_FILES=$(
    git diff --cached --name-only --diff-filter=ACM |
    grep '\.py$'
 )

# Check if there are any Python files to process
# If not, we can exit successfully without doing any checks
if [ -z "$PYTHON_FILES" ]; then
    echo "No Python files to check."
    exit 0
fi

# Initialize a variable to track if we find any errors
# We'll use this to determine whether to allow the commit
ERRORS_FOUND=0
```

```
echo "Checking files with pycodestyle..."

# Loop through each Python file and run pycodestyle
# We're checking the staged version of each file, not \
# the working directory version
for file in $PYTHON_FILES; do
    # Run pycodestyle on the file
    pycodestyle "$file"

    # $? captures the exit code of the last command
    # If pycodestyle found issues, it returns a non-zero \
    # exit code
    if [ $? -ne 0 ]; then
        ERRORS_FOUND=1
        echo "pycodestyle found issues in: $file"
    fi
done

echo "Checking files with pylint..."

# Now we do the same process with pylint
for file in $PYTHON_FILES; do
    # Pylint is more verbose, so we might want to adjust \
    # its output
    # The basic command just runs pylint on each file
    pylint "$file"

    # Check if pylint found issues
    if [ $? -ne 0 ]; then
        ERRORS_FOUND=1
        echo "pylint found issues in: $file"
    fi
done

# Now we decide whether to allow the commit
# If any errors were found, we exit with status 1, \
# which aborts the commit
if [ $ERRORS_FOUND -eq 1 ]; then
    echo ""
    echo "❌ Commit rejected due to code style issues."
    echo "Please fix the issues above and try again."
    exit 1
fi

# If we made it here, all checks passed
echo ""
echo "✅ All code style checks passed!"
exit 0
```

Type & when it runs & use case:

- Pre-commit: Runs before commit message generation. Linters, code style enforcement. Is a script on this stage exists with a non-zero status, the commit is aborted.
- Prepare-commit-message: Runs after the default commit message is created but before the commit message editor opens. Automatically populate the commit message with some information like a ticket number or inserting a template.
- Commit-message hook: Runs after the commit message edition but before the commit is finalized. Validate the commit message.
- Post-commit hook: Notification or triggering subsequent process.
- Pre-rebase: Runs before operation begins. Can be used to prevent rebasing certain protected branches or warn about potentially problematic rebases.
- Post-checkout: Runs after successful branch checkout or switch to a different commit. Used to set up working directory, maybe cleaning files or adjusting configurations based on the active which.
- Post-merge: Runs after successful merge operation.
- Pre-push: Runs before a push to a remote repository but after the remote refs have been updated locally. This is last chance to catch problems before sharing local changes. Might run full test suite here.

▼ Server-side hooks:

Type & when it runs & use case:

- Pre-receive: Runs on the server when receiving a push, before any references are updated. This is where branch protection is enforced or run security checks on incoming code.
- Update: Similar to pre-receive but, but it runs once for each branch being updated rather than for the entire push.
- Post-receive: Runs after all references have been updated successfully. Perfect place to trigger deployments, send notifications to chat systems, or update issue trackers.

▼ 👨🏻‍💻 Git under the hood:

▼ Content-addressable filesystem:

Intro:

A Content-Addressable Filesystem (CAS), often referred to as Content-Addressable Storage, is a data storage paradigm where the location or address used to retrieve information is derived directly from the content itself, rather then from a conventional file path or physical disk location.

Git is fundametally built upon the principles of a CAS, which provide the core guaranteed of data integrity and efficiency. At its simplest, a CAS acts as a key-value data storage where the key is the contents's cryptographic hash, and the value is the content itself.

Core principles:

1. **Content-Derived Addressing:**
    a. **Mechanism:** When any data (a file, a commit, a directory structure, etc.) is inserted into a CAS, a cryptographic hash functions (SHA-1 or SHA-256) is run over the entire content.
    b. **Result:** The resulting fixed-length hash string is the unique identifies (or "address") used to store and retrieve that piece of content.

c. **Contrast:** In traditional file systems, a file is retrieved by its location (e.g., `/home/user/document.txt`). In a CAS, the object is retrieved by its content's fingerprint (e.g., `830f5...`.)

2. **Immutability and integrity:**
   a. **Immutability:** Once an object is stored in the CAS, its immutable. If a single bit of the content is changed, the hash function will generate a completely different, new hash (due to avalanche effect). This required the system to store the modified data as an entirely new object, preserving the original.
   b. **Integrity:** The hash serves as a built-in checksum. When an object is retrieved, the system can recalculate its hash and compare it to the stored address. If they do not match, the system knows immediately that the data is corrupted of has been tampered with. This provides an absolute guarantee of data integrity.
3. **Automatic Deduplication:**
   a. **Efficiency:** Because the hash is generated solely from the content, two identical pieces of content — regardless of there they originate or what their filename is — will always produce the exact same hash.
   b. **Storage Saving:** A CAS only stores one copy of that content in the underlying storage, and all references simply point to that single object's hash. This leads to extremely efficient storage, as Git does't store fills copies of files that haven't changed between commits; it only stores a pointer to the existing object.

CAS is the Context of Git:

In Git, the `.git/objects` directory is the physical implementation of the CAS. The four object types Git: Blobs, Trees, Commits, and Tags — are all stored within this system:

- **Key:** The 40-character SHA-1 (or SHA-256 in newer Git versions.)
- **Value:** The compressed content of the object (e,g., the contents of a file, the structured list of a directory, or the metadata of a commit).

By using the content's hash as the primary reference, Git establishes a secure, robust, and highly efficient **Directed Asyclic Graph (DAG)** of history, where every point in the timeline is verifiable by its content.

▼ #1. Blob objects (binary large objects):

Intro:

The blob object is the simplest object type and serves as the primary storage unit for file content.

Definition:

A blob object encapsulated the raw data of a file as a specific point in time, independent of its filename, directory structure, or historical content.

Characteristics:

- **Content-Only Storage:** It stores only the exact binary content of a file.
- **Immutability:** Once created, a blob object cannot be modified.
- **Deduplication:** If two files within the repository (or the same file across different commits) have identical content, they will share the exact same blob object and, therefore, the same SHA hash. this is key to Git's efficiency.

▼ #2. Tree object:

Intro:

The tree object defines the directory structure and links filenames to their corresponding blob or other tree object.

Definition:

A tree object is an ordered list of entries, where each entry represents a file (blob) or a subdirectory (tree). It provides the snapshot of a directory's content at a specific moment.

Characteristics:

- **Hierarchical Structure:** A tree object points to blob objects for files and to other tree objects for subdirectories, creating the file system hierarchy.
- **Metadata Storage:** Each entry in a tree object contains:
  - **Mode:** The file permissions (e.g., 100644 for a regular file, 040000 for a directory).
  - **Type:** The type of object (e.g., blob or tree).
  - **SHA-1 Hash:** The unique identifier of the content object it references.
  - **Name:** The filename or directory name.
- **Snapshot integrity:** A tree object's hash is calculated based on the content of it entries ensuring that any change in the directory structure of file content results on a completely new tree object.

▼ #3. Commit objects:

Intro:

The commit object binds the file structure (via the tree object) to the historical context and metadata of a change.

Definition:

A commit object represents a complete, self-contained snapshot the the repository state at a particular time, along with the necessary metadata to connect it to the project's history.

Characteristics:

- **Root Tree Pointer:** It contains a pointer (SHA-1 hash) to the top-level tree object that represents the entire state of the working directory for that commit.
- **Parent Pointer(s):** It includes a list of zero or more parent commits object.
  - Zero parents: This defines the initial commit.
  - One parent: This defines a standard, linear commit.
  - Two or more parents: This defines a merge commit.
- **Metadata:** It stores essential information:
  - **Author:** The individual who wrote the code.
  - **Committer:** The individual who applied the commit (other the same as the author).
  - **Timestamp:** The date and time the commit was created.
  - **Commit Message:** A descriptive text that explaining the changes.

▼ #4. Tag objects:

Intro:

The tag object provides a method for assigning a human-readable, symbolic name to a specific point in the repository history, typically a commit.

Definition:

A tag object is an object that primary serves to reference another Git object (usually a commit) by a memorable name, often including additional descriptive information and cryptographic signing.

Characteristics:

- **Object Reference:** It contains a pointer to the SHA-1 hash of the object it is tagging.
- **Object Type:** It stores the type of the object being referenced (e.g., a commit).
- **Tagging Metadata:** For an annotated tag (the official, recommended type), it includes:
  - **Tagger:** The name of the person who created the tag.
  - **Date:** The date the tag was created.
  - **Tag Message:** A descriptive text for the tag.
- **Distinction from Lightweight Tags:** A lightweight tab is simply a branch pointer that doesn't move, and it does not create a separate tag object; it's merely a file in the `.git/refs/tags/` directory.

▼ Storage & Changes:

Intro:

Git does not store diffs (only made changes) in its fundamental object (blobs and trees).

What Git stores (The Core model):

When a file is changed, Git creates a completely new blob object containing the entire new content of that file. It's a fill snapshot of the file's current state.

How Git saves space (The Packfile model):

While the Core model stores full snapshots, the system eventually optimizes storage by bundling objects into packfiles. Within these packfiles, Git does use delta compression (storing differences) for efficient dist usage, but this happens after the initial object creation and is a storage optimization, not part of the fundamental object definition. The object itself is always logically a full snapshot.

When a new directory is created, a new tree object is created that references the new file's blob and the unchanged subtree/blob objects.

Packfiles in use:

Periodically, Git runs a garbage collection process that packs "loose objects" into pack files. During this process Git looks for similar blobs and stores them using delta compression — it stores one version completely and then stores subsequent versions as just the differences from the base version.

Use garbage collection:

```
# Brefore packing:
du -sh .git/objects
```

```
# may show: 50 MB

# Packing
git gc

# After packing:
du -sh .git/objects
# may show 10 MB
```

Explore Git objects:

```
# See all objects in repo
find .git/objects -type f

# Look at the content of a blob (using its hash)
git cat-file -p af5626b4a114abcb82d63db7c8082c3c4756e51b

# See what type an object is
git cat-file -t af5626b4a114abcb82d63db7c8082c3c4756e51b

# See a tree object (shows directory listing)
git cat-file -p main^{tree}

# See a commit object
git cat-file -p HEAD

# Manually create a blob
echo "test content" " git hash-object -w --stdin
```

Practical Part:

> The **practical part** covers how to work with GIT and GitHub.

▼ 🛠️ Git installation & Configuration:

> Instruction on how to install Git to a local machine and configure it properly.

▼ Locally in Linux:

Step-by-Step using APT:

1. `sudo apt install git-all` — Install GIT from APT.

Step-by-Step Installation from the Source:

1. Check your Linux system has following packages installed: `autotools`, `curl`, `zlib`, `openssl`, `expat`, and `libiconv`.
2. (If necessary run): `sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev gettext libz-dev libssl-dev`. —- install required dependencies.

3. `sudo apt install install-info` — ...
4. Download latest version of Git.
5. Install:
    a. `tar -zxf git-2.8.0.tar.gz` — Unpack the ZIP.
    b. `cd git-2.8.0` — Cd to to the unpacked folder.
    c. `make configure` — use the make util to configure the installation file.
    d. `./configure --prefix=/usr` — Execute the configure util. (Use `--prefix=/usr` if you need to install Git for all users.)
    e. `make all doc info`. — ...
    f. `sudo make install install-doc install-html install-info` . —...

▼ Git Config:

Intro:

Git has a tool named **git config** that allows to set, unset, configuration variables that affect how Git works, which can be managed in three different places:

Scope	Location and Filename	Filename only
System	`etc/gitconfig`	`gitconfig`
Global	`/home/<username>/.gitconfig` or `root/.gitconfig`	`.gitconfig`
Local	`<git-repo>/.git/config`	`config`
Worktree	`<git-repo>/.git/config`	`config.worktree`

- `/etc/gitconfig` file — provides the settings for all system users and repositories. This file is read and written using `--system` option in git config. To change it, administrative privilege is needed.
- `/.gitconfig` or `~/.config/git/config` ` file: contains user-specific values. The `--global` option in git config allows to read and write to this file, affecting all repositories on your system.
- `.git/config` file: is unique to each repository. The `--local` option is used by default by Git to read from and write to this file. You cannot use this option outside of a Git repository.

Common Client Configuration Commands:

- Add a setting (system): `git config --system <key> <value>`.
- Add a settings (global): `git config --global <key> <value>`.
- Add a settings (local): `git config --local <key> <value>`.

Global Settings:

1. `git config --global user.name "Your Name"` — Configure global username for the Git.
2. `git config --global user.email "your-email@mail.com"` — Configure global email for the Git.
3. `git config --global core.editor "code --wait"` — Set VSCode as an editor. (Alternatively, `nano`, `vim`, `nvim`.)
4. Set colors:
    a. `git config --global color.u1 true`.
    b. `git config --global color.status auto`.
    c. `git config --global color.branch auto`.

5. `git config --global init.defaultBranch main` — By default newly created Git repositories come with a branch named “master”. It’s possible to configure default branch name as “main”.
6. `git config --list --show-origin` — List all Git settings.
7. `git config --global --unset <key>` — Unset setting.

Scope Note:

When Git reads configuration variables from multiple files, it may provide unexpected results. to determine which configuration file has final say in setting variable, use following Git command: `git config --show-origin rerere.autoUpdate`.

Scopes are cascading with the most specific file taking precedence over the most general. For example, local scope overrides global scope, and global scope overrides system scope.

Create aliases:

- Create aliases for long git commands:

  `git config --global alias.<alias-name> '<full-command>'`.

  Example:

  `git config --global alias.lg 'log --oneline --graph --all --stat'`.

▼ Other setting:

Set a default pull strategy:

- Fast-forward pull only:

  `git config --global pull.ff only`.

- Explicitly create a new merge commit:

  `git config --global --pull.ff no`.

Set a default merge strategy:

- Fast-forward merge only:

  `git config --global merge.ff only`.

- Explicitly create a new merge commit:

```
git config --global merge.commit no
git config --global merge.ff no
```

Set autostash on pull:

- Auto stash on pull:

```
git config --global pull.rebase true
git config --global rebase.autoStash true
```

▼  Git History:

> Git tracks all changes in a project’s repository.

▼ Theory:

Intro:

Git history is built upon a directed acyclic graph (DAG) formed by commit objects. Where each commits objects might have parents.

Commit objects themselves contain references to the top tree object. The top tree object represents the whole repository files structure at a given time. Leaf nodes of this tree object are links to blob objects, that contain eventual content.

Each commit object has metadata including a timestamp.

▼ Explore Git History:

Intro:

Git Log is the primary powerful CLI tool for exploring and inspecting a repository history. It shows a list of commits, starting from the latest, along side with information about each commit (e.g., hash, author, date the commit was made, and the commit message).

Browsing the Commit History of a File:

The `git log <filename>` command allows to inspect the Git history of a single file.

- Compare commits (displays the diff between commits):

  `git log -p -2 <filename>`.

  - `-p` — Displays the diff.
  - `-2` — Displays the diff only for the last 2 commits.
- View statistics (displays how many lines were added/deleted/modified):

  `git log <filename> --stat`.

Formatting Output:

Built-in Formatting Options:

Option	Description
`git log --oneline`	Displays commits in a one-line format.
`git log --stat`	Shows what files were added/deleted/modified in each commit.
`git log -p`	Outlines differences in each file per commit.
`git log -2`	Limit number of commits shown, from latest.
`git log --graph`	Format log output showing branches.
`git log -s` or `git log pretty=short`.	Short output.
`git log --abbrev-comment`	Show shot commit.
`git log --relative-date`	Shows relative date. Ago from commit time.
`git log --log-size`	Show log size of each commit.
`git log --reverse`	View git history from start to end.

Custom Formatting:

Alternatives for built-in formatting is custom formatting, which is possible with the `git log --pretty=format:"%h - %s (%ar)"` command.

Options table for the `--pretty=format:""` command:

Specifier	Description Output

`%H`	Commit hash
`%h`	Abbreviated Commit Hash
`%T`	Tree Hash
`%t`	Abbreviated Tree Hash
`%P`	Parent Hashes
`%p`	Abbreviated Parent Hashes
`%an`	Author Name
`%ae`	Author Email
`%ad`	Author Date
`%ar`	Author Date Relative
`%cn`	Committer Name
`%ce`	Committer Email
`%cd`	Committer Date
`%cr`	Committer Date Relative
`%s`	Subject (commit message)

Filtering Output:

Filtering Option	Description Output	Example:
`-<n>`	Show the patch introduced with each commit.	`git log -n 2` .
`--since` (= `--after`)	Limit commits made after specific date.	`git log --since="2025-01-01"`
`--until` (= `--before`)	Limit commits made before specific date.	`git log --until="2025-01-01"`
`--author`	Limit commits by author.	`git log --author="Sergei"` .
`--committer`	Limit commits by committer	`git log --committer="Sergei"` .
`--grep`	Filter commit message using Grep	`git log --grep "<feat>*"` .
`-S`	Filter commit changes in files.	`git log -S "import pytest"` .

Note:

- It's possible to use `--grep` multiple times for filtering out.
- Use `--all-match` restricts commits to satisfy the Grep's parameters.

▼ 💾 Commit Management:

> Commits are Git object that contain changes and metadata about the changes.

▼ Commit commands:

Intro:

How does commit work under the hood:

1. **Creates a snapshot of the Staging Area:** The staged changes are stored as a new tree object.

2. **Generates a Commit Object:** The commit object contains a pointer to the new tree object, metadata (author, timestamp), and the parent commit(s).
3. **Updates the Branch Pointer.**

Options:

- Commit modified file, automatically added them in the index:

  `git commit -a`.

- Commit with message:

  `git commit -m "<commit-message-body>"`.

- Alter last commit message:

  `git commit --amend`.

- Alter last commit without message affection:

  `git commit --amend --no-edit`.

  The `git commit --amend` command is used to combine files from the staging area and the last commit, then generate an new unified commit.

  ⚠️Never use the `git commit --amend` command on commits that has already been pushed!

- Set author for the commit:

  `git commit --author="<Author-name>"`.

- Make an empty commit:

  `git commit --allow-empty`.

▼ Commit templates:

Intro:

Git allows to use custom templates for commit messages. Whenever a developer uses command the `git commit` command a text editor opens up with the content inside.

Commit templates allows to insert any sort of additional information (e.g., instructions for developers on how to write commit messages) to the commit message.

Conventionally, the template filename must be `.gitmessage.txt`.

Usage:

- Create template:

  Commit templates are simple `txt` files. Create one, write down additional information for developers that making commits to the project.

- Set template:

  `git commit --template=/path/to/template`.

  The text from the template will be inserted automatically in the commit message.

Set up in the Git Config:

- Set up a commit template globally:

  `git config --global commit.template=~/.gitmessage.txt`.

- Set up a commit template locally:

  `git config --local commit.template=~/.gitmessage.txt`.

▼ Commit message styles:

Intro:

The Conventional Commits specification is a lightweight convention on top of commit messages. It provides an easy set of rules for creating an explicit commit history; which makes it easier to write automated tools on top of.

Standard forms:

1. Short commit messages:

```
<type>(optional scope): <description>
```

2. Expanded (multi-line) commit message:

```
<type>(optional scope): <description>

<optinal body>

<optional footer(s)>
```

Specifications:

1. <type>: There are conventional commit types:
   - “feat” — implemented a feature.
   - “fix” — fixed a bug (or error).
   - “docs” — added doc strings (or other documentation).
   - “style” — styled code after static code analyzer recommendations.
   - “refactor” — refactored code (core behavior and business logic remains unchanged and passed tests prove so.
   - “test” —added unit tests.
   - ...
2. <description>: One sentence commit message.
3. (optional scope):
   - `api`.
   - `models`.
   - ...
4. <optional body: Describe what commit changes in details.
5. <optional footer(s)>: Referenced to KANs and issue ids, so on.

▼ Resetting commits:

Intro:

The `git reset` command in the history-rewriting way to undo changes. it moves the branch pointer (HEAD) to an earlier commit and has different effects on the Staging Area and Working Directory depending on the mode used.

Reset Modes:

Mode:	Action:	Use Case:
`git reset --soft <commit>`	Moves the branch pointer (HEAD) to `<commit>`, but keeps all changes staged (in the index).	To redo the last commit message or add more files before committing again. (Also, can use `git commit --amend`).
`git reset --mixed <commit>` (default)	Moves the branch pointer (HEAD) and unstages all changes, but keeps them in the Working Directory.	To uncommit the last few commits, but keep the changes in the Working Directory to continue working on them.
`git reset --hard <commit>`	Moves the branch pointer (HEAD), unstages changes, and discards all changes in the Working Directory (⚠️WARNING: irreversible).	To destroy all work since a previous commit and completely revert to exact state.

▼ Reverting commits:

Intro:

The `git revert <commit-hash>` command created a new commit that cancels out all changes made after the specified commit.

Stages:

1. Explore Git's history and choose the commit to revert.
2. Start revert:

   `git revert <commit-hash>`.
3. In case of conflict:
   a. Resolve conflicts manually, and continue the revert process:

      `git revert --continue`.
   b. Or abort the revert process:

      `git revert abort`.

Revert on Revert:

▼ Commit squash:

Intro:

Commit squash is a Git technique that allows to unite several commits into a unified one. The squash technique is useful to make the project history more readable and understandable.

Use cases:

- Consolidating feature branches: Before merging a feature branch into a main branch, one can condense its several commits into a single commit. This procedure helps in presenting integrated and understandable set of historical development for the project.
- Getting ready for pull requests: Squash commits makes sure that only the most important changes are included into the pull request before submitting it.

- Cleaning up commit history: Consolidating multiple tiny or incremental commits into fewer, larger commits can help the commit history become more readable and easier to navigate.

How to Squash Commits:

Git offers several methods for squashing multiple commits into one:

1. Squashing commits during a Git merge.
2. Squashing commits with interactive Git rebase.
3. Squashing commits through a pull request.
4. Squashing commits with Git reset.

1. Squashing commits during a Git merge:

1. Checkout to the target branch (the branch the changed are integrated into):

   `git switch <target-branch-name>`.

   Or:

   `git checkout <target-branch-name>`.

2. Merge with squashing the feature branch into the current branch:

   `git merge --squash <feature-branch-name>`.

3. Resolve conflicts manually if there are any.
4. Make a unified commit with a single commit message:

   `git commit -m "<commit message.>"`.

5. Push the commit to the remote repo:

   `git push --force-with-lease`.

2. Squashing commits with interactive Git rebase:

1. Checkout to the branch containing commits to be squashed:

   `git switch <branch-name>`.

   Or:

   `git checkout <branch-name>`.

2. Determine commit count. View the commit history and count the number of commits on the branch:

   `git log --oneline --stat --all --graph`.

3. Start interactive rebase. The interactive rebase should start for an appropriate number of commits:

   `git rebase -i HEAD~<n>`.

   Where n is the number of commits.

4. Select rebase action. In the text editor that opens, observe a list of commits. To squash commits change *pick* to *squash* for the commits you want to combine with the one above them. Save and exit.
5. Edit the commit message.
6. Push changes:

   `git push --force-with-lease`.

3. Squashing commit through a pull request:

Platform like GitHub allow to squash commit during a pull request. It is done using GUI of a website.

4. Squashing commits with Git reset:

1. Perform a soft reset to move the HEAD to the state before the commits to be squashed. This does not alter the staging area and the working directory, keeping the changes intact.:

   `git reset --soft HEAD~<number-of-commits> && git commit`.

   This command move the HEAD back by a certain number of commits, keeping all the changes in the staging area.

▼ Commits stashing:

🔎 Inspect stash:

Stashed changes are stored in the working directory as a list of stash object. Each stash object has a number, name, branch name it came from and a message.

Detailed look:

```
stash@{0}: WIP on ps_13_12: af42548 Merge pull request #17
     ^        ^        ^          ^
     |        |        |          |
 stash number  branch name    hash      Stash message
```

Command:

`git stash list`.

💾 Stash changes:

Save staged and unstaged changes and revert the working directory to the state of the HEAD commit.

Command:

`git stash` or `git stash push`.

Options:

- `-m "<comment-message>"` — add a message for the stash.
- `-u` — include untracked file.
- `-a` — include both untracked and ignored files.
- `-- <file1> <file2>` — specify files for the stash.

☑️ Apply stashed changes:

There are two main ways to apply stashed changed:

1. Apply stash and keep it in the stash list:

   `git stash apply`.

   This command applies the changes to from the most recent commit (`stash@{0}`) to the current working directory, but keeps the stash in the stash list. This is useful to apply the same stashed changes to multiple branches.

2. Apply stash and remove it from the stash list:

   `git stash pop`.

3. Apply stash in a new branch:

   `git stash branch <new-branch-name> [<stash@{n}>]`.

   It creates and checks out a new branch on the commit where the stash was originally created, then applies the stashed changes to that new branch. If the operation is successful, it automatically drops the stash from the stash list.

Options:

- `[<stash@{n}>]` — optionally, specify the stashed changes to apply.

🗑️ Removing stashed changed:

There are two main ways to delete stashed changes from the stash list.

1. Delete a particular stashed changes:

   `git stash drop [<stash@{n}>]`.

   If stash number is not specified the most recent stash is deleted.

2. Delete all stashed changes:

   `git stash clear`.

   ⚠️ Clears the stash list completely.

▼ Commit amend:

Into:

The `git commit --amend` is a powerful Git feature used to modify the most recent commit. It allows to update the commit message, add or remove changes, and fix mistakes without explicitly creating a new commit. This command essentially replaces the last commit with a new one, incorporating any changes made science that commit. While useful, it requires caution because it alters the commit history.

Fixing the commit message:

If the last commit message needs to be altered.

- Alter last commit's message:

  `git commit --amend`.

Adding new files to the last commit:

If some files were forgotten to be included to the last commit.

1. Add the missing file to the Staging Area:

   `git add <filename>`.

2. Alter the commit message:

   `git commit --amend`.

Removing files from the last commit:

If the last commit included files or changes by mistake.

1. Reset the HEAD to a file:

   `git reset HEAD^1 <filename>`.

2. Alter the commit message:

   `git commit --amend`.

Changing the author of the last commit:

- Change the author information:

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

▼ Commit cherry-picking:

Intro:

The `git cherry-pick` command is used to copy specific commit from one branch and apply its changes as a new commit on the currently checked out branch.

Unlike `git merge` and `git rebase` , which integrate all commits from one branch, the `git cherry-pick` lets to choose individual commits (or a range of commits).

Use cases:

- Hotfixes/Backports: A critical but is fixed on the main development branch, but the fix needs to be immediately applied to a release branch without merging all other features.
- Selective feature transfer: A feature branch has a few independent commits, and you only want to merge one specific change into another branch, leaving the rest of the features unfinished.
- Commits on the wrong branch: A developer accidentally commits a small changes to a feature branch instead of the main branch. They can cherry-pick it to the correct branch, then `git reset` the original branch to undo the mistake.

How to cherry-pick:

1. Identify a unique SHA-1 hash of the commit (or commits) to be copied:

   `git log --oneline`.

2. Switch to the target branch:

   `git switch <branch-name>`.

3. Run the cherry-pick command:

   `git cherry-pick <hash>`.

4. In case of:

   - **Success**: Git will apply changes from the specific commit and automatically create a new commit on the currently checked out branch. This new commit will have the same content and message but different SHA-1.
   - **Conflict**: If the changes from the picked commit conflict with the code on the currently checkout out branch, Git will pause the operation, requiring to resolve conflicts manually before running `git add` and then `git cherry-pick --continue`.

▼ 📄 File Management:

▼ Exploring Files Changes:

Intro:

The `git diff` command is used for exploring files changes in the Git history.

Commands:

- Compare the *working directory* with the *staging area*:

  `git diff`.

- Compare *staging area* to *last commit*:
- Compare the *working directory* to the *latest commit*:

▼ Adding files:

Into:

Adding files to the index (indexing), also known as staging is the process of adding files from the working directory to the staging directory.

Interactive adding:

Git allows to add files to the Staging Area in an interactive mode that lets to review and stage changes in the Working Directory line-by-line or in small blocks, called hunks.

Command:

`git add -p` or `git add --patch`.

▼ File Deletion:

Intro:

Deletion of any tracked by Git files requires to use a special Git command. This allows Git to handle the deletion correctly in the repository history. The `git rm` command removes a files from the Staging Area and Git history, but keeps it in the Working Directory.

Command:

- Remove a file:

  `git rm <filename>`.

Options:

- `-f` (= `--force`): overrides default check that verifies files are up-to-date before removal.
- `-n` (= `--dry-run`): simulates the removal procedure without acrually removing any files.
- `-r` (= `--recursive`): removes a directory with all files in it.
- `--`: specify a file (useful when file names can be misinterpreted as cli commands.)
- `--cached`: unstage files from the Stating Area and keep them in the Working Directory.
- `--ignore-unmatch`: the command will terminate with the success status of zero even if no files match the provided pattern.
- `--sparse`: to delete files in large repositories.
- `-q` (= `--quite`): no stdout.

▼ File Renaming/Moving:

Intro:

Reorganization of the directory structure, such as, renaming files, changing their location, or moving files around is done using the `mv` or `--move` command.

Moving files:

- Move file from one directory to another:

  `git mv <source> <destination-directory>`.

Renaming files:

- (✅Best Practice!) Rename a file:

  `git mv <old_filename> <new_filename>`.

  This commands handlers file renaming correctly.

- (❌Worst Practice, but it works) Rename a file:

  1. `mv <old_filename> <new_filename>`
  2. `git add <new_filename>`.
  3. `git commit -m "<feat>: Renamed file"`.

  This works by Git deleting an old file → creating a new file with the content of the old one → the Git's rename algorithm compares the content hashes of the deleted and untracked files, if they match the history of the renamed file will be continuous.

Options:

- `-v` or `--verbose` — verbose output.
- `-f` or `--force` — overwrites files with the same name.
- `-k` or `--keep-existing` — skip any move or rename operation that would cause an error.
- `-n` or `--dry-run` — simulate the command execution.

▼ Restoring files:

Intro:

There are two main approached to restore files and directories in the Working Directory.

1. Using the `git checkout`:

Note: It's unsafe to use this command as it instantly replaces any local changes with the most recent staged or committed version of file.

- **Syntax:**

  `git checkout <branch-name> -- <relative/path/to/file>`.

- **Example:**

  `git checkout main -- settings.py`.

2. Using the `git restore`:

Note: this command can also be used to unmodify modified files.

- **Syntax:**

  `git restore <relative/path/to/file>`.

  Options:

  - `--source <commit_hash>` — restore from a specific commit.
  - `--staged` — unstage files.

- **Examples:**

  - `git restore settings.py`. (restoring a file.)
  - `git restore --source=main -- settings.py`. (restoring a file from a specific branch or commit.)
  - `git restore --staged settings.py`. (unstaging a file.)

- `git restore main_app/`. (restoring directory.)

▼ 🗃️ Repository Management:

> Create, Clone, Pull, and Push.

▼ Create New Repository:

Intro:

The `git init` command is used to create a new repository. When it's executed in a directory, initializes a new, empty Git repository by creating the hidden `.git` sub-directory.

Create a new repo locally:

Create local repository:

1. Navigate to the project's root directory:

   `cd ~/my_project`.

2. Initialize the Git repository locally in the project's root folder:

   `git init`,

3. Index all the project's file to the staging area locally:

   `git add .`.

4. Make the initial commit:

   `git commit -m "Initial commit"`.

Parameters:

- `git init --template=<template-directory>`: create a new repository with the same file structure as the template directory has.
- `git init --separate-git-dir=<git-dir> <working-dir>`: the `.git` sub-directory will be separate from the working directory.
- `git init --object-format=<format>`: Default `sha-1`, optional `sha-256`.
- `git init --ref-format=<format>`: Defines the repository references storage format, default is files, alternative is `reftable`.
- `git init --bare`: Inits a new Git repository without a working directory, which is appropriate for server setups where changes are pulled and pushed but not edited directly.

Optionally, if you want to host the repo on a remote server Do:

1. Create a repository on a hosting platform (e.g., GitHub).
2. Add the remote repository to the local repository:

   `git remote add origin <remote_repository-url>`.

3. Push all local commits to the main branch of the remote repo:

   `git push -u origin main`.

4. Verify the remote repositories:

   `git remote -v`.

▼ Clone a remote Repository:

Intro:

Cloning a Git repository refers to the process of creating a copy of the remote repository on the local machine.

Authentication:

The default Git authentication method is set for a cloned repository whenever the repository is cloned via a URL link or SSH.

- **URL**: uses basic authentication (username, and password).
- **SSH**: asymmetric key authentication.

Note: it's possible to change the authentication method.

Types of cloning:

1. **Standard clone:** Clone the remote repository to the local machine with all files form the 'main' branch, and create the working directory.
2. **Bare clone:** Bare clones are repositories that do not have a working directory. This type of repositories is usually used on remote servers to host remote repositories. Cloning a bare repository does not create a working directory on the local machine. The purpose of these repositories is to act as a central, non-editable repository on a server. Since there's no working directory, no one can directly edit or commit to this copy, which prevents accidental changes.
3. **Mirror clone:** A mirror repository is one step beyond a bare repository which creates a complete replica of the source repository. It copies of reference including all branches, remote-tracking branches and tracks, as well as the repository's configuration.
4. **Shallow clone (depth):** Clone a remote repository only with the last n commits in it.
5. **Cloning with submodules**: Clone a repository with all submodules initializing them receptively.
6. **Branch clone:** Clone a specific branch from a remote repository.
7. **Directory clone:** Clone a specific directory or file from a remote repository.

Commands:

- Standard clone:

  `git clone <repository-url>`;
- Bare clone:

  `git clone --bare <repository-url>`;
- Mirror clone:

  `git clone --mirror <repository-url>`;
- Shallow clone (depth):

  `git clone --depth <n> <repository-url>`;
- Branch clone:

  `git clone -b <branch-name> <repository-url>`;
- Directory/file clone:

  `git clone <repository-url> <path/to/directory>`;
- Submodules clones:

  `git clone --recurse-submodules <repository-url>`;

▼ Pushing commits in a remote repo:

Intro:

Pushing refers to the process of uploading (synchronizing) local Git history to a remote Git host server.

Command:

`git push [<remote>] [<branch>]` .

- `[<remote>]` — name of a remote repository (e.g., “origin”).
- `[<branch>]` — name of a local branch to be pushed.

Options:

- `-f` or `--force` :
  - **Description**: overwrites the remote branch with the local branch, regardless of whether the remote history has diverged.
  - **Use case**: after the local history was rewritten and changes need to be synchronized with a remote repository. Common scenarios include: using `git commit --amend` to fix the last commit; Performing an interactive rebase (`git rebase -i`) to squash, render or delete commits.
  - ⚠️ This flag can easily overwrite work the other people have pushed in the meantime.
- `--force-with-lease` : *(preferred)*
  - **Description:** is an alternative to the `--force` flag. It performs push only if the remote branch hasn’t been updated since the last commit from it. It prevents from accidentally overwriting changes.
  - **Use case:** instead of the `--force` flag to rewrite history on a shared branch. If the push failed, it means someone else pushed new commits, you’ll need to pull and rebase them locally, then try push force with lease again.
- `--all` :
  - Description: Git’s default behavior is to push only the current branch. This flag overrides this, telling Git to push all local branches to the remote repository.
  - Use case: synchronize an entire repository.
  - ⚠️ May push branches that aren’t to be shared.
- `--tags` :
  - **Description:** used to push tags, as tags are not pushed automatically.
  - **Use case:** push local tags to the remote repository.
- `--follow-tags` :
  - **Description:**
    1. Pushes the commits on the specified branch (or current branch if not specified).
    2. Identifies all annotated tags that are ancestors of the newly pushed commits.
    3. Finally, pushes only those tags to the remote repository.
  - **Use case:** Push associated tags along with the new branch.
- `--set-upstream` or `-u` :

- **Description:** Used to link a local branch to a remote branch. Once, the upstream relationship is set, it becomes possible to use simple `git push` and `git pull` commands.
- **Use case:** Facilitate interactions between a local and the remote repositories.

- `--dry-run` :
  - **Description:** Simulate the execution of the push command.
  - **Use cases: E**xamine the commit stdout without applying it.
- `--mirror` :
  - **Description:** Create a full clone of a local repository and push it to the remote. It's the same as `git push --all`, but it pushes:
    - All branches.
    - All tags.
    - All other refs.

    If addition it applies the `--force` and `--prune` flags.

    Where:
    - `--force` will overwrite any existing references on the remote, even if the history diverged.
    - `--prune` will delete any remote branches or tags that are no longer exist in the local repository.
  - **Use cases:**
    - Create a backup of the local repository.
    - Moving repository from one hosting service to another (e.g., from GitHub to GitLab).
    - Setting up a read-only mirror of a repository for high-availability or geographical distribution.
  - ⚠️ This is a destructive command. The standard workflow with this command include:
    1. Create a bare clone of the remote repo:

       `git clone --mirror <remote-repository-url>`.
    2. Apply the mirror command.

       `cd <remote-repo>.git`.

       `git push --mirror <destination-repo-url>`.
- `--atomic` :
  - **Description:** Makes a push atomic, by ensuring that a series of pushed references (like branches and tags) are pushed successfully or non of them at all.
  - **Use cases:**
    - CI/CD
    - Linking branches and tags
    - Complex updates.
- `--no-verify` :
  - **Description:** Used to by-pass the "pre-push" hook.
  - **Use cases:** When the hook is broken, or urgent hitfixes.

- `--prune` :
  - **Description:** Used to delete remote branches that do not have a local counterpart.
  - **Use cases:** Clean up a remote repository. Useful after cleaning up the local repository and synchronize changes in the remote one.
  - **Example:**

    `git push --prune origin 'refs/heads/:refs/heads/'` .
- `-q` or `--quite` :
  - **Description:** Silence the stdout.

▼ Pulling commits from a remote repo:

Intro:

The **pull** operation updates all related remote tracking branches as well as the current working branch. The **pull** operations makes sure that the local repository is up to date and include changes from a remote one.

How it works?

1. **Fetch** — fetches the latest changes from the remote repository.
2. **Merge** — merges the latest changes into the current branch.

In case of a local branch is ahead of the remote one, conflicts are possible. Conflicts must be resolved manually.

Commands:

- General command:

  `git pull [<remote>] [<branch>]` .
- Pull to the current branch:

  `git pull` .
- Pull to a specific branch:

  `git pull origin main` .

Options:

- `--ff-only` :
  - **Description:** Merge is aborted if there are local commits ahead of the remote branch.
  - **Use case:** Git prevents automatically commit creation in case the branch histories diverged.
  - Configure default behavior:

    `git config --global pull.ff only` .
- `--no-ff` :
  - **Description:** Explicitly create a merge commit even when fast-forward is possible.
  - **Use case:** Mark every pull in the history.
- `--rebase` :
  - **Description:** Merge with rebase.
  - **Use case:** Keep history linear.

- `--no-rebase`:
  - **Description:** Instruct Git to perform a standard merge after the pull. (default behavior of the `git pull` command.)
  - **Use case:** Override default configuration.
- `-q` or `--quite`:
  - **Description:** Silence stdout of the command execution.
  - **Use case:** ...
- `--verbose`:
  - **Description:** Verbose stdout of the command execution.
  - **Use case:** ...
- `--autostash`:
  - **Description:** Autostash working directory, pull changes, auto-pop the most recent stash.
  - **Use case:** Simplify pull when the working directory is “dirty”.
- `--no-edit`:
  - **Description:** Prevent opening a text-editor to prompt a comment in case of the merge. The commit message will be autogenerated and accepted automatically.
  - **Use case:** When you do not need to have a distinct commit message in case of a merge.
- `--all`:
  - **Description:** Pulls changes from all remote repositories connected to the local repository.
  - **Use case:** ...
- `--strategy=<strategy>`:
  - **Description:** Specify merge strategy.
  - **Use case:** ...

▼ 🌿 Branch Management:

> Branches are lightweight pointers to a particular commit. These pointers can be compared between each other, created, switched, merge, and rebase.

▼ Branch comparison:

Intro:

Branch comparison refers to the process of comparing content of two branches. Git suggests several tools to compare branches, one these tools if the Git Diff CLI tool, the other one is the Log CLI tool what follows a specific argument convention.

Commands:

- List branches:

  `git branch -va`;

  Flags:

  - `-v` — verbose.
  - `-vv` — very verbose.

    - `-a` — all (including local and remote).
    - `-r` — remote branches only.
- List commits that are not on the main branch in comparison to a feature branch:

  `git log mian..feature-branch`;
- Compare a local branch and a remote branch:

  `git log origin/main..feature-brach`;
- Compare content in branches:

  `git diff <branch1> <branch2>`;

Note: If thee dots are used instead of two, changes are shown.

▼ Branch creation:

Intro:

Branch creation refers to the process of creating a new lightweight pointer to a specific commit.

Creation of new branches is possible only in the local Git repository. Direct branch creation is impossible on a remote repository. Remote repository stores new branches only after they are pushed from a local repo.

Commands:

- Create a new branch from the current (HEAD) branch:

  `git branch <new-branch-name>`;
- Create a new branch from a specific commit:

  `git branch <new-branch-name> <commit-hash>`;
- Create a new branch form the current (HEAD) branch and switch to it:

  `git checkout -b <new-branch-name>`;

▼ Branch switching:

Intro:

Switching a branch is refers to the process of moving the HEAD pointer to another branch pointer.

Restate: the HEAD pointer, is a special type of a pointer that point to a currently active branch. such branch is called the checked out branch.

Git supports multiple ways to switch branches, such as `git branch`, `git checkout`, and `git switch` commands.

Commands:

Older way, using `git branch` or `git checkout`:

- Create a new branch and immediately switch to it:

  `git checkout -b <branch-name>`;
- Create a new branch:

  `git branch <new-branch>`;

Newer approach, using the `git switch` tool:

- Switch branch:

`git switch <branch-name>`;

Options:

- `-c` (= `--create`): creates a new branch and immediately switched to it.
- `-C` (= `--create --force`): creates a new branch and immediately switches to it, even if the branch already exists. It resets the branch and overwrites it.
- `-f` (= `--force`): switches to a new branch and discard uncommitted changes.
- `--detach <commit-hash>`: switches to a particular commit and detaching the HEAD.
- `--guess`: provide a part of the branch name and Git in turn will guess the name and return the result, switching to that branch.
- `--track`: automatically sets up tracking of a new branch from a remote branch.
- `--no-track`: prevent automatic tracking.
- `--orphan`: creates a new branch with no history.
- `--discard-changes`: similar to `-f` (= `--force`).
- `--merge`: avoids losing changes by trying to merge them.
- `--progress` and `no-progress`: toggles progress stdout.

▼ Branch renaming:

Intro:

Local branches can be renamed with the `git branch` command with the `-m` flag. Renaming remote branches is tricky and requires deletion of the remote branch and pushing the one with a new name.

Commands:

- Change branch name locally:
  1. Switch to the branch you want to rename.
  2. Use the `-m` (= `--move`)command or

     `git branch -m <new-branch-name>`;

  Or:

  `git branch -m <old-branch-name> <new-branch-name>`.
- Change branch name on a remote repo: Git does not acrually allow to rename branches on a remote repo, but it can be done by:
  1. (Optionally) Clone the branch being renamed from a remote repo to local:

     `git clone --single-branch --branch <branch-name> <repo-url>`;

     Or

     `git clone -b <branch-name> <repo-url>`;
  2. Delete a remote branch:

     `git push origin --delete <old-branch-name>`;
  3. Rename the branch locally:

     `git branch -m <old-branch-name> <new-branch-name>`;
  4. Push the newly removed branch to the remote repo:

`git push -u origin <new-branch-name>`;

- `-u` — For establishing a tracking connection.

▼ Branch tracking:

Intro:

Branch tracking refers to the process of how Git synchronizes branches in a local repository with the remote repository. By tracking, Git established the relationship between a local branch and a remote one, so Git would know what branch to push and pull from automatically.

Set Up Branch Tracking:

By default, whenever a new repository is cloned, Git sets up tracking a local main branch and the remote main branch. Meanwhile, whenever a new branch is created the Git does not automatically track this branch.

Tracking can be set with this commands:

- When creating a new branch from a remote one:

  `git checkout --track origin/new-feature`;

  (= `git switch --track origin/new-feature`;)

  (= `git branch --tack <branch-name> <origin/branch-name>`;)

- When pushing a new local branch for the first time:

  `git push -u origin new-feature`;

  (= `git push --set-upstream origin new-feature`;)

- For an existing local branch manually set up the upstream branch:

  `git branch --set-upstream-to=origin/existing-branch`.

▼ Branch deletion:

Intro:

Branch deletion refers to the process of deleting branches as some branches unlike the “main” branch are not meant to live forever. The crucial is to understand the difference between local branches and remote branches, deleting the local branch does not delete the tracked remote branch automatically.

Meanwhile, local branches can be deleted using the Git’s branch CLI tool, remote branches can be deleted both locally, then pushed, and online on the Git server hosting website.

Note that, it’s impossible to delete a current HEAD branch.

Commands:

- Inspect local and remote branches:

  `git branch -vva`;

- Delete local branch:

  `git branch -d <branch-name>`.

- Delete a remote branch:

  `git push origin --delete <branch-name>`.

Options:

- `-f` (= `--force`) — used in case branch contains unintegrated commits (unique content).
- `-D` — force deletion.

▼ Branch merging:

Intro:

Branch merging refers to the process of moving commits from one branch into another. There are multiple branch merging strategies Git supports. All these strategies differ how Git operates with commits from one branch.

Basic merge:

1. Switch to the branch that should receive the changes:

   `git switch main`;

   (Merges are integrated to the main branch.

2. Execute the merge command with the name of the branch that has desired changes:

   `git merge feature-zyz`;

   Resolve conflicts if there are any.

Fast-forward merge:

Command:

`git merge feature-xyz --ff-only`;

Explanation:

With fast-forward merge there are two possible scenarios:

- If forward-merge is possible (no local commits): No merge commit created, Git simply moves the the local pointer to the remote branch's latest commit, maintaining a linear history.
- If a fast-forward pull is not possible (Divergent history): It means there are local commits ahead of the remote branch. in this case the `git merge --ff-only` will refuse and abort the merge process.

No fast-forward merge:

Command:

`git merge feature-xyz --no-ff`;

Explanation:

Git explicitly creates a new merge commit even when the fast-forward is possible.

Merge strategies:

There are two types of strategies:

1. Strategy: `--strategy`.

   Examples:

   - `git pull --strategy=ours origin main`.

2. Strategy Options: `--X` (= `--strategy-options`): Applies only to conflicting blocks.

   Examples:

   - `git pull -X theirs origin main`.

Strategies for the `--strategy` option:

Strategy	Description	When to Use
`ort` (Default for one head)	The One-Result-Tree strategy, which is the current default for merging two branches. It is a highly capable three-way merge algorithm.	This is the standard default and is generally the best choice for merging two branches.
`recursive` (Older default for one head)	This is the predecessor to ort and works on two heads using a three-way merge. It can detect and handle merges involving renames.	Still available, but ort is generally preferred.
`octopus` (Default for multiple heads)	Used for merging more than two branch heads. It resolves the cases but refuses to do a complex merge that would require manual resolution.	When bundling similar feature branch heads together.
`ours`	Resolves any number of heads, but the resulting tree of the merge is always that of the current branch's HEAD, effectively ignoring all changes from all other branches.	When you want to combine history but explicitly discard all changes from the branch(es) being merged/pulled.
`subtree`	An extension of the ort strategy to handle subtrees, where one repository is a subdirectory of another.	When merging two projects that are related as a main project and a subproject.
`theirs`		

Some strategies, particularly the default `ort` and `recursive`, accept extra options using the `-X` flag.

Strategies for the `-X` (= `--strategy-options`) option:

- `X ours`: When a conflict occurs, automatically favor the contents from the **local branch** (our version) for the conflicting hunks. This is a *hunk-level* preference, not a whole-tree preference like the `ours` strategy. Changes that don't conflict are still incorporated.
- `X theirs`: When a conflict occurs, automatically favor the contents from the **pulled branch** (their version) for the conflicting hunks. Changes that don't conflict are still incorporated.
- `X patience`: Uses a different diff algorithm that takes more time but is sometimes better at avoiding mis-merges, especially when branches have diverged widely.

Merge with autostash:

Command:

`git merge --autostash`;

- Used to facilitate pull when the working directory is “dirty”.

How it works?

1. Stashes changes.
2. Makes a pull:
3. Pops the most recent stash.

⚠️ **Important consideration:** If new changes conflict with the new commits, Git will pause the operation, report the conflict, and leave the stash entry intact. Conflicts must be resolved manually.

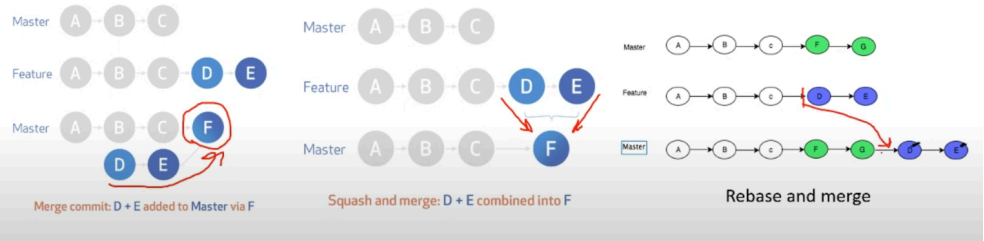
▼ Branch merging strategies:

Intro:

There are three main branch merging strategies:

1. **Merge**: Creates a new merge commit object that has two or mote parents.
2. **Squash**: Creates a new commit that unites all commits from merged branch.
3. **Rebase**: Does not create new commits preserving a linear Git history.

Schema:



General branch merging strategies schema

▼ Branch rebasement:

Intro:

Rebasement is a branch integration technique allowing to integrate commits from one branch into another, and it serves as an alternative to the merging technique.

Commands:

1. Switch to a feature branch:

   `git switch feature-xyz` ;

2. Run the rebase command with the branch name changes are rabased to:

   `git rebase main` ;

▼ Branch pointer operations:

Intro:

Branch pointer operation refer to a set of operation that can be conducted on branch pointers. There are many ways to affect branch pointer in Git.

General command is `git reset <mode> <commit-hash-or-reference>` ;

Types of resets:

- **Soft** (command: `git reset --soft <commit-hash>`):

  Effect:

  - **HEAD:** Moves to a specific commit.
  - **Staging area:** All changes from the discarded commit are staged (`git status` will show them as “Changes to be committed”).
  - **Working directory:** All files remain unchanged.

- **Mixed** (default) (command: `git reset --mixed <commit-hash>`):

  Effect:

  - **HEAD:** Moves to a specific commit.
  - **Staging area:** Cleared. All changes from the discarded commit are unstaged.
  - **Working directory:** All files remain unchanged.

- **Hard** (command: `git reset --hard <commit-hash>`):

Effect:

- **HEAD**: Moves to the specific commit.
- **Staging area**: Cleared.
- **Working directory**: All files are reset to the state of the target commit. Any uncommitted changes are permanently deleted.

🏷️ Tag management:

> Tags are simple pointers to particular commits, and contain additional metadata. Used to mark significant event in the project's history.

Tag operations:

Commands:

- Create a tag (If commit hash if not specified Git use the latest commit in the current branch):

  `git tag <tag-name>`;

  Or:

  `git tag <tag-name> <commit-hash>`;

  Example:

  `git tag v1.0`;

- Create an annotated tag (If commit hash if not specified Git use the latest commit in the current branch):

  `git tag -a <tag-name> -m "<tag-message>"`;

  Example:

  `git tag -a v1.0 -m "Version 1.0 release"`;

- Show details about an annotated tag:

  `git show <tag-name>`;

- Delete a tag from a local repository:

  `git tag -d <tag-name>`;

- Delete a tag from a remote repository:

  `git push origin --delete <tag-name>`;

Checking out Tags:

The `git checkout` operation can be performed in order to view the file versions linked to that tag. This operation, however, puts the repository in a detached HEAD state, which have certain unfavorable effects. Specifically, any changes made and committed in this state will not be a part of any branch and can only be accessed by their unique commit hashes, which will not modify the tag. It's usually advised to create a branch, if any modification need to be made.

- Checkout a tag:

  `git checkout <tag-name>`.

- Checkout a tag in a new branch:

  `git checkout -b <new-branch-name> <tag-name>`.

Pushing Tags to a Remote Repo:

By default, tags are not automatically pushed to the remote repo when the `git push` command is used. Tags need to be explicitly included in the push command.

- Push a specific tag to a remote repository:

  `git push origin <tag-name>`.

- Push all tags to a remote repository:

  `git push origin --tags`.

▼ Best practices with tags:

1. Use annotated tags for releases.
2. Follow a consistent naming convention.
3. Push tags in the remote repository.
4. Tag major and minor changes: For example `v1.0` is a major change and `v.1.0.1` in a minor change.

▼ 🔑 Keys Managements:

> Git allows to use asymmetric keys for authentication to a remote repository via SSH.

▼ Key-Generation:

Intro:

GitHub supports asymmetric key authentication via SSH which allows to access remote repository securely using a a pair of keys.

Note: It's important to use a passphrase for keys, but it's unnecessary if the key will be used by CI.

Commands:

- Generate a pair of Private and Public key for SSH:
  `ssh-keygen -t ed25519 -C "your_email@example.com" -f ~/.ssh/your_keyname`;

  Alternatively:

  `ssh-keygen -t rsa -b 4096 -C "your_email@example.com" -f ~/.ssh/your_keyname`;

▼ SSH agent and keys management:

Intro:

Working on a remote server (e.g., VPS) might require to set up the ssh-agent manually as there may be no GNOME.

Commands:

1. Run the ssh-agent:

   `eval "$(ssh-agent -t 40)"`.

   Flags:

   - `-t <time_in_seconds>` — set the time period for the key to be active.

2. Add Key:

   `ssh-add ~/.ssh/github`. (May require the password).

▼ </> Git CLI:

▼ Necessary Commands Overview:

- `git status` — Check the state of your working directory and staging area.
- `git add` — Add changes to the staged area.
- `git commit` — Commit staged changes.
- `git push` — Send commit to the remote repository.
- `git log` — Log tool. Allows to overlook the version control tree.
- `git diff` — Diff tool. Outlines differences between commits.
- `git pull` — Fetch changes from a remote repository and merge them into your local branch.
- `git branch` — Branch tool. Allows to work with branches locally.
- `git checkout` — Switch to another branch.
- `git merge` — Merge tool. Allows to merge branches.
- `git rebase` — Reapply commits on top of another base branch.

▼ 📝 Patch management:

> Patches are plain-text files containing differences between two files or commits. Using patches involves to main steps: creating the patch and applying the patch.

▼ Creating a patch:

Intro:

The primary command to create a patch that includes that full commit metadata is `git format-patch`.

Commands:

- Create a patch file for a single last commit:

  `git format-patch -1 HEAD`.
- Create a patch file for a specific commit:

  `git format-patch -1 <commit-hash>`,
- Create patches for N last commits:

  `git format-patch -N`.
- Create a patch file for every commit on the current branch that doesn't exist on the "main" branch:

  `git format-patch main`.
- Create a patch file containing changes in the staging area. This is a simple diff that does not include commit metadata:

  `git format-patch --cached > my_changes.patch`.
- Create a patch file containing all changes in the working directory (stages and unstaged). This is a simple diff that doesn't include commit metadata:

  `git diff > my working_changes.patch`.

▼ Applying a patch:

Intro:

There are two main ways for applying a patch, depending on the patch format and desired outcome: `git apply` and `git am`.

Using `git apply`:

Applies changes from a patch file intro the working directory and staging area, without creating a new commit.

- Apply the changes to the working directory. Files are modifies but not staged for commit:

  `git apply <patch-file.patch>`.
- Check if the patch can be applied cleanly without actually applying it:

  `git apply --check <patch-file.patch>`.
- Apply the patch to both working directory and the staging area, ready to be committed manually:

  `git apply --index <patch-file.patch>`.

Using `git am`:

The `git am` (Apply Mailbox) command is used for patches created with the `git format-patch` command, as these patches contain metadata. This command applies the patch and automatically creates a new commit with the stored metadata.

- Apply the patch and create a new commit on the current branch:

  `git am <patch-file.patch>`.
- Stop the `am` process and reverts the branch to its state before the command started:

  `git am --abort`.
- Continue the `am` process after resolving conflicts manually:

  `git am --continue`.

▼ 🅰️ Attributes:

▼ Theory:

Intro:

The fundamental idea is to create a files called `.gitattributes` in repository root directory (similar to how `.gitignore` works), and specify patterns that match files along with attributes that should apply to those files. Git then uses these attributes to modify its behavior when working with those files.

Use case:

Git is fundamentally designed around text files — source code, configuration files, documentation. It treats every file is either text (line-based content that can be diffed and merged) or binary (opaque blobs that Git shouldn't try to understand).

Meanwhile, in real world development, there are types of text files that shouldn't be diffed or merges (images, text-based databases such as sqlite, Word documents, etc.)

Attributes are used to tell Git how to treat those files, for example `.attributes` can contain instruction to completely ignore sqlite database file.

The `.gitattributes` format:

Each line follows the pattern: specify a file (using the same glob syntax as `.gitignore`), then list one or more attributes. Attributes can be set (just the name), unset (with a minus sign), to a specific value (with an equals sign).

```
# Patterns Attributes
*.txt text
*.jpg binary
*.pdf --diff
docs/*.md merge=ours
```

When Git encounters a file, it looks through `.gitattributes` to see if any specific patterns match that file's path. If multiple patterns match, the last matching pattern wins (just line CSS specificity). The attributes from matching patterns then influence how Git handlers that file.

▼ Examples:

Built-in attributes:

1. **Attribute** `text`:

- **Description:** Different platforms use different line endings. Windows `\r\n`, while Unix-like systems `\n`. When developers on different platforms work on the same files, chaos can ensue.

  The `text` attribute tells Git "this is a text file, please normalize its line endings".

- **Example:**

```
# Ensure all source files are stored with LF
# but checked out appropriately
*.py      text
*.java    text
*.cpp     text
*.h       text

# Configuration files should always use LF, even on Windows
*.sh      text eol=lf
Makefile  text eol=lf

# Windows-specific files should always use CRLF
*.bat     text eol=crlf
*.ps1     text eol=crlf
```

2. **Attribute** `binary`:

   - **Description:** The `binary` attribute is shorthand for `-text --diff`, meaning "don't try to normalize line endings and don't generate diffs".

     Without this setting, Git would try to apply line ending conventions to binary files (corrupting them) or attempt to generate diffs that are just noise.

   - **Example:**

```
# Mark binary files explicitly
*.png     binary
*.jpg     binary
*.pdf     binary
*.exe     binary
*.so      binary
*.dylib   binary
```

3. **Attribute** `diff`:
   - **Description:** The `-diff` disables Git to show differences for a file. Or specify a custom diff driver that knows how to generate meaningful diffs for special file formats.
   - **Example:**

     ```
     # In .gitattributes
     *.docx    diff=word

     # In .git/config or ~/.gitconfig
     [diff "word"]
       textconv = docx2txt
     ```

4. **Attribute** `merge`:
   - **Description:** The `merge` attribute allows to customize how Git merges changes to specific files. For example, this is used for files that shouldn't be merged automatically.

     Also, its possible to specify either merge strategy or a special merge driver that understand the structure of specific file types.
   - **Example:**

     ```
     CHANGELOG.md merge=ours
     ```

5. **Attribute** `filter`:
   - **Description:** The `filter` attribute transforms file content as i flows between the working directory and the repository.

     The filtering process is done through clean and smudge filters.
     - clean — filters process files when adding them to the index (making them clean for storage).
     - smudge — filters process files when checking them out (smudging them for use).
   - **Example:**

     ```
     # In .gitattributes
     config/database.yml    filter=credentials

     # In .git/config
     [filter "credentials"]
       clean = sed 's/password=.*/password=REDACTED/'
       smudge = sed 's/password=REDACTED/password=\
         '$DB_PASSWORD'/'
     ```

6. **Attribute** `ident`:
   - **Description:** The `ident` attribute enables keyword expansion similar to older VCS like CVS or Subversion. When enabled, Git expands the string `$Id$` in files to include the blob's SHA-1 hash.
   - **Example:**

```
# In .gitattributes
*.c    ident
```

▼ Best practices:

1. Ensure `.gitattributes` is in the repository root directory, thus everyone in the team gets the same behavior. Unlike `.git/config` settings which are local, `.gitattributes` is meant to be shared.
2. Be explicit rather than relying on auto-detection for critical files. If a shell script absolutely must have LF endings, specify `text eol=lf` rather then hoping auto-detection gets it right.
3. Test attributes before committing them. Use `git check-attr` to see which attributes apply to a file.
4. Complex Git attributes can flow down Git operations significantly if there are complex, and bugs in filter scripts can corrupt the repository.
5. Remember that attributes only affect operations, not stored content (except for filters). If an attribute is changes, a renormalization to the repository might be needed.

```
git add --reconra,ize .
git commit -m "Normalize line endings"
```

GitHub Part:

> The **GitHub** part covers how to work with GitHub.

▼ ⚙️ Repository Settings:

▼ 🔀 Actions (CI/CD):

▼ 💼 GitHub projects:

▼ Managing a project:

Intro:

Managing a Git-based project involves managing the repository to ensure a clean, stable codebase while facilitating collaboration among contributors. Project maintainers are responsible for reviewing code changes, guiding contributions, and ensuring project's overall health.

Contribution guidelines:

- Set a clear rules of how contributors should interact with the project, including coding standards, commit message formats, and pull request processes.
- Create a "CONTRIBUTING.md" file in the repository that outlines these guidelines, including the process for reporting issues and submitting pull requests.

Branching strategies:

- Employ a branching strategy to manage the development workflow. Some popular approaches include:
  - Git Flow: Use separate branches for features, releases, and hot-fixes. The main branch represents production-ready code, while the development branch serves as a working branch for upcoming releases.

    - GitHub Flow: A simpler approach were all new features and bug fixes are developed in branches divided from the main branch and merged back in the main after review.
- Establishing a naming conventions for branches (e.g., “feature/”, “`bugfix/”, “hotfix/”, etc.) to keep the workflow organized.

Review changes and pull requests:

- Review pull requests and pay attention to:
    - Code quality, readability, and adherence to coding standards.
    - Potential bugs, security issue, or logic errors.
    - Proper test coverage.
- Use code review tools (like GitHub built-in features) to leave comments, suggest changes, and approve or request changes.
- When merging pull requests, use merge strategies that fit project’s needs:
    - Squash and merge: Combines all commits into a single commit on the main branch, which keeps the commit history clean.
    - Rebase and merge: Reapplies the commits the the PR to the main branch, preserving linear commit history.
    - Merge commit: Creates a merge commit

Issue handling and bug reports:

Automate testing and continuous integration (CI):

- Set up CI pipeline.
- Integrate linting or static code analysis, into CI pipeline.

▼ Contributing to a Git project:

Intro:

Contributing to a Git-based project involves working collaboratively on a codebase that is typically hosted on platforms like GitHub. it enables multiple developers to work on a project simultaneously, manage changes, and streamline the process of incorporating new features or fixing bugs.

Workflow:

1. Fork the repository.
    - Forking creates a personal copy of the repository, under your account. This allows to make changes freely without affecting the original project.
    - This is typically performed by clicking the “Fork” button on the project’s page.
2. Clone the forked repository:
    - Once you forked the repository you need to clone it to the local machine.
    - `git clone <forked-repo-url>`.
    - This downloads a copy of the codebase to the local machine.
3. Set up the upstream remote:
    - Setting up the upstream remote connects the local repository to the original repository.
    - `git remote add upstream <original-repository-url>`.

- This allows to keep the forked updated with the latest changes from the main project.

4. Create a new branch:
   - It's best practice to create a new branch for each feature or bug fix, rather than making changes directly on the main branch.
   - `git checkout -b <feat-xyz>`.
   - Using names like "bugfix-issue123" or "feature-new-ui-component" makes it easier to understand the purpose of the branch.
5. Make changes and commit:
   - Edit the code and commit changes.
   - `git add .`;
   - `git commit`;
6. Push the changes to the forked repo:
   - After committing changes push them to the forked repository.
   - `git push origin feature-xyz`;
7. Create a pull request (PR):
   - Once the change has been pushed to the forked repository, create a pull request to the original project.
   - Navigate the main repository, add you will typically see an option to create a pull request from the forked branch.
   - In the pull request description, explain the changes were made, and any relevant issue number for the context.
8. Respond feedback:
   - The project maintainers might review the pull request and provide feedback.
   - Make any requested changes, commit them to the same branch and push again. The pull request will be updated automatically.
9. Sync with upstream changes:
   - To avoid merge conflicts, keep the development branch up-to-date with the latest changes from the original repository.

Best Practices:

- Read the contribution guidelines: Most projects have that outline coding standards, commit message formats, and branch naming conventions.
- Write c clear commit message: Use descriptive and concise commit messages to make it easy for other to understand the purpose of the changes.
- Test the changes: Always test the changes thoroughly before submitting a pull request to avoid introducing bugs.
- Stay organized: When working on multiple contributions, use different branches for each change to keep work manageable.

F.A.Q Part:

> The **F.A.Q** part covers common questions and how to tackle common issues.

🚨 Common Issues:

Restore file changes:

Issue Description:

Let's say you are working on the `bug_fix` branch and you have made changes in a particular file. Now, you need to restore these changes. To restore the caches you can pull the file from the branch the `bug_fix` branch was forked.

Solution:

- `git checkout <your_branch_name> -- <path/to/file>`.

Explanation:

- `git checkout <your_branch_name>` — The branch where the needed file is located.
- `--` — Two minus signs allow to specify a particular file.
- `<path/to/file>` — Relative path to the file.

Example:

- `git checkout main -- personal_site/personal_site/settings.py`.

Stop Tracking a File in Git without Deletion:

Issue description:

There is a file you forgot to add to the `.gitignore` file and accidentally started tracking its changes in the Git history. Adding this file to the `.gitignore` won't resolve the issue, as far as, the `.gitignore`'s settings apply only to "untracked files".

Solution:

1. Make sure or add the file to be ignored to the `.gitignore` file:

   `echo my_cache.pyc >> .gitignore`.

2. Remove the file from git (but keeping it locally in the working directory):

   `git rm --cached path/to/my_cache.pyc`.

3. Commit changes to the Git repository:

   `git commit -m "Ignore the 'cache.pyc' file."`.

Additional Info:

- Check ignored files and their statuses:

  `git status --ignored`.

- Add files to be ignored to the global settings:

  `git config --global core.excludesfile '~/.gitignore_global'`.

Broken Staging Area with `git read-tree`:

Issue Description:

You have accidentally entered the `git read-tree` command at the root of the repo and after you have encountered an issue with the staging are execution `git status`.

The `git read-tee` is a low-level command that tells Git how to read the index tree, and when it is executed without any argument the Git starts to read the index as Empty.

Issue Details:

1. `git read-tree`: The command you ran, `git read-tree`, is a low-level "plumbing" command. When used without any arguments, it reads the contents of the empty tree and populates the index (staging area) with it. This is not a common command for everyday use.
2. `git read-tree .gitignore`: You also ran this command, which is a bit of a typo. Git likely interpreted `.gitignore` as a tree-ish (a reference to a tree object). Since `.gitignore` is a file and not a tree, this command would have likely failed or had an unexpected result.
3. `git status`: After these commands, your working directory still contains all your files, but your index (staging area) has been completely emptied. When you run `git status`, Git compares your working directory to the now-empty index. It sees that every single file that exists in your working directory is a new file that isn't in the index.
4. Deleted and Added: Git's status output is a comparison. It shows files that have been "added" from the perspective of the index being empty. This is an unusual state, which is why it's so confusing. The output you're seeing is the result of a comparison between a full working directory and an empty index, where the entire working directory is seen as a new, uncommitted change.

Solution:

- Reset index to match the last commit:

  `git reset HEAD`.

▼ Rollback Project's State to a certain commit:

Issue Description:

You want to rollback the project's current state rewriting the working directory's state and staging area's state.

Make rollback:

Note: It's always better to make this kind of changes in a dedicated branch.

1. Inspect the Git commits history and choose the commit to rollback to:

   `git log`.

2. Move the HEAD pointer to the chosen commit, and rewrite the index and working directory in a single command:

   `git reset --hard <commit_hash>`

Revert rollback:

There are two scenarios to make a rollback to the original state of the working directory and staging area.

1. **Case:** You haven't made any commits while being in the *detached state (HEAD pointed to the other commit but not chronologically the latest):*

   `git reset --hard main`.

2. **Case:** You have made commits being in the *detached state*:

   Key thing to understand is that old commits made in the detached head state do not belong to any branch, and will be stored in Git history only for 30-90 days.

   a. Inspect git ref log and choose the last commit you rollbacked to:

      `git reflog`.

   b. Make a HEAD reset by the number of commit:

      `git reset --hard HEAD@{n}`.

   Alternatively try:

   `git reset --hard <main_branch_name>`. (e.g., `git reset --hard main`.)

▼ Change Authentication Method from HTTPS to SSH:

Issue Description:

You have cloned a Git repository from a remote server and the repository using URL access by default demanding username and password from GitHub. It's possible to change the authentication method from URL to SSH.

Solution:

1. Verify ssh keys valid and added to the GitHub account.
2. Check current push method: `git remote -v`.
3. Change to method: `git remote set-url origin git@gitbub....git`.

▼ Accidentally staged a secret file:

Issue description:

You have accidentally staged a confidential files that was not supposed to be in the Git repository.

Solution:

Unstage:

- `git restore --staged secret.txt`;
  - `git restore` is used for restoring files from different sources.
  - `--staged` tells it to operate on the staging area (index).
- `git reset HEAD secret.txt`;
  - `git reset` is used to move branch pointer or modify the staging area.
  - `HEAD` refers to the current commit.
  - This approach was widely used before Git introduces the `git restore command`.

⚠️**Unstage and also discard changes in working directory:**

- `git restore --staged --worktree secret.txt`;

▼ ❓ Common questions:

▼ Describe the difference between git `restore`, `revert`, `reset` commands:

Into:

The Git commands: `git restore` , `git revert` , `git reset` all serve to undo changes, but they operate at different scope (file vs. commit) and use different safety mechanisms (non-destructive vs. history-rewriting).

Commands description:

1. **Git restore:** file-level undo operations in a working directory and staging area. It never touches the commit history.
2. **Git revert:** Is a safe, non-destructive way to undo a committed changes. It preserves the commit history.
3. **Git reset:** is the history rewriting way to undo commits. It moves the branch pointer (HEAD) to an earlier commit and has different effects on the Staging Area and Working Directory depending on the mode used (soft, mixed, hard).

The difference summary table:

Command:	Primary Scope:	Primary Action:	History Impact	Safety/Use case:
`git restore`	File/Uncommitted	Undo changes in working directory or staging area.	None. Does not touch commits.	High. Managing local changes before committing.
`git revert`	Commit	Undoes a commit by creating a new opposite commit.	Preservers history. Adds a new commit.	Highest. Used for public, shared history.
`git reset`	Commit	Moves the branch pointer (HEAD) to an earlier commit.	Rewrites history. Deletes commit from the branch.	Low. Used for local, private history.

▼ How to set up Git to use a template for commit messages:

Intro:

An organization uses a specific git commit message template, how to set it up?

Solution:

Setting up a Template is a straightforward process consisting of two steps:

1. Create the file:

   Example:

```
# COMPANY COMMIT MESSAGE TEMPLATE
# Ticket: [JIRA-XXXX] or [TASK-YYYY]
# Type: [feat/fix/chore/docs/refactor] (Max 50 chars)

<Enter brief, imperative summary here>

# Body: Explain the WHAT and WHY of the change. Wrap at 72 chars.
```

```
# The HOW is left to the code.

<Enter detailed description here>

# BREAKING CHANGE: If any, describe it here.
```

2. Set up:
   a. Globally (recommended for corporate policy):

      `git config --global commit.template ~/.gitmessage.txt`.

   b. Locally (per-repository):

      `git config commit.template .gitmessage.txt`.

▼ How to create a new repository based on a template:

Intro:

You are working on a project that requires a specific project file & directories structure. How to create a new repository based on an existing template:

Solution:

`git init --template=/path/to/template`.

▼ What data structure is used for Git history:

Answer:

A directed acyclic graph (DAG), where each commit points to its parent or parents.

- **Directed:** The relationships (edges) between commits (nodes) have a specific direction - they point backward in time from a child commit to its parent(s).
- **Acyclic:** There are no loops or cycles in the graph, meaning a commit can never be its own ancestor.
- **Graph:** The history is not strictly linear. Because a commit can have *multiple parents* (in the case of a merge commit), it forms a graph structure that allows for branching and merging.

▼ Version Control System Transitions:

Intro:

A company moves to Git from another version control system. You're required to create a repository, containing all project files, but not their modification history.

Solution:

`git init && cp -r old_project/ . && git add . && git commit -m "Initial commit"`.

▼ 👥 Team collaborations:

▼ Questions:

About workflow:

- What branching strategy does the team use?
- Naming convention for branches? Ticket numbers?

- Should I rebase or merge when updating my branch with main?
- Should I rebase or merge commits in my local branches that haven't been pushed?
- How often should I push my work? (Small and frequent pushes can prevent regular occurrence of conflicts.)
- Is it okay to force-push to my feature branches, or should I avoid that?

About pull requests:

- What's the typical size of a pull request here?
- How many reviewers do I need to approve my PR?
- Are there specific people I should always request reviews from for certain areas o the codebase?
- What's the expected turnaround time for code reviews?
- Should I merge my own PRs after approval, or does someone else do that?

About commit practices:

- What's the commit message template?
- Should I squash my commits before merging them, or keep the full history?
- What Git Hooks are in use and where to get them?
- Do you use signed commits? (Some security-conscious teams require GPG signatures)

About CI/CD pipeline:

- Can I run these checks locally before pushing?
- What should I do if the CI fails on my PR?

About branch protection and permissions:

- Are there protected branches I shouldn't push directly?
- Can I delete my feature branch after merging, or does the team keep them?
- What's the policy on hot-fixes and emergency changes?

About collaboration:

- If I need to collaborate with someone else on a feature, how should we share a branch?
- What do I do if I accidentally commit to main? (Hopefully the answer is "you can't, it's protected!")
- How do you handle dependencies between features in different branches?

▼ Setting up local environment:

Beyond Git configuration, set up the local development environment to match the teams.

Ask about:

- Required development tools and their version.
- Environment variables or configuration files you need.
- Any Git hooks the team uses.
- Local database setup or test data requirements.

- Documentation about the development setup process.

▼ Mistakes to avoid:

- Do not force-push to shared branches.
- Do not commit secret or sensitive data.
- Do not make commits too large.
- Do not let the local feature branch get too far behind main.

Sources:

1. https://www.tutorialspoint.com/git/git-version-control.htm
2. https://www.youtube.com/watch?v=e2IbNHi4uCI&ab_channel=freeCodeCamp.org
3. https://www.conventionalcommits.org/en/v1.0.0/