## 1.1 Why Git?

### What is Version Control?

- A system that tracks changes to files over time, enabling:
    - **Collaboration**: Multiple people can work on the same files without overwriting each other's changes.
    - **History Tracking**: Allows you to revert to previous versions of a file.

### Why Git is Better Than Other VCS?

- **Distributed System**: Every developer has a full copy of the repository, enabling offline work.
- **Speed**: Git operations like commits, branching, and merging are very fast due to local operations.
- **Open Source and Widely Supported**: Works with many platforms, tools, and hosting services (GitHub, GitLab, Bitbucket).

### Configuration:-

Run these commands after installation:

Set your name:

```
git config --global user.name "Vikash Kumar"
```

Set your email:

```
Git config --global user.email "vikashkr@gmail.com"
```

View Configuration:

```
git config --list
```

----------------------------

1.3.1 Initializing a Repository

```
git init
```

1.3.2 Cloning a Repository

```
git clone <repository-url>
```

1.3.3 Checking Repository Status

```
git status
```

1.3.4 Adding Files

git add <file name> or [git add . ] → It will add all files

1.3.5 Committing Changes

```
git commit -m "Your commit message"
```

1.3.6 Viewing Commit History

git log or [git log --oneline]

# **************Chapter 2*************

## 2.1 What Are Branches in Git?

- A **branch** in Git is like a parallel line of development.
- It allows you to work on a feature, bug fix, or experiment without affecting the main codebase (usually called `main` or `master`).

## Why Use Branches?

- **Isolation**: Keep work on features/bugs separate from the main branch.
- **Collaboration**: Multiple developers can work on different branches simultaneously.
- **Version Control**: Helps track and manage different versions of the codebase.

## Structure of Branches

- **HEAD**: A pointer to the current branch you're working on.
- **Default Branch**: Typically `main` or `master`

**2.2 Basic Branching Commands**

## 2.2.1 Creating a Branch

- git branch <branch name>

### 2.2.2 Viewing Branches

git branch

To view remote branches:- [git branch –r]

To view both local and remote branches:- [git branch -a]

### 2.2.3 Switching Branches

git switch <branch_name> or git checkout <branch_name>

## 2.3.1 Merging a Branch

First, switch to the branch you want to merge into (e.g., main):→ git switch main

Merge the feature branch:→ git merge feature/login

## 2.3.2 Fast-Forward Merge

- If the main branch hasn't changed since the feature branch was created, Git simply moves the main branch pointer forward.

### 2.3.3 Three-Way Merge

- If both branches have changes, Git creates a new commit that combines the histories of the branches.

### 2.3.4 Resolving Merge Conflicts

- **What is a Conflict?**

  Happens when changes from two branches conflict.

**Steps to Resolve**:

1. Git will show conflict markers in the file:

```
<<<<<<< HEAD

code from main branch

=======

code from feature/ branch name

>>>>>>> feature/branch
```

2. Manually edit the file to resolve the conflict.
3. Add the resolved file to the staging area:
   ```
   git add <file>
   ```
4. Complete the merge:
   ```
   git commit
   ```

**Chapter 2: Working with Branches**

---

**2.1 What Are Branches in Git?**

- A **branch** in Git is like a parallel line of development.
- It allows you to work on a feature, bug fix, or experiment without affecting the main codebase (usually called main or master).

## Why Use Branches?

- **Isolation**: Keep work on features/bugs separate from the main branch.
- **Collaboration**: Multiple developers can work on different branches simultaneously.
- **Version Control**: Helps track and manage different versions of the codebase.

## Structure of Branches

- **HEAD**: A pointer to the current branch you're working on.
- **Default Branch**: Typically main or master.

---

### 2.2 Basic Branching Commands

#### 2.2.1 Creating a Branch

```
git branch branch_name
```

- Example:→ git branch feature/login
- This creates a new branch feature/login.

#### 2.2.2 Viewing Branches

```
git branch
```

  - Shows all local branches. The current branch is marked with *.
- To view remote branches:

```
  git branch -r
```

- To view both local and remote branches:

```
git branch -a
```

#### 2.2.3 Switching Branches

```
git switch branch_name
      Or
git checkout branch_name
Ex:- git switch feature/login
```

- Git moves HEAD to the new branch, allowing you to work on it.

---

### 2.3 Merging Branches

#### What is Merging?

- Merging integrates changes from one branch into another.
- Usually, changes from a feature branch are merged into the main branch.

#### 2.3.1 Merging a Branch

- First, switch to the branch you want to merge into (e.g., main):
  git switch main
- Merge the feature branch:
  git merge feature/login

#### 2.3.2 Fast-Forward Merge

- If the main branch hasn't changed since the feature branch was created, Git simply moves the main branch pointer forward.

#### 2.3.3 Three-Way Merge

- If both branches have changes, Git creates a new commit that combines the histories of the branches.

### 2.3.4 Resolving Merge Conflicts

- **What is a Conflict?**
  - Happens when changes from two branches conflict.
- **Steps to Resolve**:
  1. Git will show conflict markers in the file:

     ```
     <<<<<<< HEAD
     code from main branch
     =======
     code from feature/login branch
     >>>>>>> feature/login
     ```

  2. Manually edit the file to resolve the conflict.
  3. Add the resolved file to the staging area:
     ```
     git add <file>
     ```
  4. Complete the merge:
     ```
     git commit
     ```

---

#### 2.4 Deleting Branches

**Why Delete a Branch?**

- Once a feature is merged, the branch is no longer needed and can be deleted to keep the repository clean.

**Delete a Local Branch**
```
git branch –d <branch_name>
```

- Force delete (if the branch is not fully merged):

  ```
  git branch -D <branch_name>
  ```

**Delete a Remote Branch**

```
git push origin --delete <branch_name>
```

### 2.6 Tags

### What Are Tags?

- Tags are references to specific commits, often used for marking releases (e.g., v1.0).

### Creating a Tag

- Lightweight tag:
  ```
  git tag <tag_name>
  ```
- Annotated tag (includes metadata like date, author):
  ```
  git tag -a <tag_name> -m "Tag message"
  ```

## Viewing Tags

git tag

## Pushing a single tag:

git push origin <tag_name>

## Push all tags:

git push origin –tags

******************Chapter 3 *******************

## 3.1 Remote Repositories

### What Is a Remote Repository?

- A **remote repository** is a version of your Git project hosted on a server (e.g., GitHub, GitLab, Bitbucket).
- Collaborators can **push**, **pull**, and share code from/to this repository.

### Basic Remote Commands

1. **View Remote Repositories**:
   - Shows the list of remotes:→ git remote –v

2. **Add a Remote Repository**:

   Link your local repository to a remote:→ git remote add origin repository-url

3. **Remove a Remote Repository**:→ git remote remove remote_name

4. **Fetch Changes from Remote**:→ git fetch origin

5. **Pull Changes from Remote**:→ git pull origin branch_name

6. **Push Changes to Remote**:→ git push origin branch_name

## 3.2 Forking and Cloning

### What Is Forking?

- Forking creates a copy of someone else's repository in your GitHub/GitLab account.
- Used for contributing to open-source projects.

### Steps to Fork and Contribute:

1. Fork the repository from the GitHub UI.
2. Clone the forked repository to your local machine:
   git clone <forked-repository-url>
3. Add the original repository as an upstream remote:
   git remote add upstream <original-repository-url>

4. Fetch upstream changes:
   git fetch upstream

5. Sync your fork with the original repository:
   git pull upstream main

6. Make changes, commit, and push to your fork:
   git push origin <branch_name>

7. Create a **Pull Request (PR)** on GitHub.

### 3.3 Pull Requests

**What Is a Pull Request?**

- A pull request (PR) is a proposal to merge changes from one branch (usually a feature branch) into another branch (e.g., main).

**Steps to Create a Pull Request:**

1. Push your changes to your fork:
   git push origin <branch_name>
2. Go to the repository on GitHub/GitLab.
3. Click **"New Pull Request"**.
4. Compare changes and submit the PR.

## 3.4 Rebasing

### What Is Rebasing?

- Rebasing moves or replays your branch commits onto another branch.
- Used to keep feature branches up to date with the `main` branch.

### Rebase Workflow:

1. Checkout the feature branch:→ git checkout feature/login
2. Rebase onto the `main` branch:→ git rebase main
3. Resolve any conflicts.
4. Force push the rebased branch:→ git push --force-with-lease origin feature/login

*******************Chapter 4************

### 4.1 Stashing Changes

**What Is Stashing?**

- Stashing allows you to temporarily save changes (working directory) without committing them, so you can switch branches or pull updates.

**Common Stashing Commands:**

1. **Stash Current Changes**:→ git stash
2. **View Stash List**:→ git stash list
3. **Apply Stashed Changes**:→ git stash apply
4. **Apply and Drop Stash**: → git stash pop

5. **Stash Specific Files**:→ git stash push –m “stash message” <file>
6. **Drop Stash**:→ git stash drop
7. Clear All Stashes:→ git stash clear

### 4.2 Cherry-Picking

**What Is Cherry-Picking?**

- Cherry-picking allows you to apply a specific commit from one branch to another.

**Use Case:**

- When you need just one commit from a feature branch without merging the entire branch.

**Command:**

1. Identify the commit hash from git log.
2. Cherry-pick the commit:→ git cherry-pick <commit-hash>

## 4.3 Reset, Revert, and Checkout

**4.3.1 Reset**

- Used to undo changes **locally**.

**Types of Reset**:

1. **Soft Reset**: Moves HEAD but keeps changes staged.
   ```
   git reset --soft <commit-hash>
   ```
2. **Mixed Reset (default)**: Moves HEAD and unstages changes.
   ```
   git reset <commit-hash>
   ```
3. **Hard Reset**: Discards all changes.
   ```
   git reset --hard <commit-hash>
   ```

**4.3.2 Revert**

- Safely **undo changes** in history by creating a new commit.
- Use when changes have already been pushed.

**Command**:
```
git revert <commit-hash>
```

**4.3.3 Checkout**

- Switches between branches or restores files.

**Restore a Specific File**:
```
git checkout <commit-hash> -- <file>
```

**4.4 Rewriting History**

**Amending Commits**

- Modify the last commit:→ git commit --amend
- Used for correcting commit messages or adding files.

**Interactive Rebase**

Modify multiple commits (squash, edit, reorder) in history.

**Command**:

git rebase -i HEAD~<n>

Replace pick with:

squash: Combine commits.
edit: Modify a specific commit.
reword: Change the commit message.

**4.5 Git Hooks**

**What Are Hooks?**

Hooks are custom scripts that run before/after specific Git events (e.g., commit, push).

**Common Hooks:**

1. **Pre-commit Hook**: Runs checks before committing.
2. **Post-commit Hook**: Executes actions after committing.
3. **Pre-push Hook**: Checks before pushing to a remote repository.

# **************Chapter 5 ********************************

**5.1 What Are Git Workflows?**

A **Git workflow** defines how branches, commits, and merges are managed in a team. It ensures:

Smooth collaboration.
Code quality via reviews and CI/CD.
Organized history and clean releases.

The most common workflows are:

1. Feature Branch Workflow
2. Git flow Workflow
3. HitHub Flow
4. Trunk-Based Development

**5.2 Feature Branch Workflow**

1. Developers create a **feature branch** off main for every feature, bug fix, or experiment.
2. Work is completed, commits are made, and the branch is pushed.
3. Changes are reviewed and merged back into main.

Steps:

1. Create a feature branch: → git checkout –b feature/new-feature
2. Commit changes:→ git commit –am “Add new feature implementation”
3. Push the branch: → git push origin feature/new-feature
4. Create a Pull Request(PR) for review.
5. Merge the branch after approval.

## 5.4 GitHub Flow

A simplified workflow ideal for small teams and continuous delivery.

There’s only one long-running branch: `main`.

**Steps:**

1. Create a branch for the task:→ git checkout –b feature/new-feature
2. Push changes: → git push origin feature/new-feature
3. Open a Pull Request (PR) for review
4. Merge the PR into main
5. Deploy changes