INTRODUCTION TO GIT AND GITHUB

By RUTH OLUMO

What is Git?

- **Git** is a popular version control system.
- A **Version Control System (VCS)** is a tool that helps developers track and manage changes to source code (or any set of files) over time.
- It is used for:
  - Tracking code changes
  - Tracking who made changes.
  - Coding collaboration.

Why use Git?

- Over 70% of developers use Git.
- Developers can work together from anywhere in the world.
- Developers can view the full project history.
- Developers can revert to earlier versions of a project.

Core Git Concepts

- **Repository:** A folder where Git tracks your project and its history.
- **Clone:** Make a copy of a remote repository on your computer.
- **Stage:** Tell Git which changes you want to save next.
- **Commit:** Save a snapshot of your staged changes.

Continuation

- **Branch:** Work on different versions or features at the same time.
- **Merge:** Combine changes from different branches.
- **Pull:** Get the latest changes from a remote repository.
- **Push:** Send your changes to a remote repository.

Setup & Installation

- **Windows:** Download installer from git-scm.com
- **macOS:** Install via Homebrew [*brew install git*] or download the .dmg file and drag Git to your Applications folder.
- **Linux:** Run ***sudo apt–get install git*** on Ubuntu

Initial Configuration

- *git config --global user.name "Your Name"*
- *git config --global user.email "[email@example.com]“*
- The above sets the identity for commit
- Set a default text editor.
  - **Example: Set VS Code as Default Editor**
  - *git config –global core.editor “code –wait”*
- Add Git to your PATH. You can use Git commands in any terminal window

Basic Git Workflow

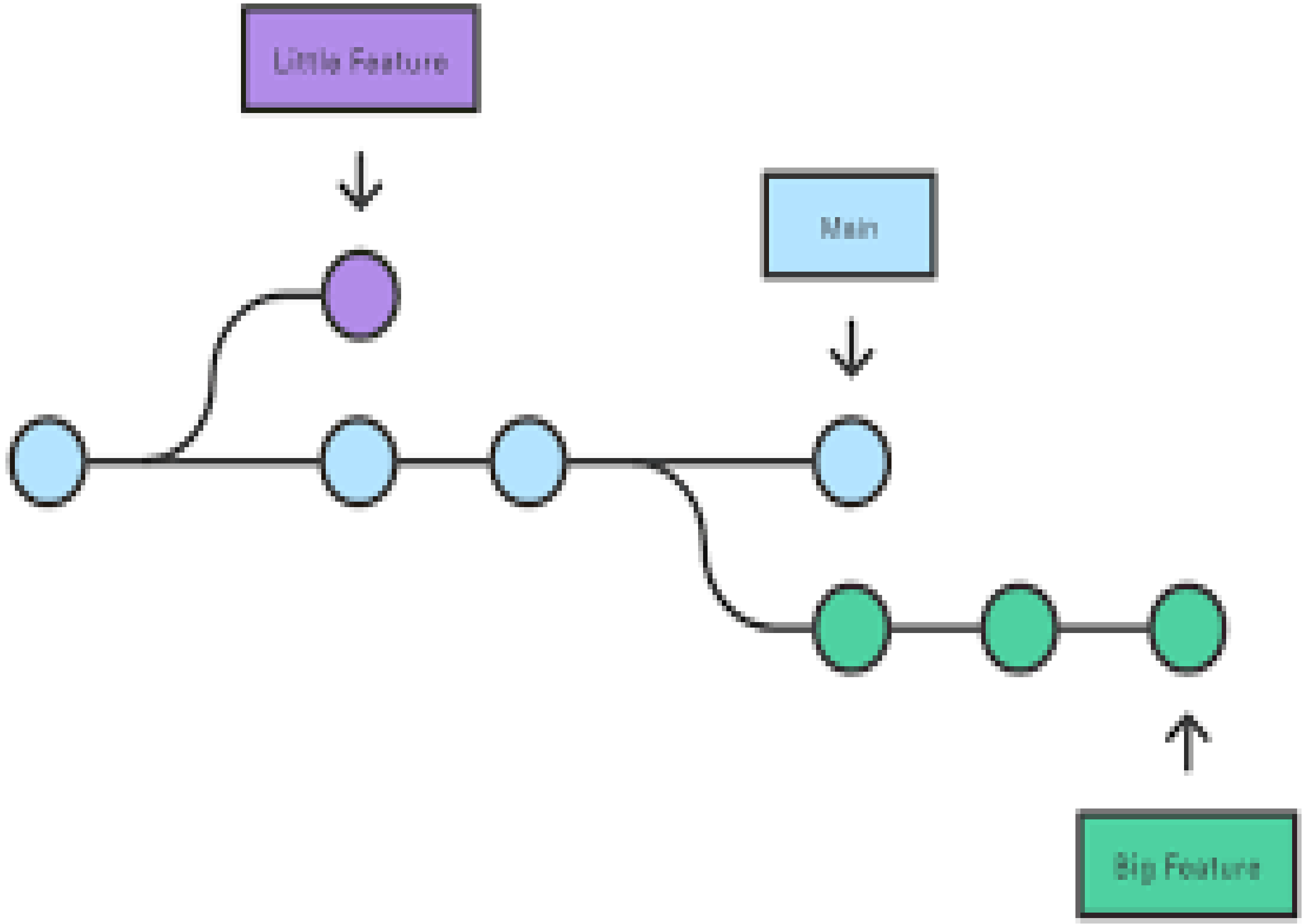
- **Initialize or clone:** *git init*
  - OR *git clone <repo-url>*
- **Edit and Stage**
  - *git add <file>* - Stage a file
  - *git add --all Or git add –A* - Stage all changes
  - *git status* – See what is staged
  - *git restore --staged <file>* - Unstage a file.

Continuation

- **Commit**
  - *git commit –m "message"* - Commit staged changes with a message
  - *git commit –a –m "message"* - Commit all tracked changes(skip staging)
  - *git log* – See commit history
- **Pushing the changes to the repository**
  - *git push*

Git Branching

- A branch is like a separate workspace where you can make changes and try new ideas without affecting the main project
- Branches let you work on different parts of a project, like new features or bug fixes, without interfering with the main branch.


Little Feature
Main
Big Feature

Git Branching

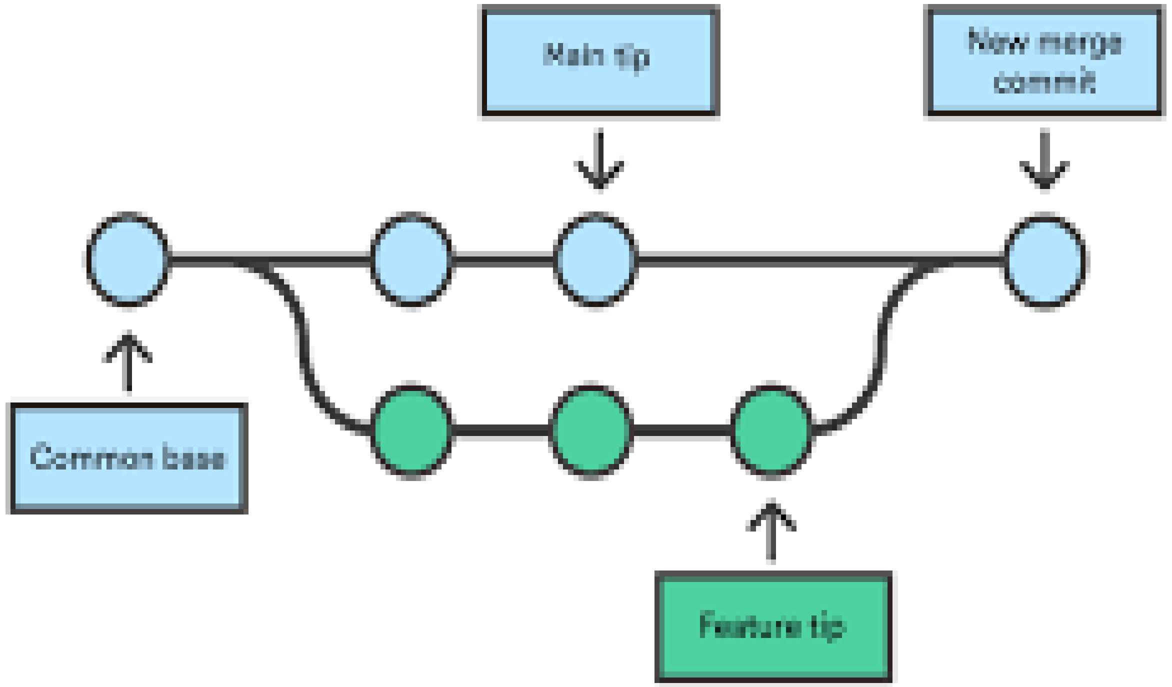
- **Creating a new branch**
  - *git branch <branch_name>*
- **Listing all branches:** *git branch*
- **Switching between branches:** *git checkout <branch_name>*
- **Deleting a branch:** *git branch –d <branch_name>*

Git Merging

- Merging means combining the changes from one branch into another
- It's how you bring your work together after working separately on different features or bug fixes.
- **Merging branches:** *git merge <branch_name>*
  - First, switch to the branch you want to merge into.
  - Run the merge command with the branch name you want to combine into.

NOTE!

Always commit or stash your changes before staging a merge


Main tip
New merge commit
Common base
Feature tip

Git Stashing

- Git stash lets you save your uncommitted changes and return to a clean working directory.
- You can come back and restore your changes later.
- **Common Use cases:**
  - **Switch branches safely:** Save your work before changing branches.
  - **Handle emergencies:** Stash your work to fix something urgent, then restore it.
  - **Keep your work-in-progress safe:** Avoid messy commits or losing changes.
- **Stash your work**: *git stash OR git stash push –m "message"*
- **Apply the stashed changes back:** *git stash apply*
- **Drop a Stash:** *git stash pop*

Git and GitHub Integration

What is GitHub?

- A cloud-based platform for hosting Git repositories.
- Provides a user interface for version control, issue tracking, and collaboration.
- Owned by Microsoft.

Common GitHub Commands

- **Push:** Upload local commits to GitHub
  - *git push origin main*
- **Pull:** Download changes from GitHub to your machine
  - *git pull origin main*
- **Clone:** Copy a GitHub repository to your local system
  - **e.g** *git clone https://github.com/username/repo.git*
- **Fork:** Copy a GitHub repository to your personal GitHub account.

GitHub Features

- **Pull Requests**: Propose and discuss code changes
- **Issues**: Report bugs, plan tasks
- **Actions**: Automate workflows like tests or deployments
- **Forking**: Copy someone's repo to your GitHub to make independent changes

Collaborating with GitHub

Typical Team Workflow

1. Fork a repository (copy to your account)
2. Clone to your machine.

   git clone https://github.com/yourname/forked-repo.git
3. *Create a feature branch*
   1. *Git checkout –b feature-x*
4. Make Changes – (*git add . ; git commit –m "message"*)
5. Push Changes.
6. Create a **Pull Request (PR)** on GitHub to propose changes
7. Team reviews, discusses, and **merges** PR into main branch.

Pro Tip

Use .gitignore **to avoid pushing unnecessary files(e.g. node_modules, .env)**

Branching Strategies (Collaboration Modality)

What is a Branching Strategy?

- A **branching strategy** is a **workflow or plan** for how developers manage, collaborate, and organize their **source code using branches** in a version control system like **Git**.
- A **branching strategy** defines **when**, **why**, and **how** to create, merge, and delete those branches.

Why is it Important?

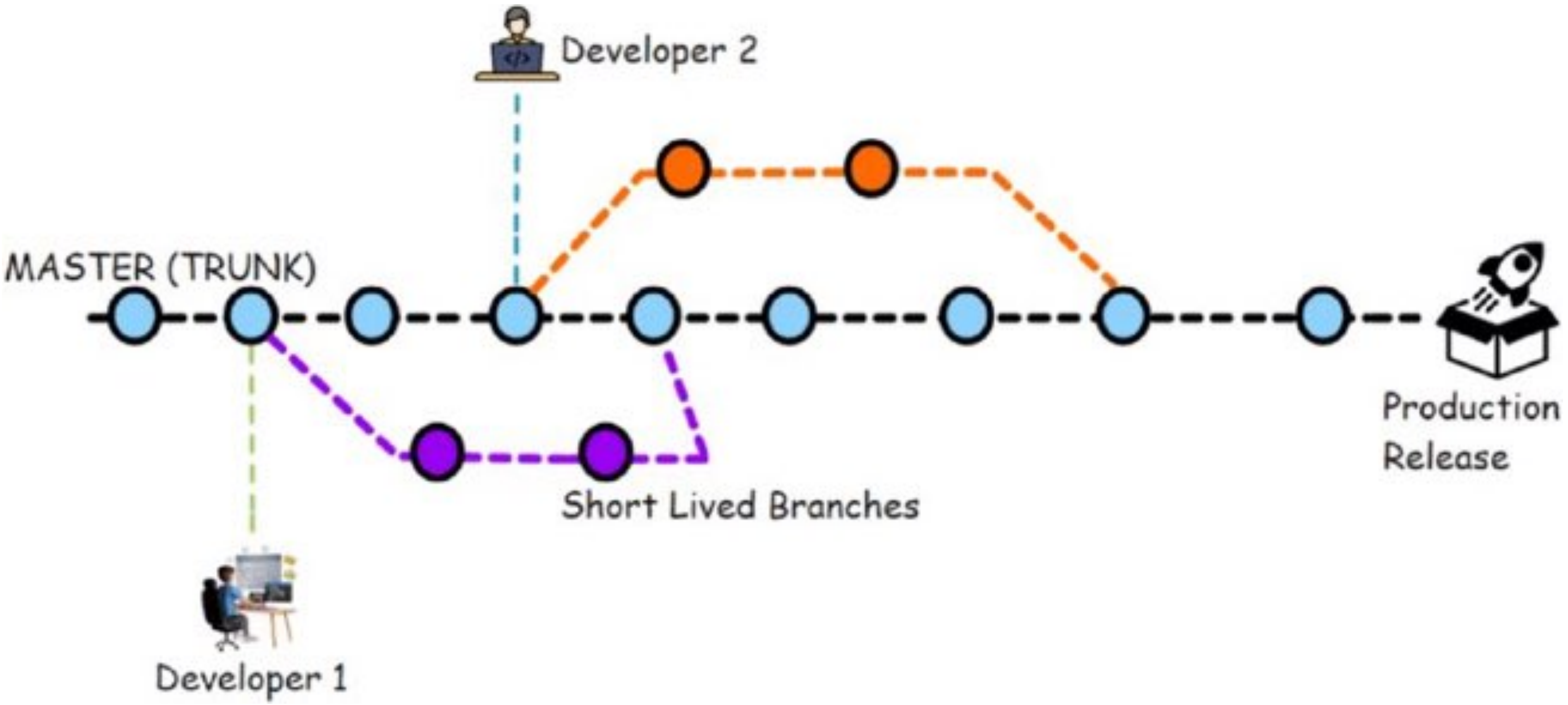
- Prevents conflicts when many developers work on the same codebase.
- Keeps the main codebase (e.g., main or master) stable.
- Supports testing, feature development, hotfixes, and releases more efficiently

Common Branching strategies

- Feature Branching
- Git Flow
- GitHub Flow
- Trunk Based Development

Trunk Based Development

- Developers commit directly to **main**
- Short-lived branches
- Very frequent merges (daily)
- Relies on CI/CD for testing


Developer 2
MASTER (TRUNK)
Production Release
Short Lived Branches
Developer 1

Advantages of TBD

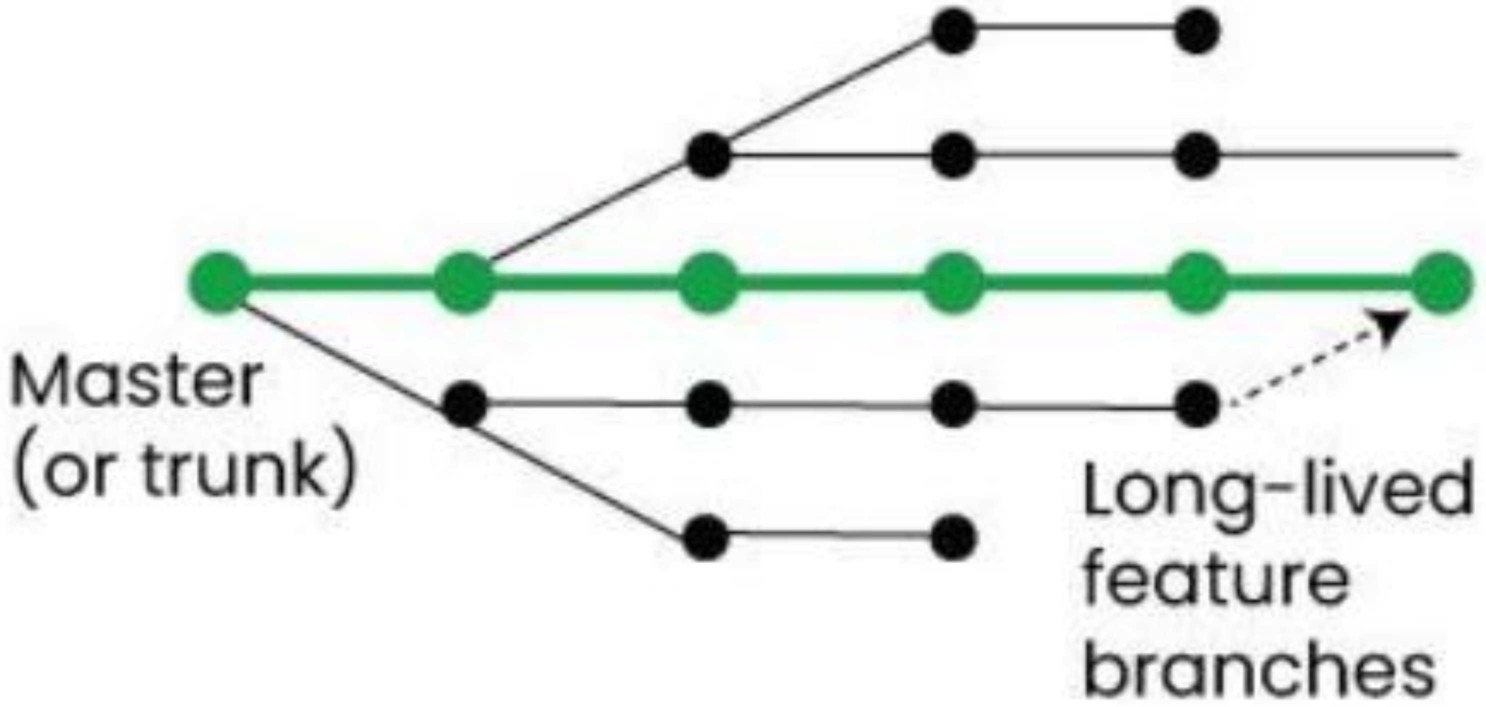
- Faster feedback loops
- Reducing merge conflicts
- Improved Collaboration
- Easier code reviews
- Continuous integration and delivery

Disadvantages of TBD

- Limited isolation
- Increased pressure on testing.
- Requires a strong team culture

Feature Branching

- Each new feature is developed in its own branch.
- Long-lived branches.
- Merged into main or develop after completion.


Master
(or trunk)
Long-lived
feature
branches

Advantages of Feature Branching

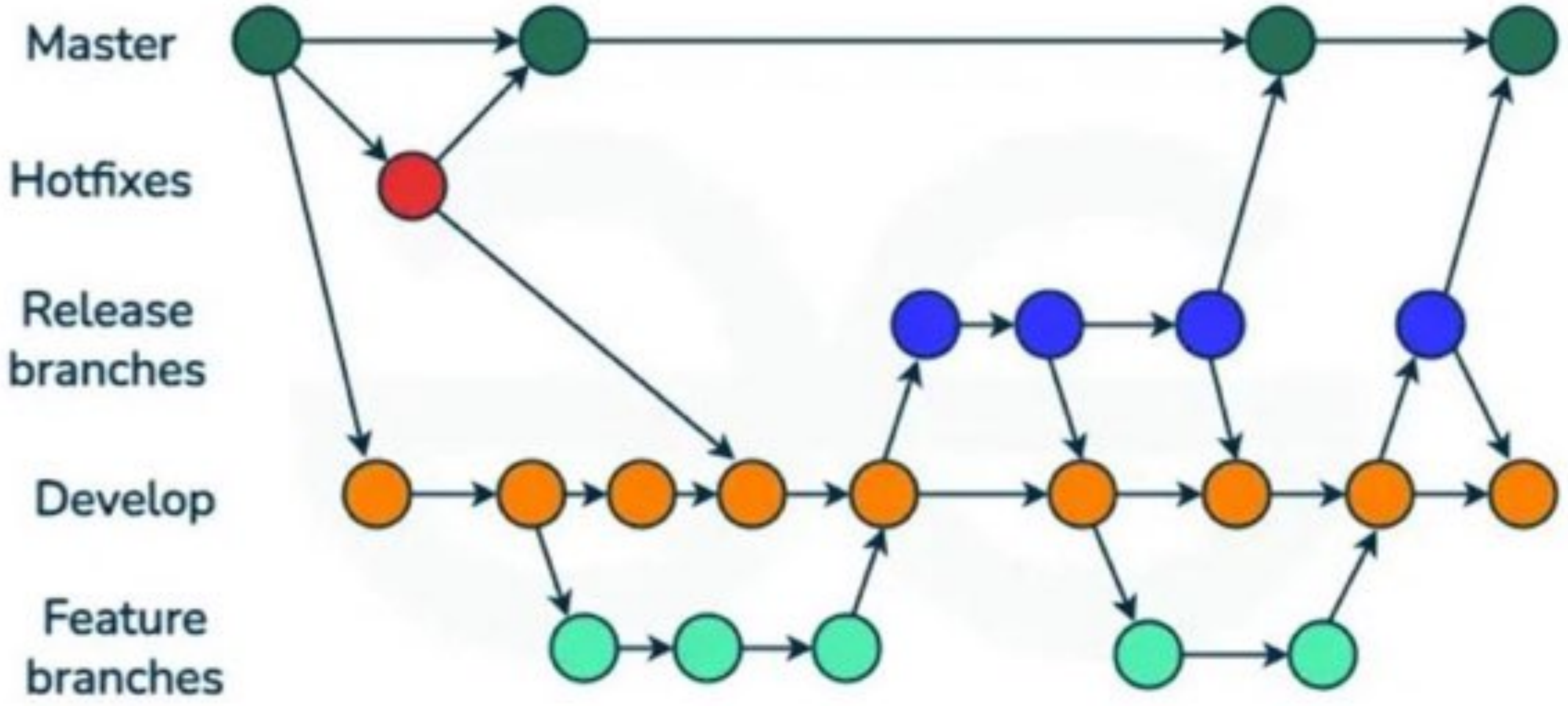
1. **Isolated Development**
   Each feature is independent, reducing the risk of interfering with other parts of the system.
2. **Enables Code Reviews**
   PRs allow team members to review code and catch bugs before integration.
3. **Maintains a Stable Main Branch**
   main or develop remains clean and deployable while features are developed in isolation.
4. **Parallel Development**
   Multiple developers or teams can work on different features at the same time.
5. **Traceability**
   It's easier to trace what changes were made for a particular feature or bug fix

Disadvantages of Feature Branching

1. **Merge Conflicts**
   If branches live too long without syncing, they may lead to complicated conflicts.
2. **Delayed Integration**
   Features that take too long to merge can drift away from the current state of the main codebase.
3. **Slower Feedback**
   Bugs might only appear once the feature is merged and tested with the rest of the system.
4. **Branch Management Overhead**
   Requires discipline in naming, tracking, and cleaning up branches.

Git Flow

- Popular in large teams/projects.
- Uses multiple main branches:
  - **main/master** → production-ready code
  - **develop** → integration branch for features
  - **feature/**, **release/**, **hotfix/** → specific purposes


Master
Hotfixes
Release branches
Develop
Feature branches

Advantages of Git Flow

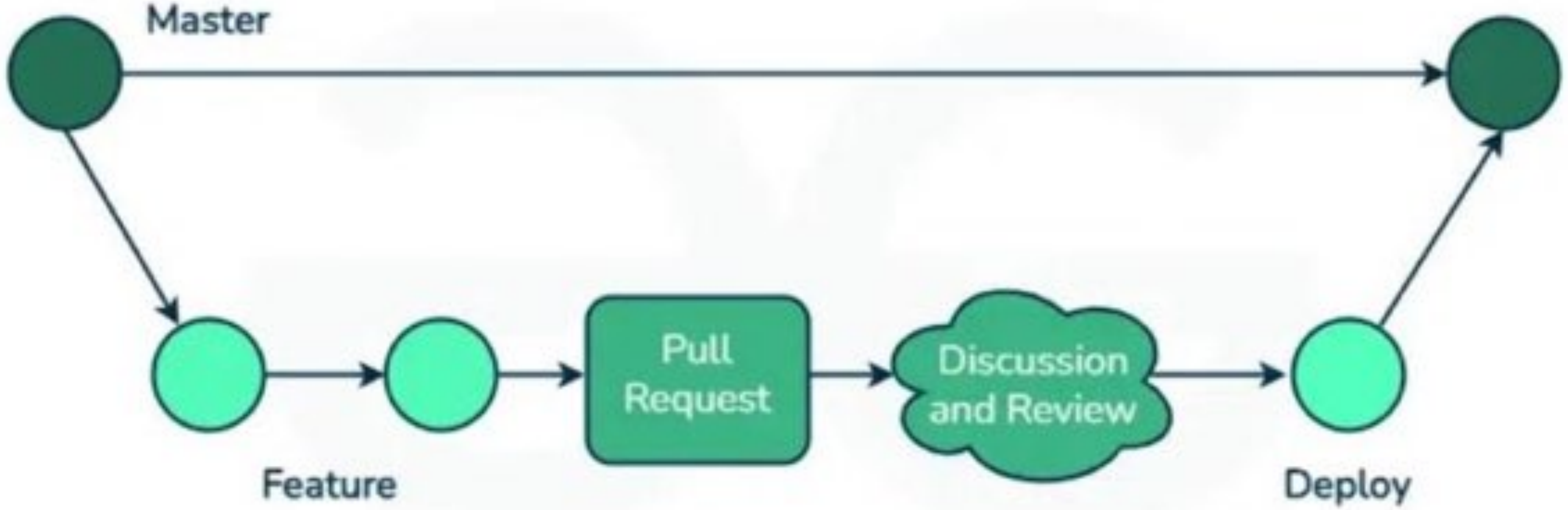
1. **Highly Structured**
   Each branch has a specific purpose. This brings clarity to the development process.
2. **Safe Releases**
   Code undergoes integration, testing, and staging before reaching production.
3. **Support for Multiple Versions**
   You can maintain older releases (via hotfix) while continuing development on the next version.
4. **Good for Teams with Defined Roles**
   Developers, testers, and release managers can work independently on different branches.
5. **Ideal for Large Projects**
   Especially where long-term maintenance, staging environments, and scheduled releases are required.

Disadvantages of Git Flow

1. **Complexity**
   Involves many branches and merge operations. May be overwhelming for small teams or fast-paced projects.
2. **Slower Integration**
   Features take longer to reach production due to multiple intermediate steps. May not be ideal for teams with very short release cycles or those striving for continuous delivery.

GitHub Flow

- Lightweight, simple workflow.
- Only **main** branch is permanent.
- Developers create short-lived feature branches, open pull requests, merge quickly.


Master
Feature
Pull Request
Discussion and Review
Deploy

Advantages of GitHub Flow

1. **Lightweight and Simple**
   Very easy to understand and implement. Fewer long-lived branches to manage.
2. **Encourages Continuous Deployment**
   Ideal for projects that require rapid iteration and frequent updates.
3. **Promotes Collaboration**
   Pull requests serve as a hub for team discussion, reviews, and visibility.
4. **Easy Integration with GitHub Tools**
   Seamless with GitHub Actions, Issues, Projects, and more.

Disadvantages of GitHub Flow

1. **Not Ideal for Versioned Releases**
   Doesn't support managing multiple production versions at the same time.
2. **No Dedicated Release Process**
   Everything merges into main, so additional steps are needed if you want staged releases.
3. **Requires Reliable Testing Infrastructure**
   Without automated testing, broken code could easily reach main.

Choosing a Branching Strategy

- Consider **team size** (small vs. large).
- Look at **project complexity**.
- Think about **merge conflict risks**.
- Ensure it supports **collaboration & code reviews**.
- Choose something that's **easy to maintain long-term**.

REFERENCES

- Introduction to Git and GitHub

Thank You!!

Q&A