SHARE
EDUCATE • NETWORK • INFLUENCE

Practical Git Development on the Mainframe

Anthony Anter
DevOps Architect
BMC Software

Who is this guy!!

- Worked for a major US credit card company as a DevOps architect and evangelist creating an enterprise DevOps offering for the company to automate their entire delivery pipeline
- Managed end to end DevOps pipelines and development for both Distributed and Mainframe teams
- Between 2018 & 2020, I architected and delivered an end-to-end Mainframe DevOps toolchain and migrated the entire portfolio onto this new platform
- I joined BMC to evangelize DevOps on the Mainframe and help Companies envision what is possible with their most important platform
- I love to read about and discuss App Development DevOps across both the Mainframe and distributed environments as well as help organizations automate everything

In the Beginning

Mainframe systems had their own Source Code Management systems, and the code was stored on platform.

Builds and deploys were handled on platform as well

As the mainframe became more open though, things changed. Source Code started moving off platform into GIT

What is Git?

Developed by Linus Torvalds, Git is a decentralized version control system which today is by far the most popular version control system in use. Over 90% of organizations and 93% of developers currently use Git for their source code management.

Switching from your current Source Code Management system to Git changes the way your development team creates and delivers software

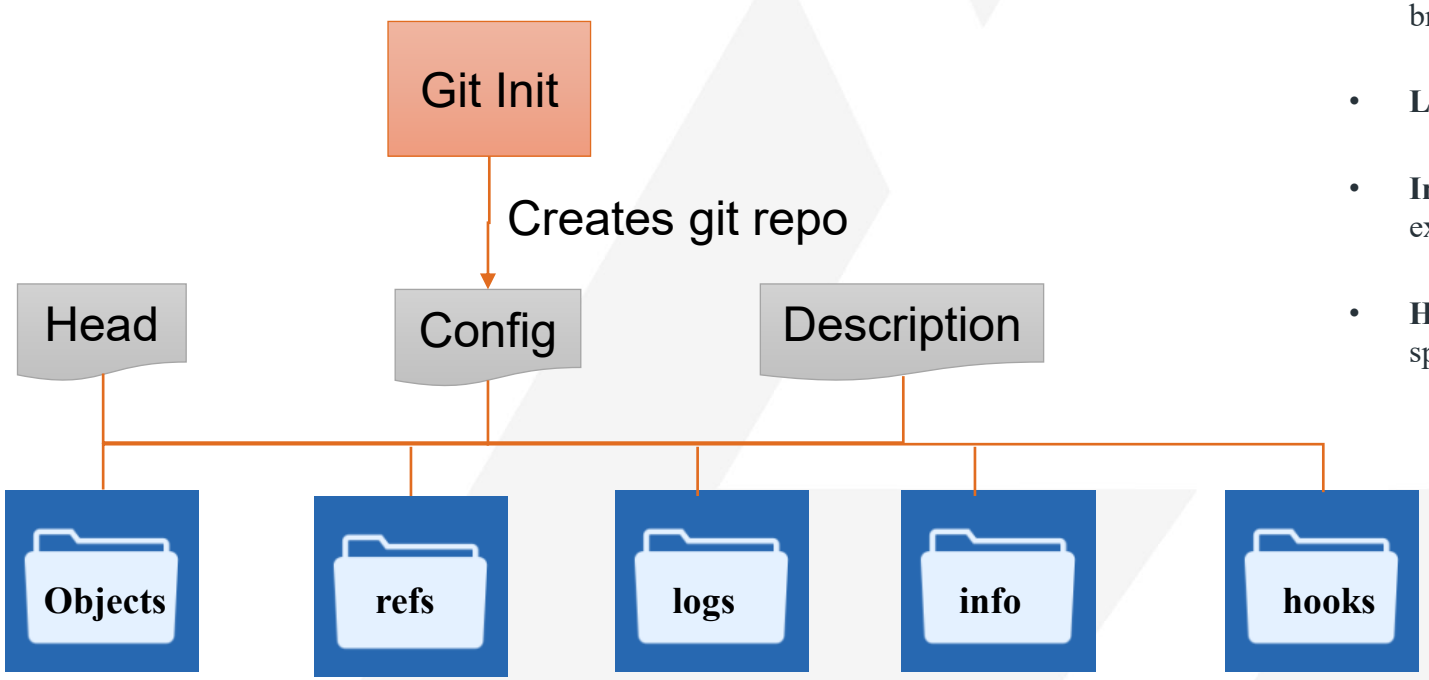
- Git supports Agile and DevOps workflows with distributed version control, flexible branching, and collaboration features
- Task branching, code reviews, and enhance efficiency, transparency, and quality
- Git enables teams to work in parallel, experiment safely, and deliver changes rapidly
- Integrate Git into your Agile workflow to streamline development, improve collaboration, and accelerate delivery

    - Most customers ultimately want to adopt Git as their enterprise-wide decision.

How does Git work?

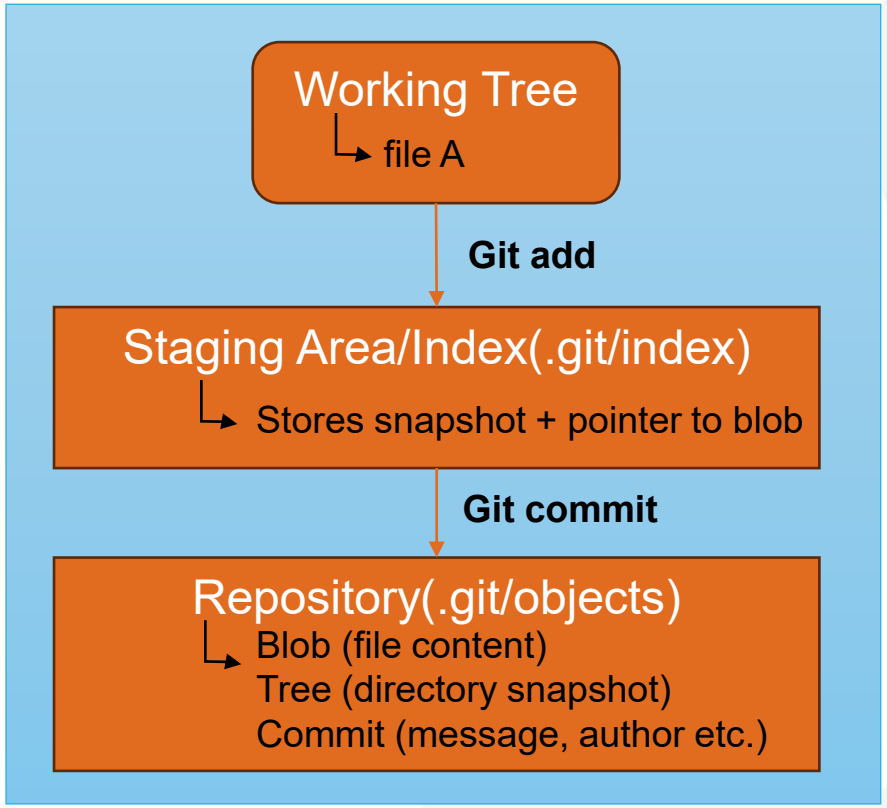
Basically, Git is a content addressable file system with version control built on top

- Everything is stored and retrieved via SHA-1 hash of the contents versus filename of location
- Files are stored as snapshots versus diffs
- All internal data, history and metadata is is stored via a .git folder in your working directory



- **HEAD**: A file that points to your current "branch"
- **config**: A settings file for your repo. Holds details like your name, email, and other Gi preferences (editable anytime).
- **description**: A simple text file for a short note about your project .→ no usage
- **objects folder**: The "storage room" for your project's data
- **refs folder**: The "labels" folder. Keeps pointers to specific points in your history, like branch names "main" or tags "v1.0". You can run command **git reflog**
- **Logs folder :** it records every change in the branch like commits , checkout etc..
- **Info folder :** local only configuration space, store settings only applicable to local rep exclude file which acts like a git ignore but only for this repo and has highest priority
- **Hooks folder :** An automation folder which can have scripts that runs automatically a specific point in git workflow

How Git stores files internally



Git stores everything as an object in .git/objects/

There are three main types

Blob→ files content (stores the contents of a single file – noting else)

Tree→ directory structure (stores a directory structure, including filenames, etc.

Commit→ snapshot + parent + metadata (stores metadata about a snapshot and point to a tree object representing the repository at that point)

.git/objects → all blob, trees, commits stores as hashed files
.git/refs/heads/→branch pointer → latest commit hash
.git/head → points to current branch
.git/index → staging area

The 4 Stages of a Git Workflow

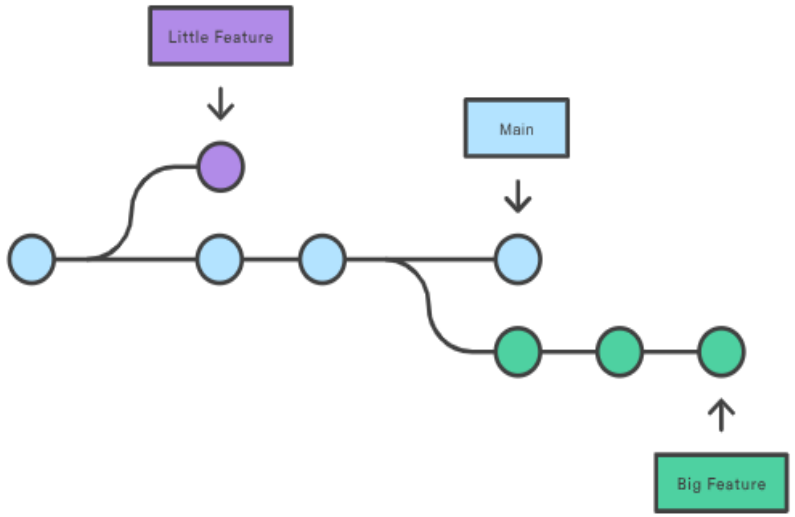
Working Directory: This is your local workspace on your computer where you create, edit, and delete files. Files in this area are considered *modified* or *untracked* until you decide to include their changes in the version control system.

Staging Area (Index): This is a temporary holding area that acts as a middle ground between the working directory and the local repository. You use the git add command to stage changes, selectively choosing which modifications should be included in the next commit. This gives you precise control over what gets saved in each snapshot.

Local Repository: When you run the git commit command, the changes from the staging area are permanently saved as a snapshot in your local repository. This area contains the complete history of your project and its commits, all stored locally on your machine.

Remote Repository: This is a shared repository hosted on a central server, such as GitHub, GitLab, or Bitbucket. It serves as a central point for collaboration and backup. You use git push to upload your local commits to the remote repository, and git pull to fetch and integrate changes made by others into your local copy

Git Branching

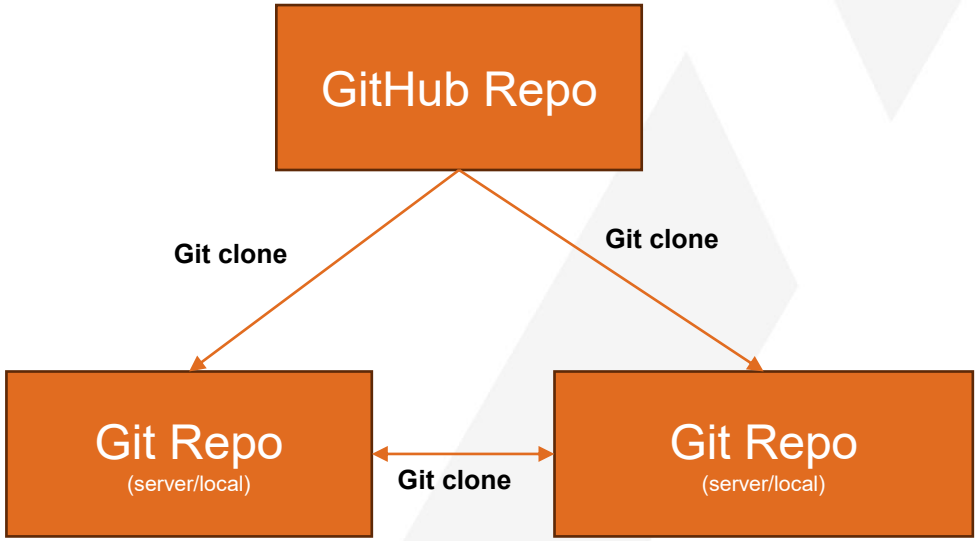


A Git Branch is a separate workspace where a developer can make changes without affecting the main project. Changes made to a branch are isolated and do not affect anything but that branch

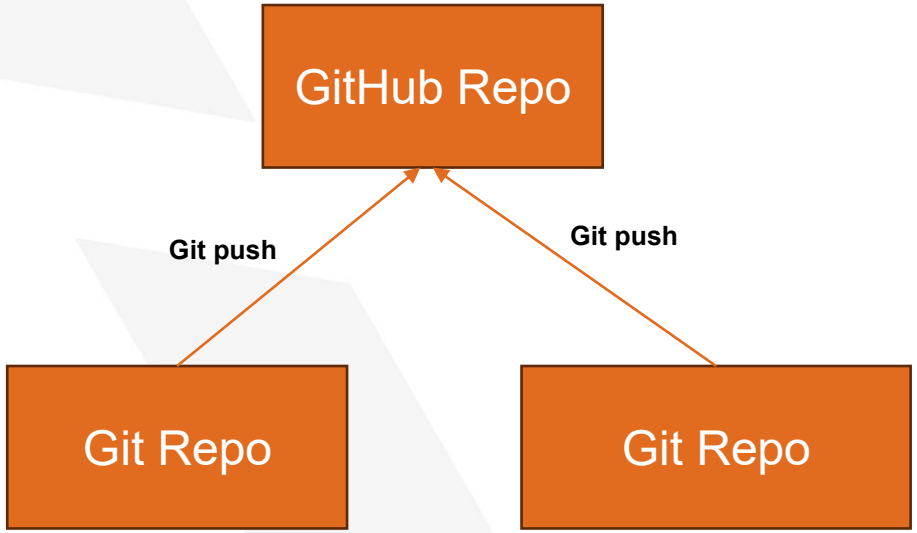
git branch: Lists all local branches
git branch <branch-name> : Creates a new branch with the specified name without switching to it
git checkout <branch-name> : Switches to an existing branch in the repository
git checkout -b <branch-name> : Creates a new branch with the specified name and immediately switches to it
git switch <branch-name> : An alternative to the checkout command

Cloning and Pushing

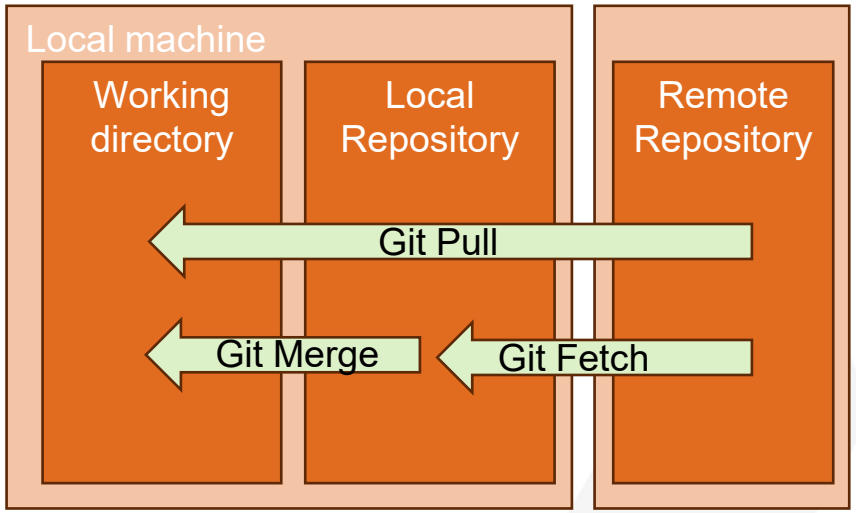
A git clone is a local copy of the repository including the history, commits, branches and tags that allows you to work locally



A git push is used to copy your local changes to a remote repository



Fetching, Pulling & Merging



Git Fetch: Downloads changes to local repo cache only, no merge

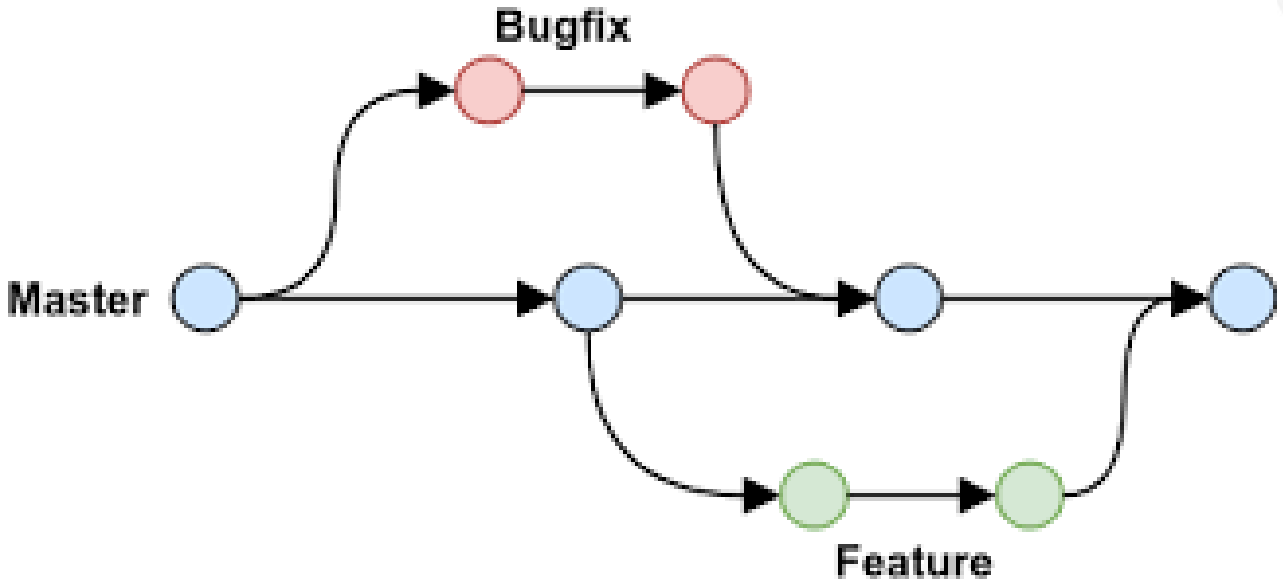
Git Merge: Used to combine changes from multiple branches into a single branch

Git Pull: Does a fetch and a merge in a single operation, fetching from the remote repo and then merging those changes

*These are different than a **pull request** which is not a git command but a feature of distros like GitHub, GitLab and Bitbucket*

How do I manage my code?

Branching strategies: A way to organize your work across multiple lines of development. Basically, how you organize your work across multiple independent branches for release.

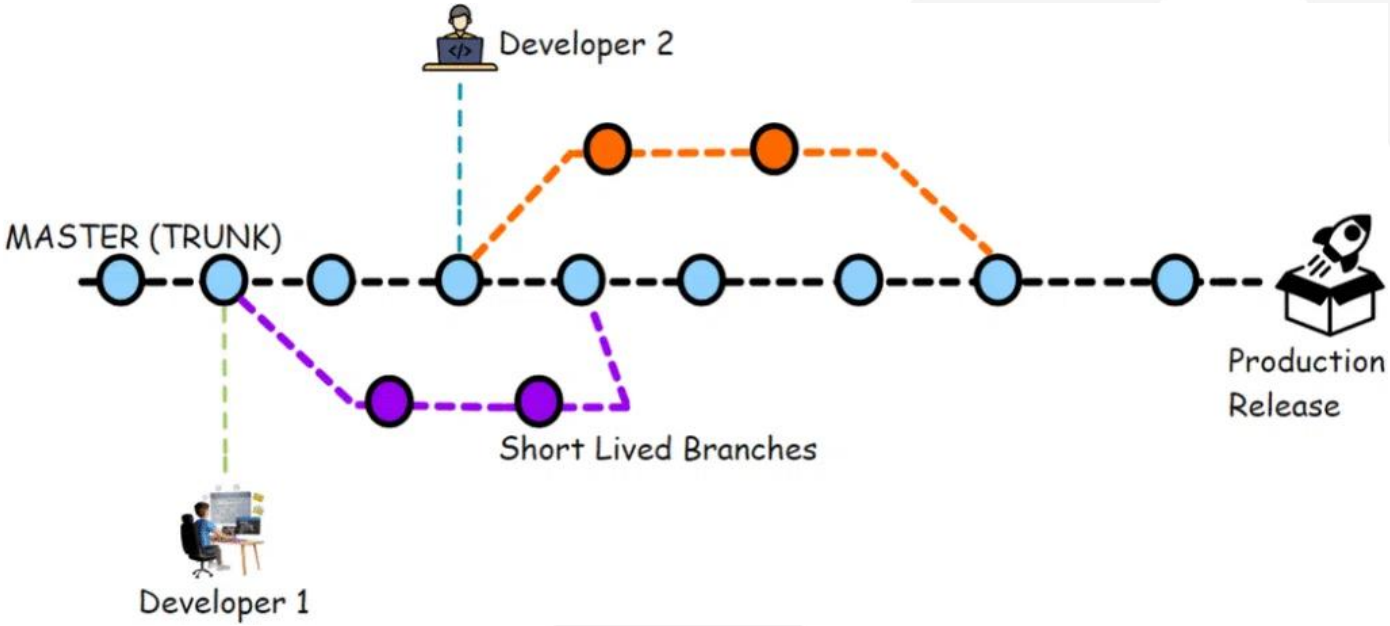


Options:

- GitHub Flow
- Git Flow
- Trunk-Based Development

Trunk Based Development

Trunk-Based Development is a source control branching model where developers collaborate on code in a single branch called "trunk" (main), using very short-lived feature branches or committing directly to trunk.



Use Cases

- ✓ High-maturity engineering teams
- ✓ Microservices architecture
- ✓ Projects with extensive automation
- ✓ Continuous deployment environments
- ✓ Google, Facebook-style development

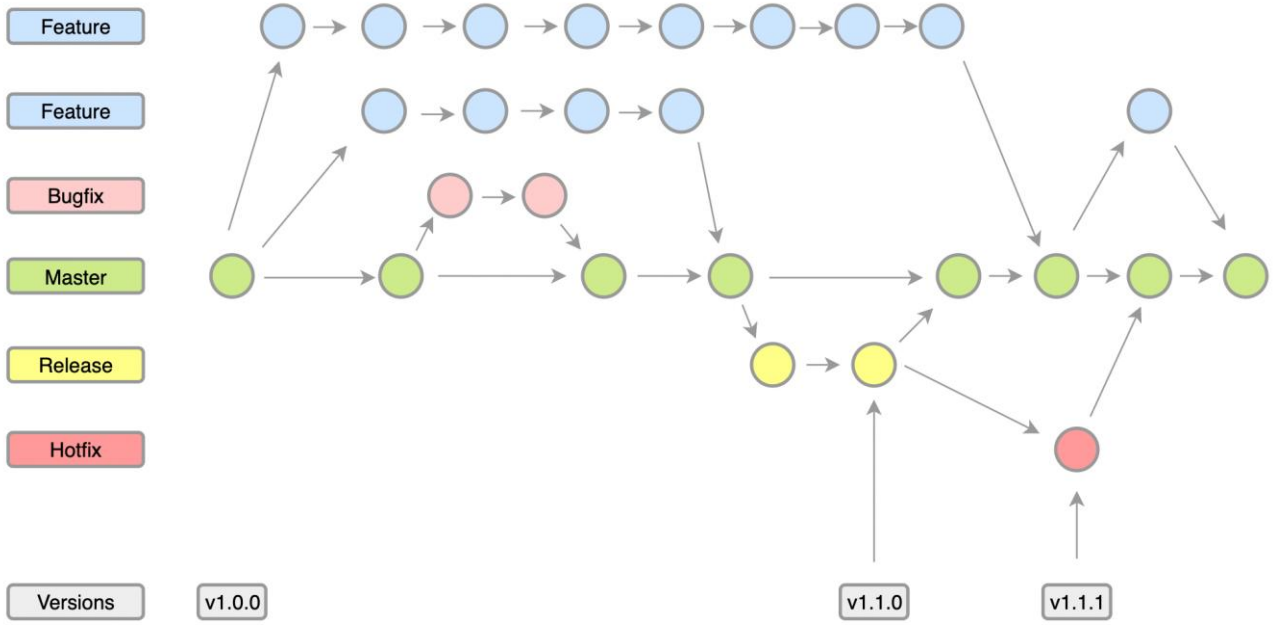
Key Characteristics:

- ✓ Branches live < 1 day
- ✓ Frequent integration
- ✓ Feature flags for incomplete work
- ✓ Robust automated testing

Advantages	Disadvantages
Maximum integration frequency	Requires significant infrastructure
Reduced merge conflicts	Needs highly skilled team
Fast feedback loops	Demands robust testing
Scales with team size	High initial setup cost
Encourages small, atomic changes	Risk if CI/CD fails

GitHub Flow

GitHub Flow is a lightweight, branch-based workflow designed for continuous deployment. It's simple, straightforward, and perfect for teams that deploy frequently.



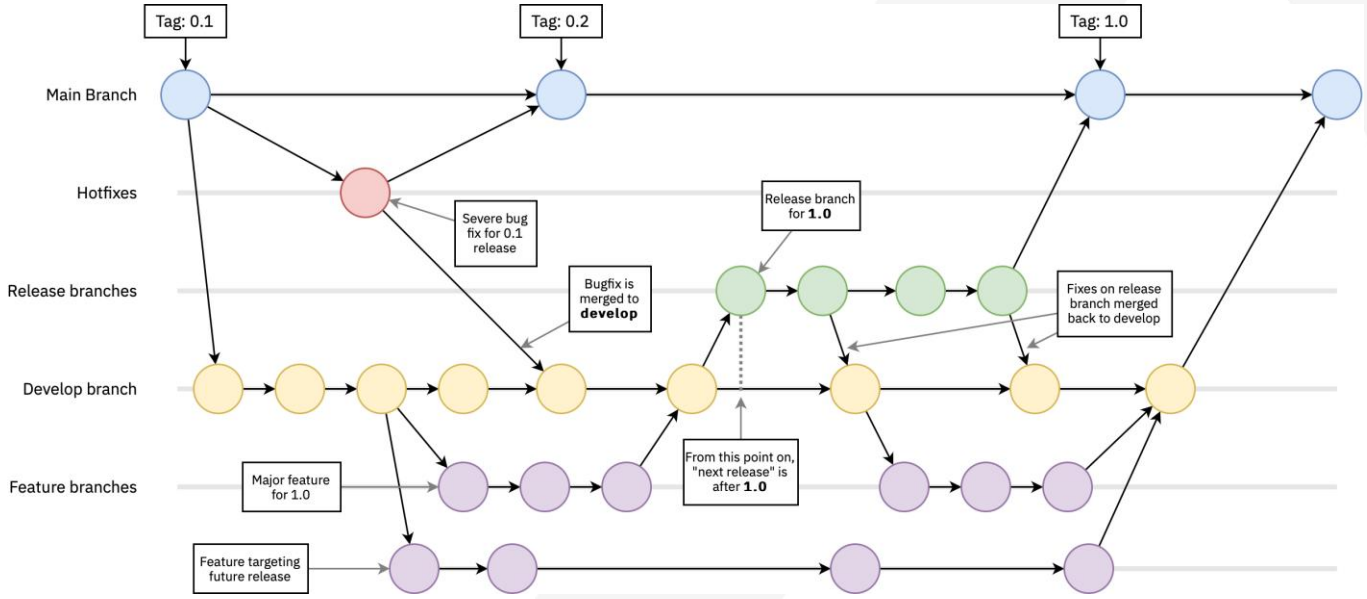
Use Cases

- ✓ Continuous Integration/Continuous Deployment (CI/CD)
- ✓ Applications with frequent releases
- ✓ Small to medium-sized teams
- ✓ Projects that don't require multiple production versions

Advantages	Disadvantages
Simple and easy to understand	Less suitable for versioned releases
Fast deployment cycle	No support for multiple production versions
Encourages continuous delivery	Requires robust CI/CD pipeline
Good for small teams	

GitFlow

Git Flow is a branching model designed for projects with scheduled releases. It uses multiple long-lived branches and supports versioned releases with clear separation between development and production.



Use Cases

- ✓ Software with versioned releases (v1.0, v2.0)
- ✓ Desktop applications
- ✓ Mobile apps
- ✓ Projects requiring release planning
- ✓ Large teams with dedicated QA cycles

Advantages	Disadvantages
Clear separation of concerns	Complex for beginners
Supports multiple production versions	Slower release cycle
Structured release process	More merge conflicts possible
Emergency hotfix capability	Requires disciplined adherence
Great for scheduled releases	

A common mistake – the Google Trap

X Google uses Trunk-Based Development, so should we!
X Facebook deploys 1000x per day, let's do that!
X Netflix uses microservices, we need that too!
X This mobile app uses Git Flow, we should copy it!

THE REALITY

Google has:	You might have:
1,000+ engineers	5-50 engineers
Infrastructure	Limited budget
Tooling investment	Basic CI/CD or none
99.99% test coverage	30% test coverage
Feature flag infrastructure	No feature flags

Before choosing ANY strategy, answer these critical questions:

Question 1: What Are Your Current Developer Steps?

Question 2: How Mature Is Your CI/CD?

Question 3: What will work best for my developers, my team and my org?

Every team is different. Every application is different. There is not a right or wrong answer.

Getting ready to be Git Native

- Break your code down into smaller pieces – 5GB best practice for a repo
- Have a plan – understand how you are going to work and what it takes to get there
- Don’t make your git into a Mainframe
- Git has nothing to do with your IDE
- You will not be working in the same way

Your feedback is important!

SHARE
EDUCATE • NETWORK • INFLUENCE

Submit a session evaluation for each session you attend:

www.share.org/evaluation