Git Introduction

- **Resources**:
  - Pro Git book

1. Version Control

- A system that tracks and manages changes to files over time.
- Enables collaboration by allowing multiple people to work on the same project.
- Provides history, rollback capabilities, and branching to experiment safely.

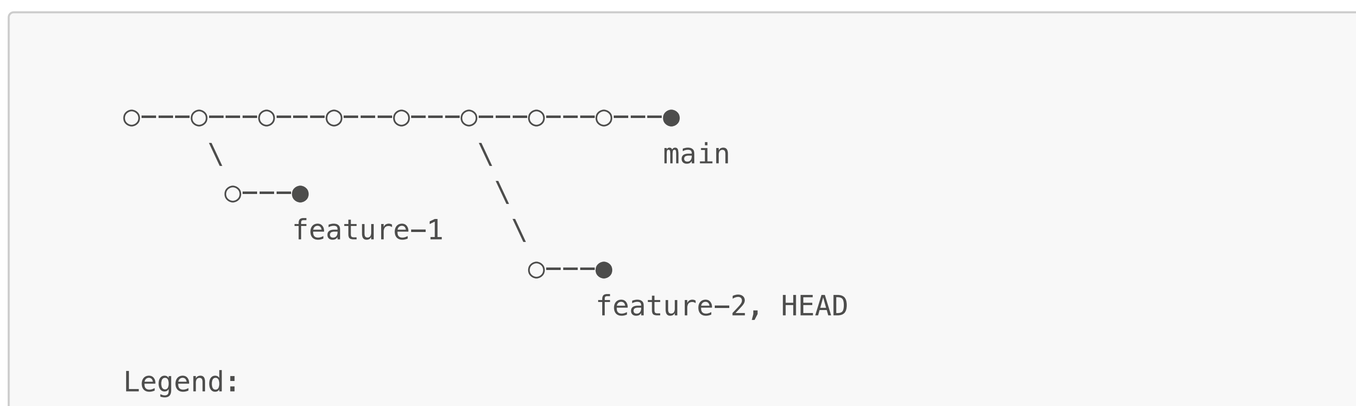
1.1. Git

- Git is a distributed version control system (DVCS).
- In a DVCS, each user has a complete copy of the entire repository, including its history, stored locally.
- Remote hosting platforms (like GitHub, GitLab, or Bitbucket) provide a central place to store, share, and collaborate on code.

2. Git Core Concepts

- **Commit**: A snapshot of your project's files at a specific point in time. It records what changed, who made the change, and when. Each commit includes:
  - **Hash**: Unique identifier for the commit.
  - **Author**: Person who made the commit.
  - **Timestamp**: Date and time of the commit.
  - **Message**: Short description of the changes.
  - **Parent(s)**: Reference(s) to previous commit(s).
  - **Snapshot**: The actual content and changes of the files at that point.
- **Working directory**: The current state of your project files that you see and edit.
- **Staging area (index)**: Where you prepare changes before committing.
- **Branch**: A pointer to a specific commit, allowing parallel development. By default, all repositories start with a `main` branch.
- **HEAD**: A pointer to the current commit or branch. If HEAD doesn't point to a branch, it's called a detached HEAD.
- **Repository (.git directory)**: Stores all commits, branches, and the complete history.

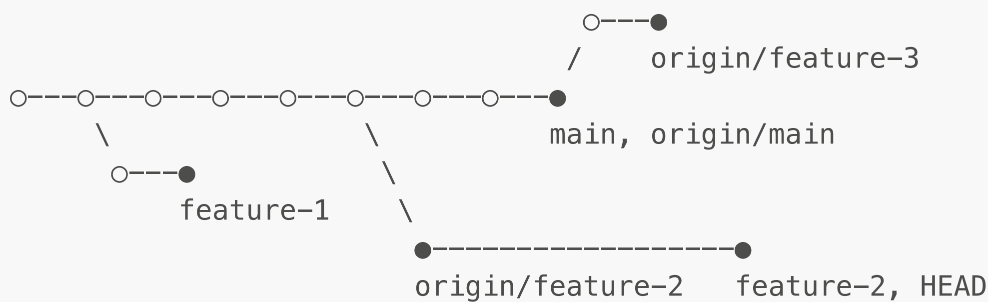
Example of a Git repository structure:



```
    ○          = commit
    ●          = commit with a pointer
    \, -, /    = parent relationship (parent on the left)
    main       = main branch pointer
    feature-1 = feature-1 branch pointer
    feature-2 = feature-2 branch pointer
    HEAD       = current checked out branch
```

- **Remote**: A reference to a version of your repository hosted on another server. Remotes let you push your changes to, and pull updates from, a shared repository.

Example with a remote repository named `origin`:




```
Legend:
origin/main      = points to the same commit as main
origin/feature-2 = one commit behind feature-2
origin/feature-3 = two commits ahead of main
Notice:
- There is no origin/feature-1 branch in the remote repository
- There is no feature-3 branch in the local repository
```

3. Basic Commands

Setup

- `git init` - Start a new repository.
- `git clone <url>` - Copy an existing repository.
- `git config` - Configure username and email.

Daily Workflow

- `git status` - Check the current state, including:
  - The current branch
  - Whether the branch is in sync with any remote branches
  - Changes staged for commit
  - Changes not staged for commit
  - Untracked files

- `git add <file>` - Stage changes in `<file>`.
- `git commit -m "message"` - Save a snapshot of staged changes.
  - When a commit is created, HEAD and the current branch point to it, and the previous commit becomes its parent.
- `git log` - View commit history.
- `git diff` - Show unstaged changes.
- `git restore <file>` - Discard changes in the working directory.

Example output from `git status`:

```
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   main.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   config.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Readme.md
```

Tip: The output of many Git commands is informative and often suggests other useful commands (see above).

Remote Collaboration

- `git remote -v` - List remote repositories.
- `git push` - Send commits to the remote repository.
- `git fetch` - Get remote changes without merging.
- `git pull` - Fetch and merge remote changes into your current branch.

Example: Basic Git Workflow

```
git clone https://github.com/username/repo.git   # Clone a remote repository
git status                                       # Check the status of your working directory
git add filename.txt                             # Stage changes
git commit -m "Describe your changes"            # Commit your changes
git pull origin main                             # Pull the latest changes from the remote
```

```
git push origin main                               # Push your commits to the
remote repository
```

4. Branches

Branches let you work on different features, fixes, or experiments in parallel without affecting the main codebase. Common branching operations:

- `git branch` - List all branches and show the current branch.
- `git branch feature-x` - Create a new branch called `feature-x` from your current commit.
- `git switch feature-x` or `git checkout feature-x` - Switch to the `feature-x` branch. HEAD now points to this branch.
- `git merge feature-x` - Combine the changes from `feature-x` into your current branch.
- `git branch -d feature-x` - Delete the `feature-x` branch locally. Use this after merging if you no longer need the branch.

Pushing a New Branch and Setting Upstream

When you create a new branch locally and want to share it with others, you need to push it to the remote repository. The first time you push a new branch, Git does not automatically know which remote branch it should track. To set this up, use:

```
git push -u origin my-feature-branch
```

- The `-u` (or `--set-upstream`) flag tells Git to set the remote branch (`origin/my-feature-branch`) as the upstream for your local branch.
- After this, you can simply use `git push` and `git pull` without specifying the branch.

Merging Branches

Merging combines changes from one branch (e.g., a feature branch) into another (usually `main`). This is a key part of collaborative development.

When you run `git merge feature-x` while on `main`, Git tries to integrate the changes from `feature-x`:

- If `main` has not changed since `feature-x` branched off, Git performs a **fast-forward merge**. The branch pointer simply moves forward to the latest commit:

  ```
  Before `git merge feature-x`

  ○---●---○---○---○---●
      main, HEAD      feature-x

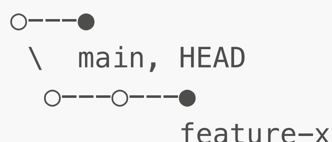

  After `git merge feature-x`

  ○---○---○---○---○---●
  ```

```
                        feature-x, main, HEAD
```

- If the two branches have diverged, changes are in different parts of the code, Git will perform a **true merge**. The new commit is a child of the commits the two branches were pointing to, and both `HEAD` and `main` now point to it.

```
Before git merge feature-x
```




```
After git merge feature-x
```

```
- If the same lines were changed in both branches, a **merge
conflict** occurs. Git marks the conflicting sections like this:

  - When a conflict happens, Git marks the conflicting files and
sections like this:

```text
<<<<<<< HEAD
your changes in the current branch
=======
their changes from the branch being merged
>>>>>>> feature-x
```

- You must edit the file to keep the correct code, then stage the resolved file with `git add` and complete the merge with `git commit`. This will create a new commit with your manual conflict resolution in the same way as before.

Edit the file to resolve the conflict, then stage it with `git add` and complete the merge with `git commit`. If you want to abort the merge, use `git merge --abort`.

5. Pull Requests

Pull requests (PRs) are a feature of platforms like GitHub, GitLab, and Bitbucket that help teams review and discuss code changes before merging them into the main branch. (Some platforms call them merge requests.) They also allow for automated checks (tests, linting, etc.) before merging.

A pull request is a request to merge changes from one branch (often a feature or bugfix branch) into another (usually `main` or `develop`). It allows others to review, comment, and suggest improvements
```

before the changes are integrated. Pull requests are a key part of collaborative development, especially in open source and team environments.

Typical Pull Request Workflow

1. Create a new branch and make your changes.
2. Push your branch to the remote repository.
3. Open a pull request on the platform's website, selecting the source and target branches.
4. Team members review the changes, discuss, and may request modifications.
5. Once approved, the pull request is merged—either automatically or by a maintainer.

6. Tagging

Tags are used to mark specific points in your repository's history as important, such as releases or milestones. They are commonly used to identify release versions (e.g., `v1.0`, `v2.1.3`).

Example: Basic Tag Usage

```
git tag v1.0                              # Create a lightweight tag
git tag -a v1.0 -m "Release version 1.0"  # Create an annotated tag (recommended for releases)
git tag                                   # List all tags
git push origin v1.0                      # Push a specific tag to the remote repository
git push --tags                           # Push all tags to the remote
```

Tags are read-only references and are not meant to be changed or moved. They are especially useful for marking release points before or after merging branches.

7. Gitflow and Trunk-Based Development

Gitflow

Reference: Gitflow

Gitflow is a branching model that defines a structured workflow for managing releases, features, and hotfixes. It uses several long-lived branches:

- `main`: Always contains production-ready code.
- `develop`: Integration branch for features; contains the latest delivered development changes.
- `feature/*`: Used for developing new features; branched off from `develop`.
- `release/*`: Used to prepare for a new production release; branched from `develop` and merged into both `main` and `develop` when finished.
- `hotfix/*`: Used to quickly patch production releases; branched from `main` and merged into both `main` and `develop`.

Pros:

- Clear structure for large teams and projects with scheduled releases.
- Isolates different types of work (features, releases, hotfixes).

Cons:

- Can be complex and heavy for small teams or fast-moving projects.

Trunk-Based Development

Reference: Trunk-Based Development

Trunk-based development is a simpler workflow where all developers work in short-lived branches or directly on a single branch (often called `main` or `trunk`).

- Developers create small, frequent pull requests or commits to the main branch.
- Feature branches are short-lived (ideally less than a day or two).
- Emphasizes continuous integration and frequent collaboration.

Pros:

- Encourages rapid integration and reduces merge conflicts.
- Simpler process, well-suited for continuous delivery and DevOps.

Cons:

- Requires discipline to keep changes small and the main branch stable.
- May not suit projects needing long-term feature isolation.

Summary Table:

Workflow	Main Branches	Feature Branches	Release Branches	Hotfix Branches	Typical Use Case
Gitflow	main, develop	Yes (long-lived)	Yes	Yes	Large teams, scheduled releases
Trunk-Based Development	main (trunk)	Yes (short-lived)	No	No	Fast-moving, CI/CD projects