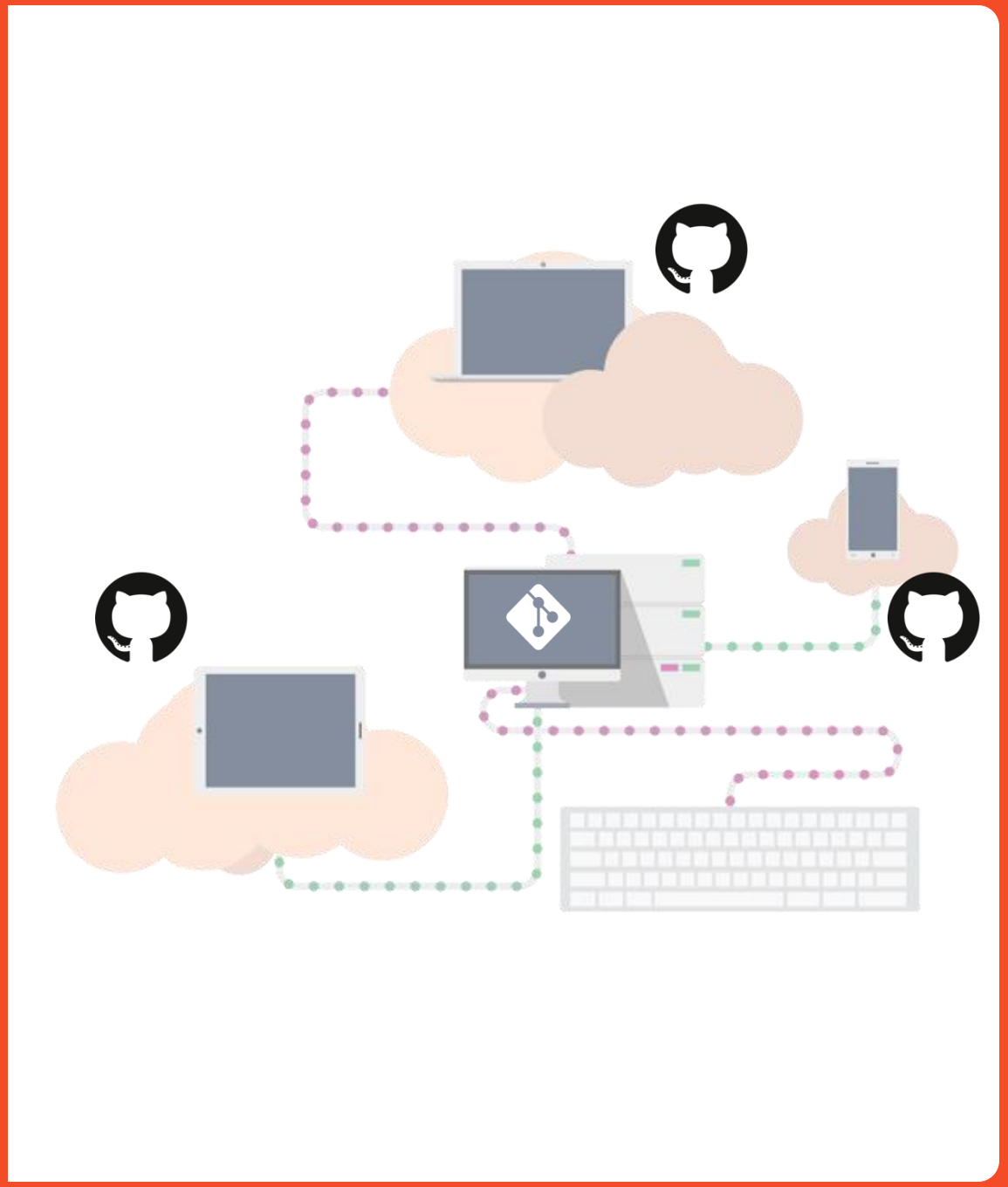
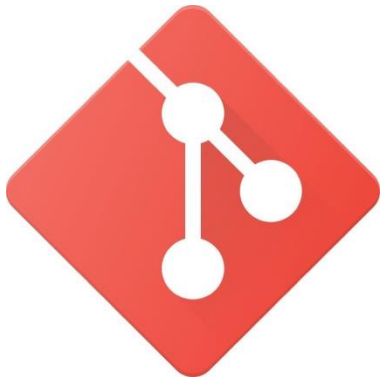
Git & GitHub Bootcamp: Build, Track & Collaborate

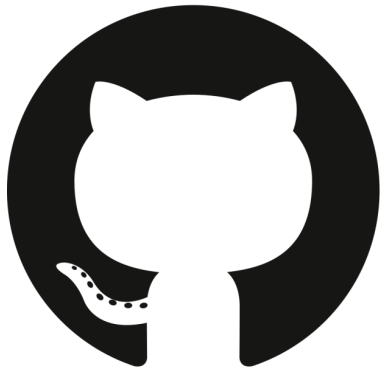


By – Thinknyx Technologies LLP

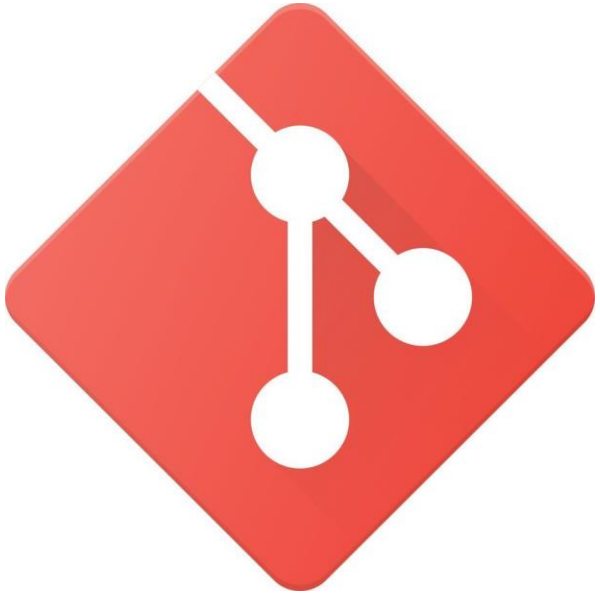
Version Control

- Developer
- DevOps Enthusiast
- Software Engineering Team





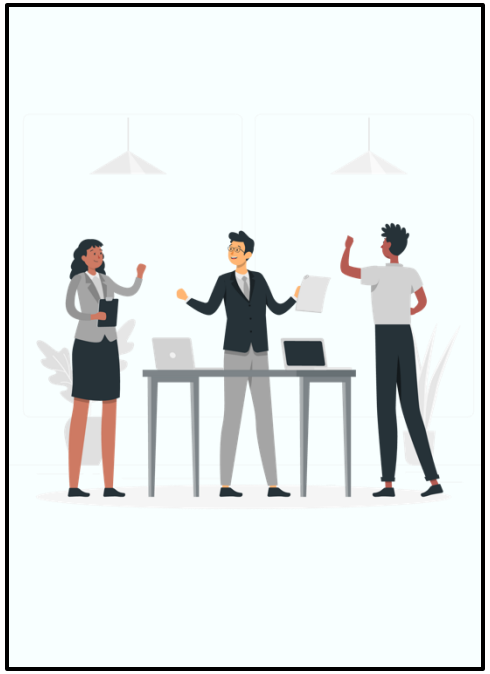
- Code
- Collaboration
- Creativity



Yogesh Raheja

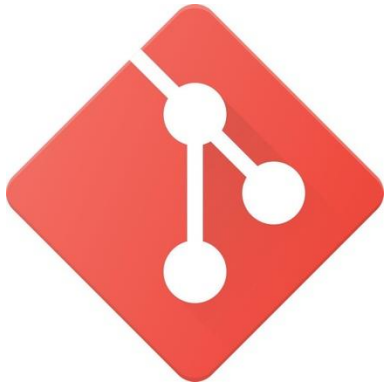
Dheeraj Sain

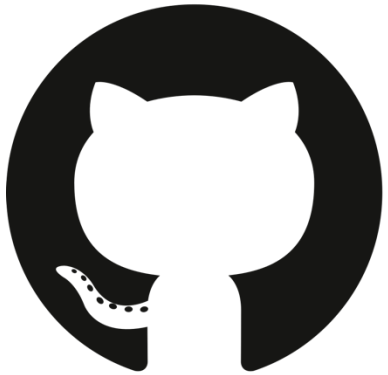
Madhuri Jha



Thinknyx Team

How Will This Course Work?





- **Introduction to Version Control**
- **Git Fundamentals**
- **Branching, Merging & Collaboration**
- **How Teams Use GitHub?**
- **Pull requests, Issues, and Forking Projects**
- **Advanced Git & GitHub Features**
- **GitHub Projects, Wikis, Actions, & Pages**

Introduction to VCS
(Version Control System)

Introduction to VCS

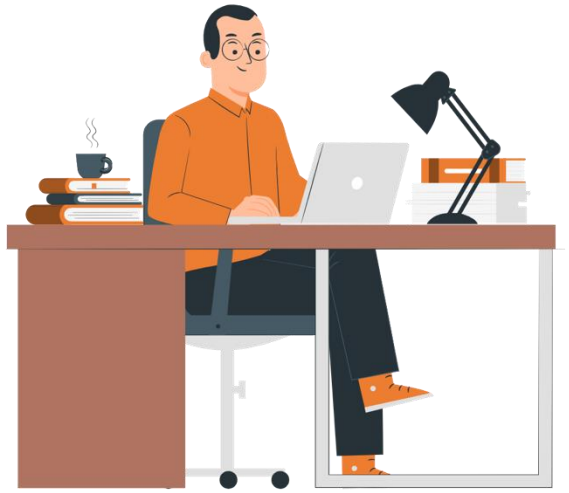
Section Overview

- What is Versioning
- Version Control Systems (VCS)
- Key benefits of using VCS
- Source Code Management (SCM)
- Different types of VCS
- Popular tools

Versioning, Version Control System & Source Code Management

What is Versioning

→ Process of tracking changes to a product over time



What is Versioning

Challenges Before Versioning

Tracking Changes

Recovering Old Versions

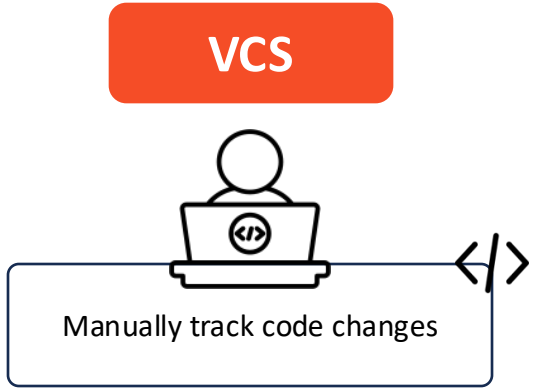
Collaboration Confusion

What is Version Control System?

Time machine for your code

What is Version Control System?

→ **VCS** is a tool that helps you track changes to documents, code, and other files over time



- Slow
- Error-prone
- Data loss

→ Stores different versions of code in a central location

4 W's : Who, What, When & Why

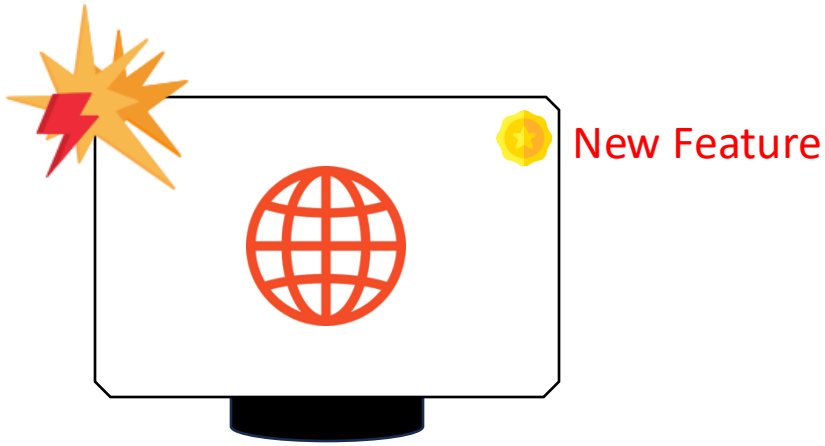
Who made the changes?

What changes were made?

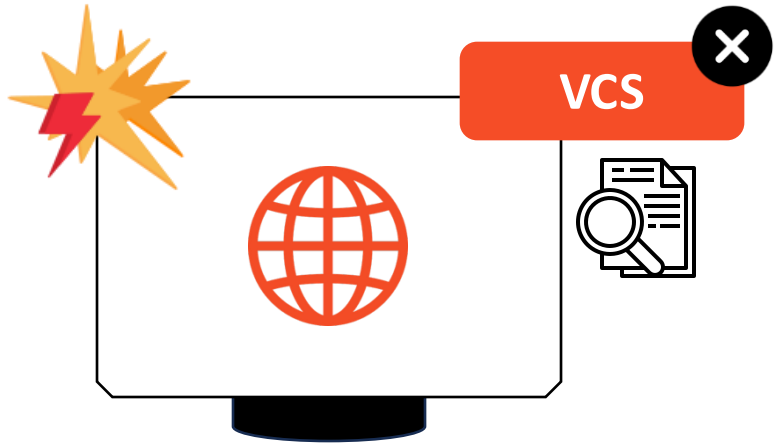
When were the changes made?

Why was the change made?

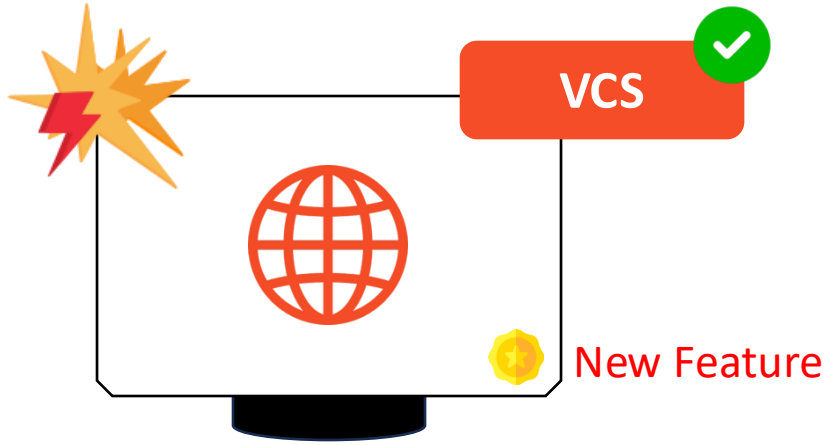
Why do we need VCS?



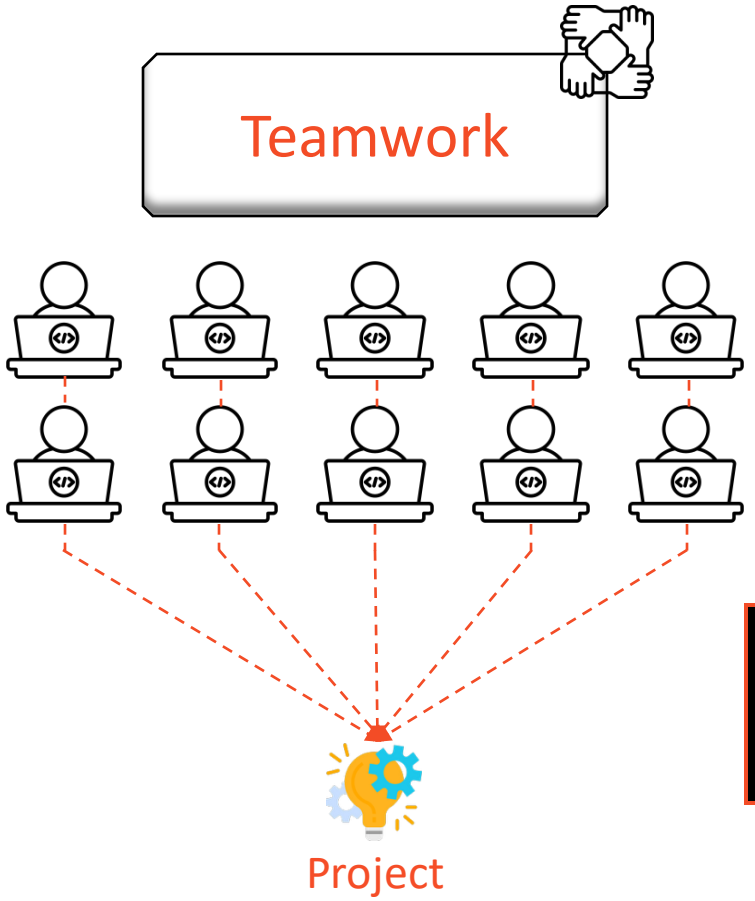
Why do we need VCS?



Why do we need VCS?



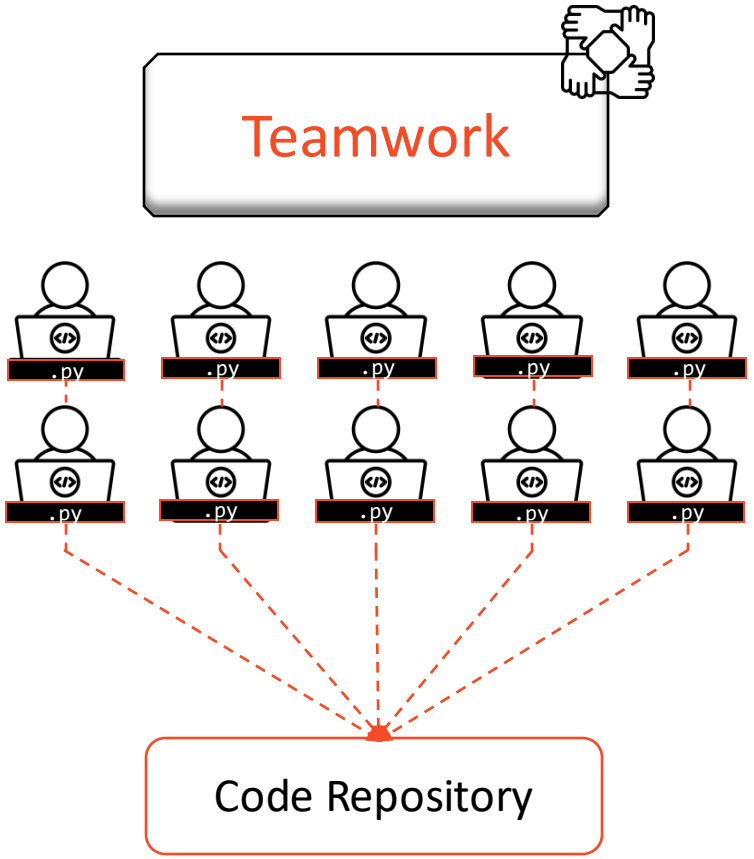
Why do we need VCS?



VCS

```
project_final_v1.py
project_final_v2.py
project_final_reallyfinal.py
project_final_THISONEWORKS.py
```

Why do we need VCS?



VCS

4 W's :
Who, What, When & Why

Benefits of VCS

Benefits of VCS

History of Changes

Parallel Development

Experiment Freely

Accountability

Disaster Recovery

Source Code Management (SCM)?

→ **SCM** is simply how we organize and control our code using a Version Control System

Tool

Version Control System

Practice

Source Code Management

Tool

GitHub or GitLab

- Collaborate
- Create branches
- Review pull requests
- Merge code

Why is Source Code Management Important?

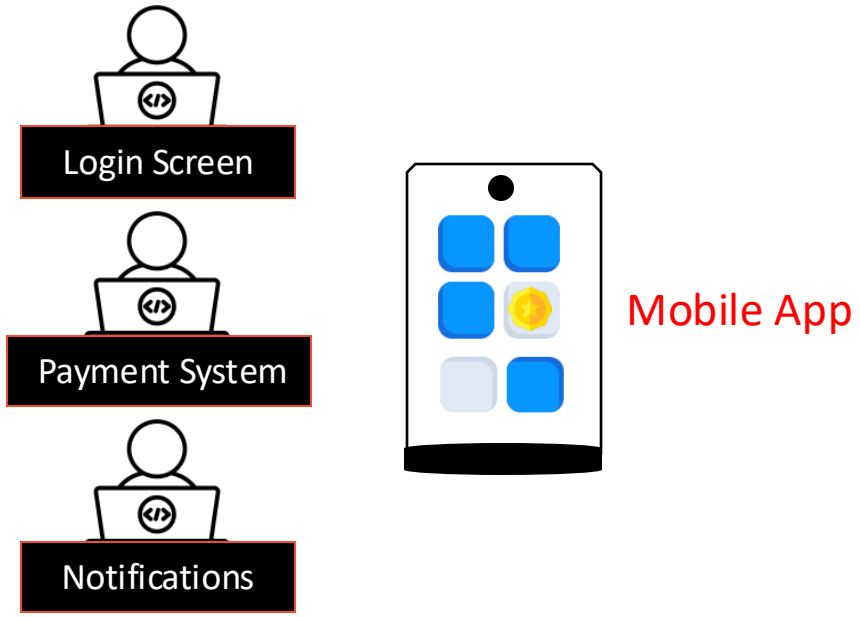
Keeps Code Organized

Improves Teamwork

Provides Faster Development

Quality Control

Why is Source Code Management Important?



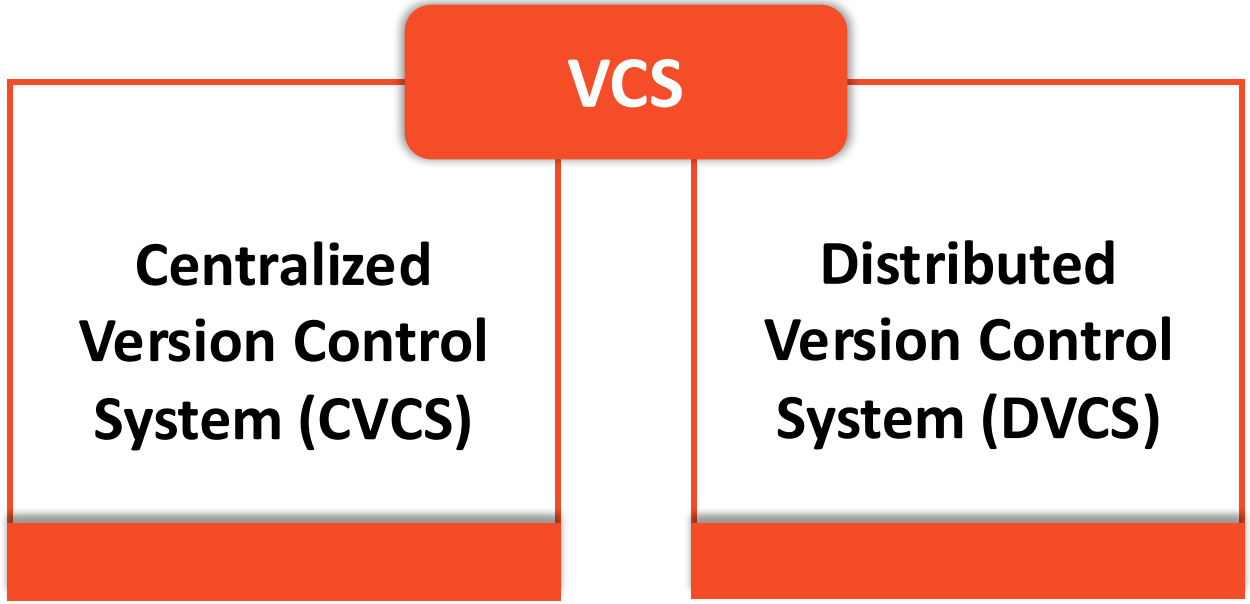
Why is Source Code Management Important?

→ **Version Control System** is the technology that tracks code changes

→ **Source Code Management** is the process of using VCS effectively

Type of Version Control System

Types of Version Control Systems (VCS)



Centralized Version Control System (CVCS)

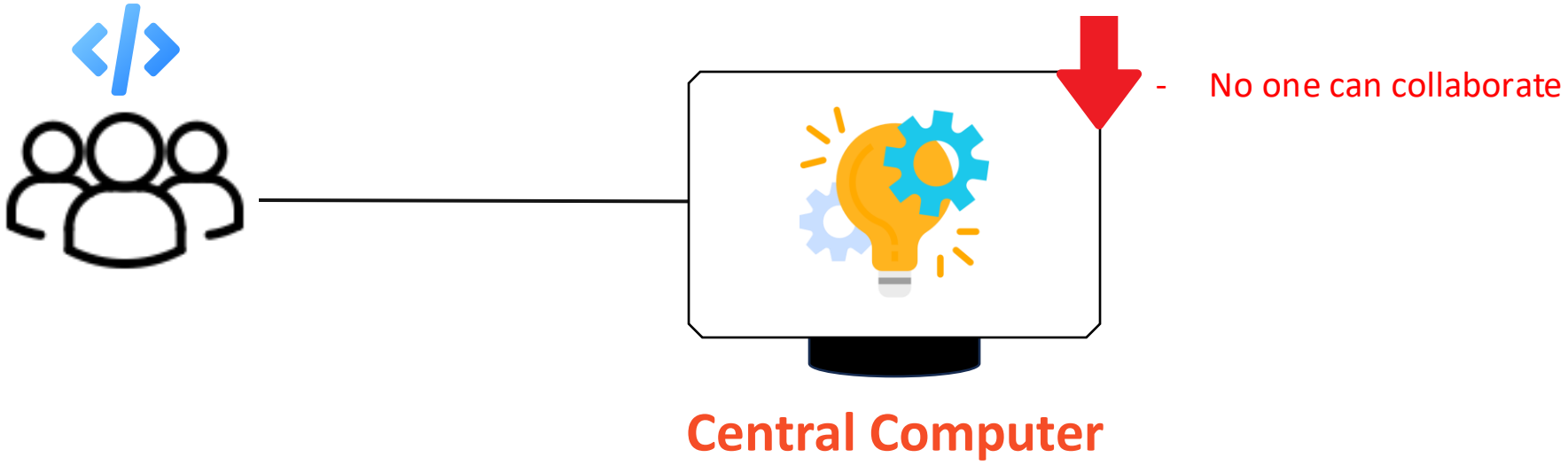
Centralized Version Control System (CVCS)

Single Central Server

```
check out (download)
```

```
commit (upload)
```

Centralized Version Control System (CVCS)



Pros & Cons of CVCS

Pros of CVCS

→ It's simple to understand and use

→ Everyone can see the latest version directly from the server

Cons of CVCS

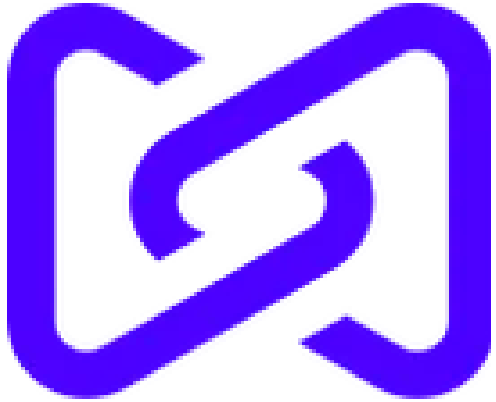
→ If the server crashes, you may lose everything

→ Developers need internet or network connection to work with the code and to track their changes

CVCS Tools

SVN (Apache Subversion)

CVS
(Concurrent Versions System)



Perforce

Distributed Version Control System (DVCS)

Distributed Version Control System (DVCS)

Single Central Server

Distributed Version Control System (DVCS)

Distributed Version Control System (DVCS)

Library

Pros & Cons of DVCS

Pros of DVCS

- → You can work offline since you have the full history locally
- → Even if the main server is down, you can continue your work and keep track of your changes locally and submit your changes to the central server once it's up again
- → Collaboration is much easier and safer

Cons of DVCS

- → Takes more disk space since everyone has a complete copy
- → A bit more complex to learn compared to centralized systems

DVCS Tools

Git

Mercurial

Bazaar

Centralized VCS

Distributed VCS

Code Repositories

- Maintain all project data on a single central server
- Create single point of failure & makes offline work impossible

- ✓ Each developer has a full copy of the repository
- ✓ Full offline functionality & reduce dependency on network connectivity

Distributed VCS

- ✓ Each developer has a full copy of the repository
- ✓ Full offline functionality & reduce dependency on network connectivity
- ✓ Offers faster local operations
- ✓ Sharing changes still requires network access, affecting collaboration speed
- ✓ Provides better fault tolerance due to multiple complete copies of the repository

Introduction to Git

Introduction to Git

Section Overview

- → *Getting Started with Git*
- → *Installing Git on Windows, macOS and Linux*
- → *Configuring Git on Linux*
- → *Git Architecture & Key Concepts*
- → *Understanding .git Folder & git init command*
- → ***Demonstration:** .git Folder & git init command*

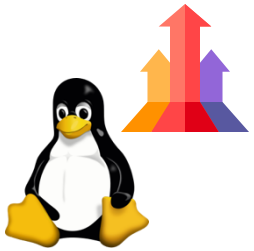
Getting Started with Git

What is Git?

Global Information Tracker

Open-source tool that has revolutionized how developers manage their code

History of Git?



Grown Enormously

Distributed version control system

Linus Torvalds

2005

History of Git?

Designed to overcome these challenges and manage large-scale projects more effectively

History of Git?

Linus Torvalds created Git to handle large-scale, decentralized software development, aiming for speed, efficiency, and robustness to support projects like the Linux kernel.

Alternative Version Control Systems

Git alternatives

Subversion

- Single central repository
- Older than Git
- Lacks Git's distributed capabilities

Mercurial

- Scalability and performance
- Uses different command syntax

Perforce

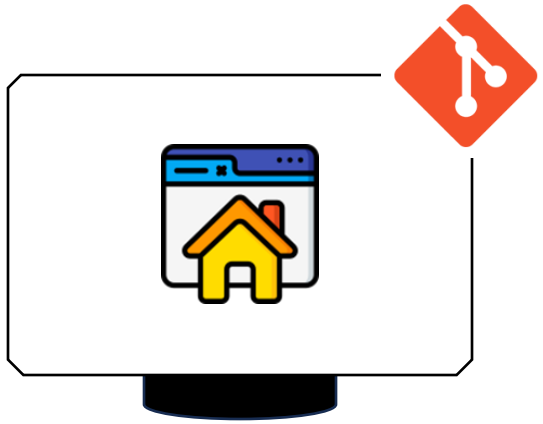
- Large-scale commercial projects
- Emphasizing performance and scalability

Why Git?

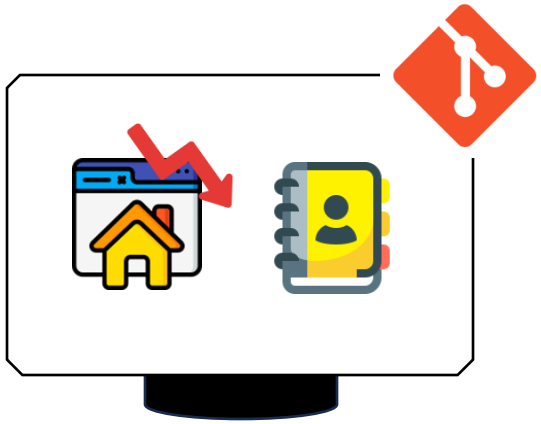
Real-World Benefits for Developers

- **Track Changes Over Time**
- **Work Safely Without Fear of Breaking Things**
- **Collaboration Made Easy**
- **Git Works Offline Too**
- **Git Is the Industry Standard**

Day-1

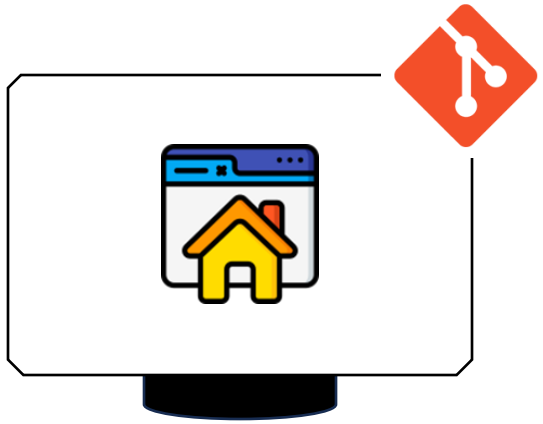


Day-2

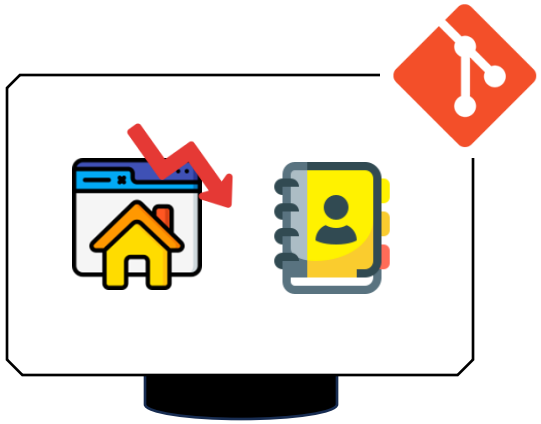


Rollback to find when the bug was introduced

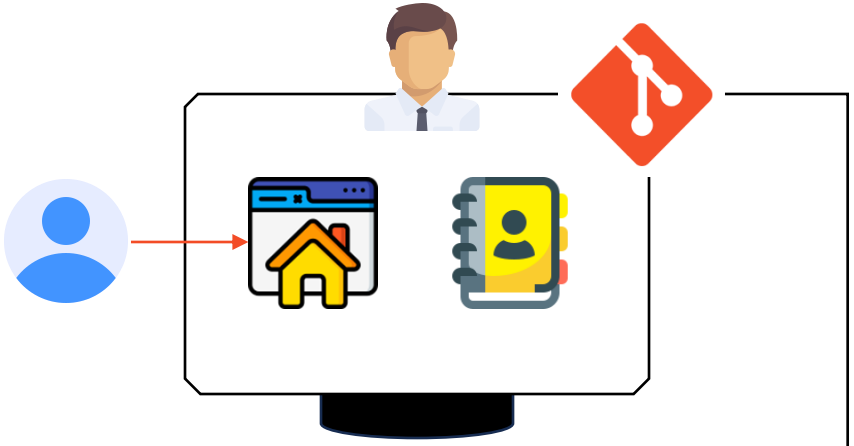
Day-1



Day-2

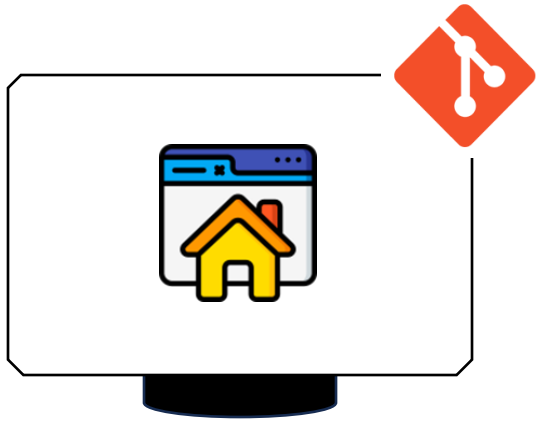


Day-3

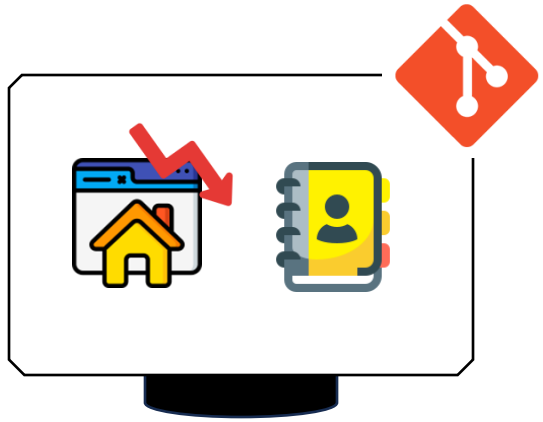


Git lets both of you work independently and then combine everything smoothly

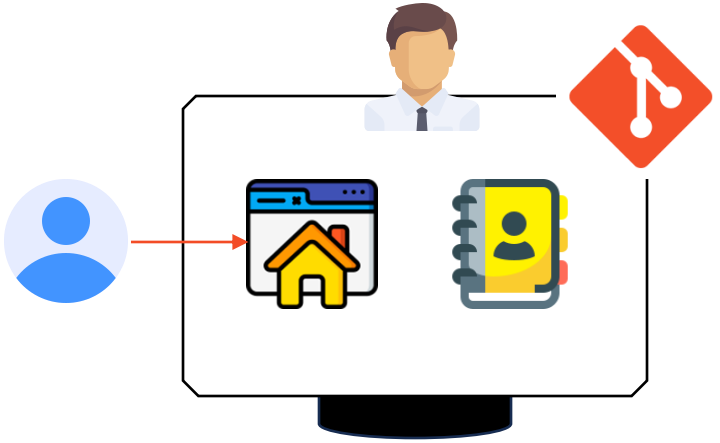
Day-1



Day-2



Day-3



Saves you time

Avoids conflicts

Safe project history

Git Architecture & Key Concepts

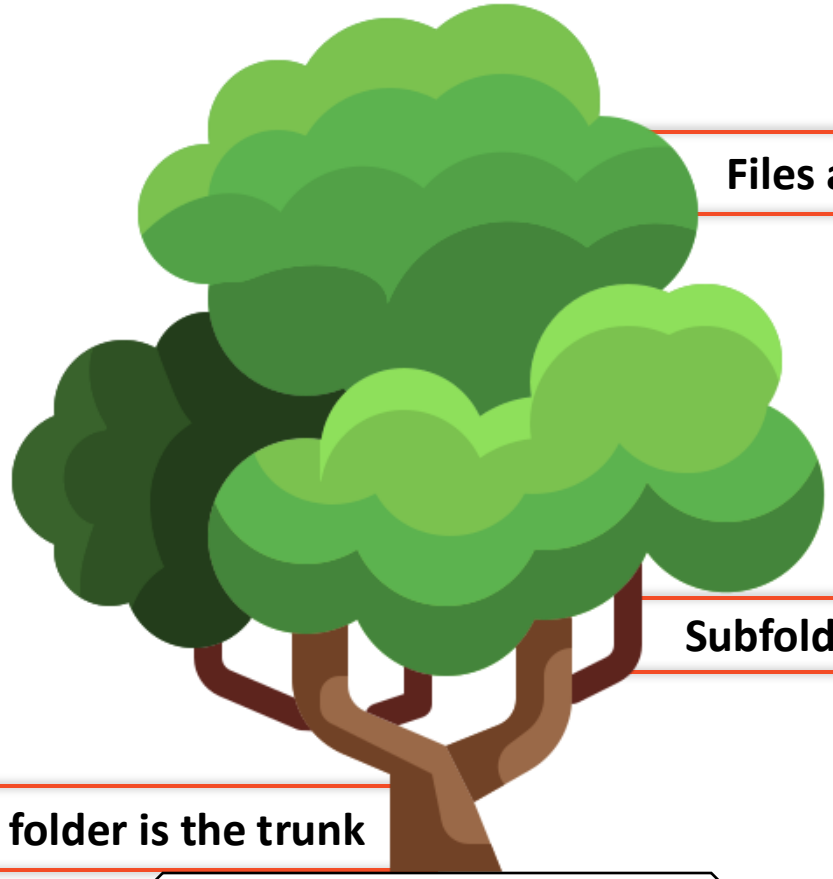
Why Understanding Git Architecture Matters

Better Troubleshooting

Better Workflow Decisions

Essential Terminology

Tree



Files are the leaves

Subfolders are branches

The root folder is the trunk

Tree Data Structure

Essential Terminology

Tree

Tree Data Structure

For Example

The Root

The Branches

The leaves

```
my-app/
├── src/
│   ├── app.py
│   └── utils.js
├── tests/
│   └── app.test.js
└── README.md
```

Essential Terminology

Essential Terminology

Repository Tree

Permanent Storage

- Check it out
- Edit it
- Commit it back

Library

Two-Tree to Three-Tree Architecture

The Old Way: Two-Tree Architecture

Working Directory

Where you make changes

Repository

Where changes are permanently stored

Two-Tree to Three-Tree Architecture

The Old Way: Two-Tree Architecture

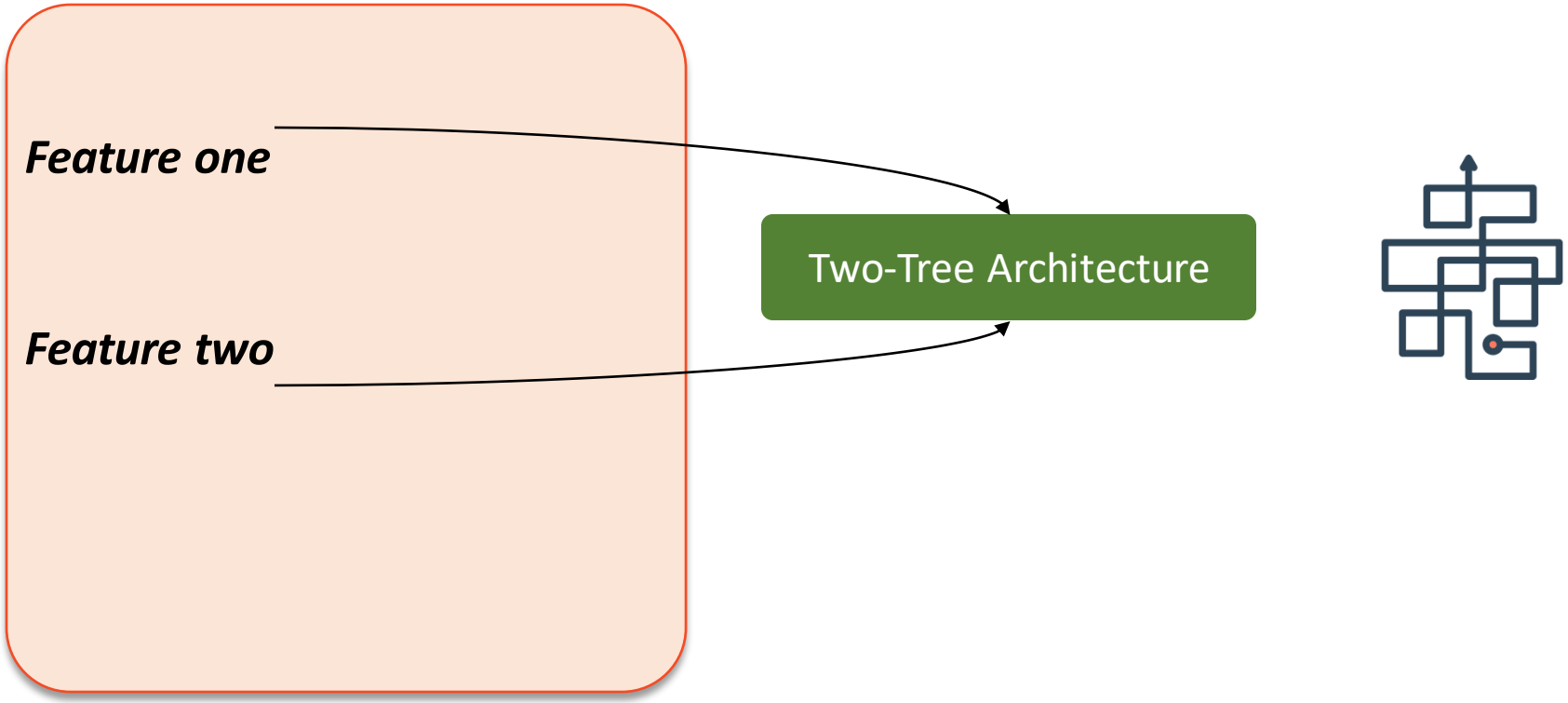
Working Directory

Repository

Problem

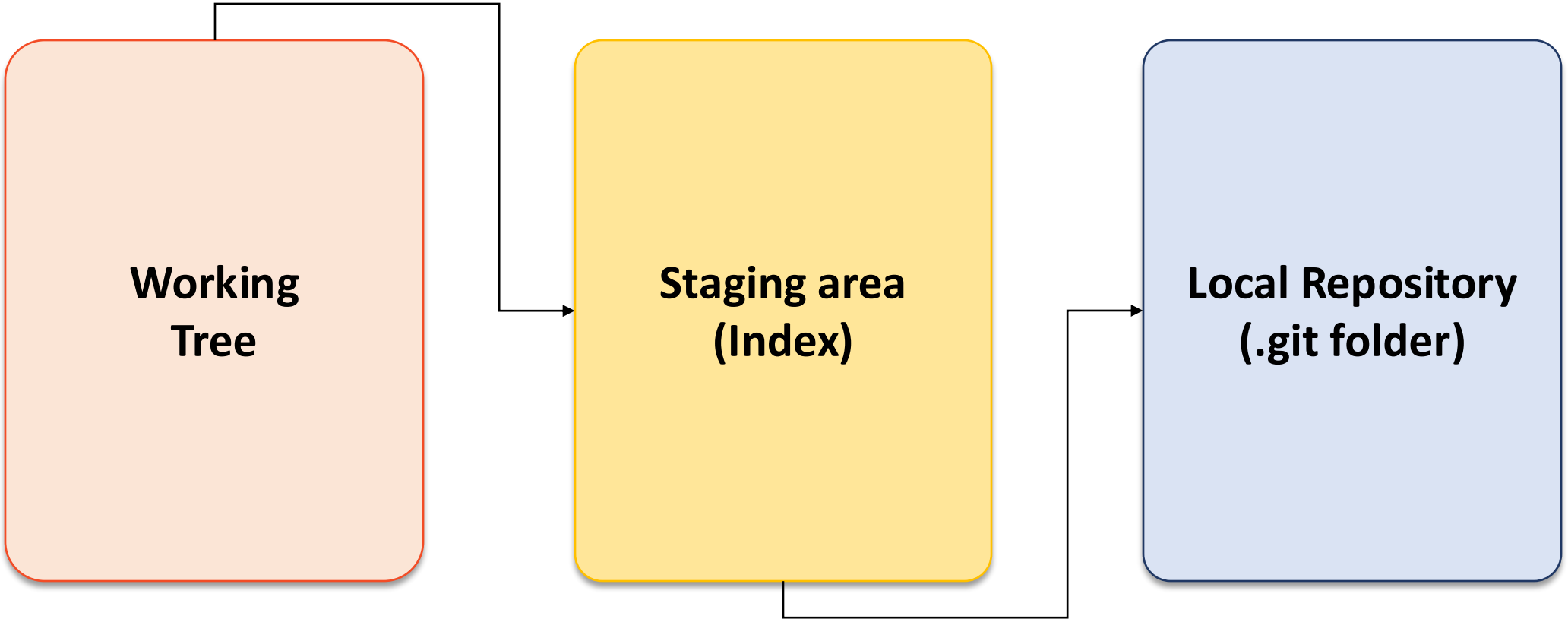
- Changes move directly from your workspace to permanent storage
- No way to review changes before committing
- Difficult to create clean, logical commits
- Hard to partially commit files

Two-Tree to Three-Tree Architecture



Git's Innovation: Three-Tree Architecture

Git's Three-Stage Workflow



Git's Three-Stage Workflow

Stage-1 **Working Directory**

- Create
- Edit
- Delete Files

```
import pandas as pd

data = {
  "calories": [420, 380, 390]
}

#load data into a DataFrame
object:
df = pd.DataFrame(data)

print(df)
```

app.py

Working Directory

Untracked

Git's Three-Stage Workflow

Stage-2 **Staging Area (Index)**

→ Allows you to choose exactly which changes you want to include

→ Move changes here with the ***git add*** command

Browse the store (working directory)

Add items to cart (staging area)

Checkout (commit)

Git's Three-Stage Workflow

Stage-3 **Local Repository (.git folder)**

→ Git’s internal database where all your committed changes live

→ Running ***git commit*** saves “staged changes” permanently in the local repository

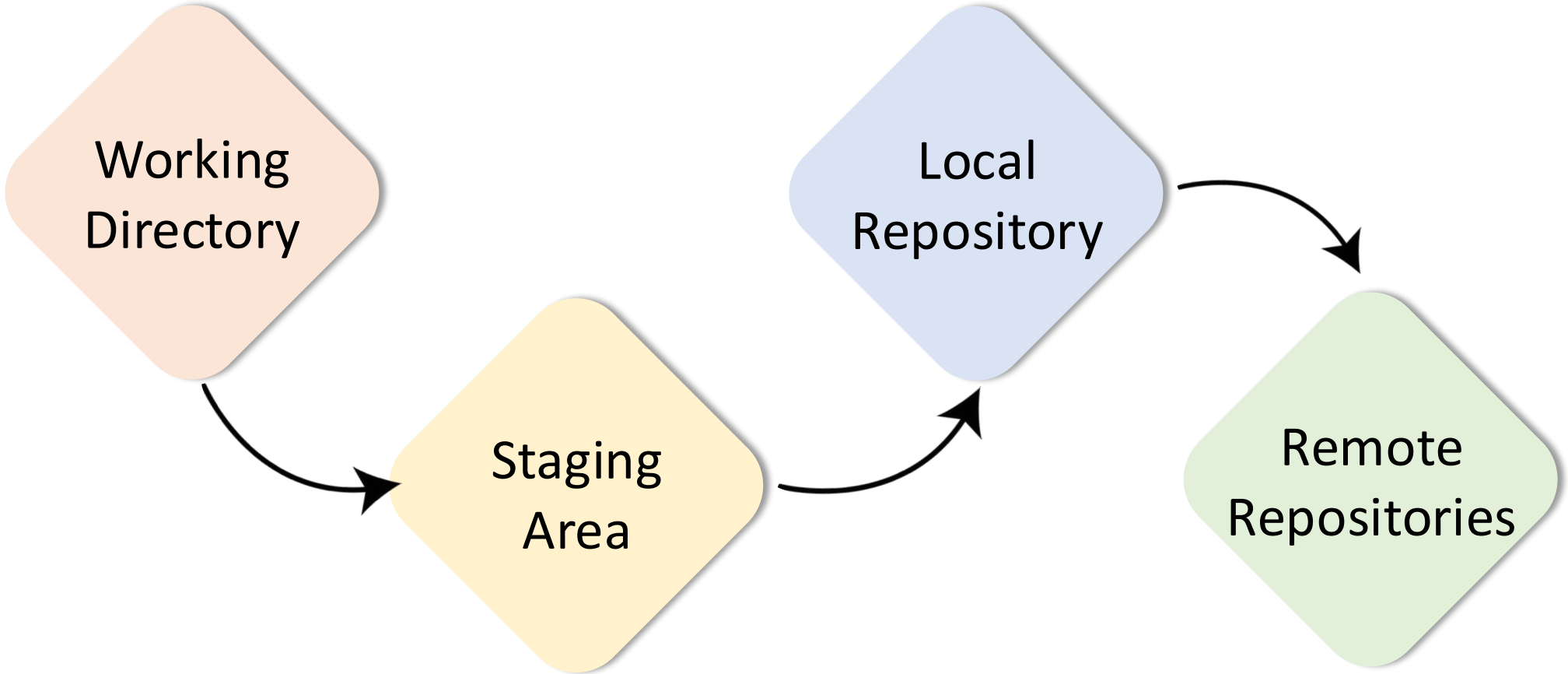
→ This forms the official version history of your project, stored inside the hidden **.git folder** on your machine

Remote Repositories

GitHub

GitLab

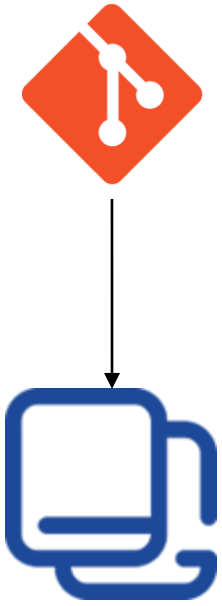
Complete Workflow with Remote Repository



Login Feature

Working Directory	Staging Area	Local Repository	Remote Repositories
`Edit login.py file`	`git add login.py`	`git commit -m "Implement user login"`	`git push origin main`

How Git Stores Data



Doesn’t save differences or deltas line by line

Snapshot

How Git Stores Data

If a file hasn't changed, Git doesn't store it again,
instead, it points the version that's already stored

Snapshot

Every snapshot is stored in a compressed form, identified by a unique **SHA-1 hash**

```
e4d1f9c8a2b3c6e7f1234567890abcdeffedcba9
```

How Git Stores Data

- Fast
- Reliable

→ Any change in a file produces a new hash, so Git can instantly detect tampering

→ You can always verify the integrity of your project history

Git Objects

Blob
(Binary Large Object)

→ Represents the content of a single file

→ It doesn’t store filename, only file content

Git Objects

Tree

→ Represents the directory

→ The tree object can point to multiple blobs (files) or other trees (subfolders)

Git Objects

Commit

→ Represents a snapshot of the project at a certain time

→ A commit points to a tree (the root directory), and it stores metadata like author, timestamp, message, and even parent commits

Tree Object + Author Info + Commit Message + Parent Commit Reference

HEAD and Branches

Git

Head

→ Pointer that usually points to your current branch

→ That branch is itself just a pointer to a commit

→ When you create a new commit, the branch pointer moves forward, and HEAD moves along with it

HEAD and Branches

If **HEAD** points directly to a commit instead of a branch, that's called a detached **HEAD** state

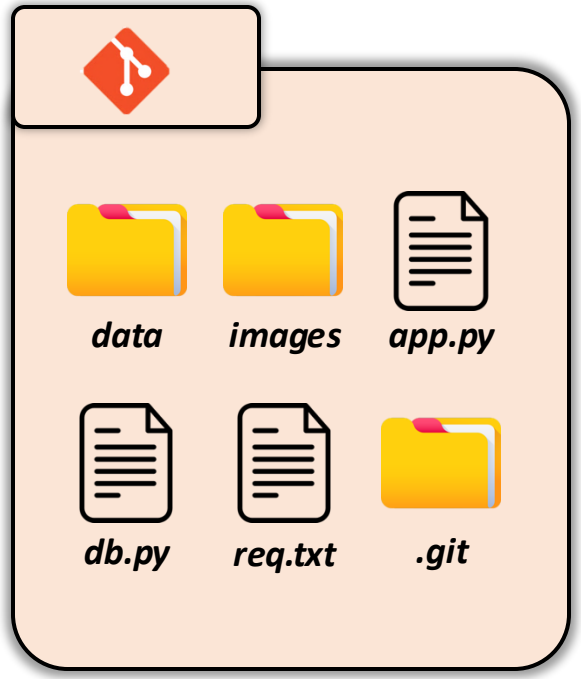
Means you're exploring a past commit, not working on a branch

Branches themselves are just lightweight pointers, nothing more

Branches in Git is cheap & instant compared to other systems

Understanding .git Folder & git init command

.git Folder



.git Folder

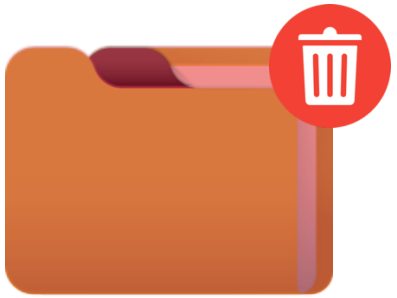
.git

- Commits History
- Branches
- Configuration
- Staging Area

House

Control Room

.git Folder



.git

Project will lose its version history &
Git will stop tracking changes

.git Folder

.git

```
HEAD
Config
index
objects/
refs/
logs/
hooks/
info/
```

Points to the current branch or commit you're on

Stores repository-specific configuration

Represents the staging area where changes wait before committing

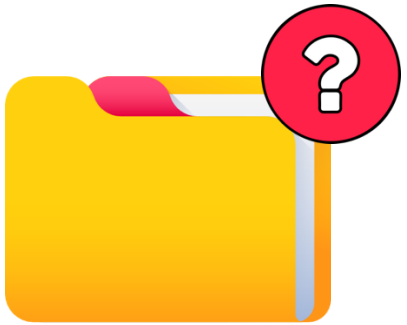
Contains all Git data (blobs, trees, commits) identified by SHA-1 hashes

Stores references to branches, tags, and remote pointers

Records movements of HEAD and branch updates

Contains scripts that run automatically on Git events

Holds additional exclude rules not in .gitignore file



.git

```
git init
```

.git

```
git init
```

- Turn folder into Git repository
- Create a hidden **.git** folder inside it

Git Repository

.git

Git stores all the history, branches, and configuration of your project

```
git init
```

Creates **.git** folder and turns your normal folder into a Git repository

Working with Files in Git

Working with Files in Git

Section Overview

- → *Tracking & Untracking Files*
- → ***Demonstration:*** *git status, git add, git commit, and git log*
- → *Understanding .gitignore file*
- → ***Demonstration:*** *git rm, git mv, and git diff*
- → *Undoing Changes in Git*

Tracking & Untracking Files with Git

Tracking & Untracking Files with Git

Untracked	New file that Git doesn’t know about yet
Modified	File that’s already tracked, but you’ve made changes that are not staged yet
Staged	File whose changes are marked and ready to be included in the next commit
Committed	File that is safely stored in Git’s history, inside the .git Folder

Tracking & Untracking Files with Git

Untracked **Modified** **Staged** **Committed**

Demo

Tracking & Untracking Files : git status, git add, git commit & git log

Tracking & Untracking Files with Git

Check status	`git status`
Stage Changes	`git add`
Commit Changes	`git commit`
View History	`git log`

Understanding .gitignore file

Understanding .gitignore file

- → Temporary files like ***.log***
- → Build directories like ***dist/*** or ***target/***
- → OS-specific files like ***.DS_Store*** (macOS) or ***Thumbs.db*** (Windows)
- → Virtual environment or ***.env*** files

Understanding .gitignore file

`.gitignore`

Plain text file

which files & directories
should be ignored by Git

Understanding .gitignore file

`.gitignore`

Functions as a filter for Git's tracking system, allowing developers to exclude non-essential, sensitive, or generated files

Common Patterns

- → ***.log***, ignore all log files
- → ***node_modules/,*** ignore the entire folder
- → ***.tmp***, ignore temporary files
- → ***!important.log*** - exclamation mark means **“do not ignore this file, even if it matches a rule”**

Demo

Understanding .gitignore file

Demo

Working with Files : git rm, git mv & git diff

```
git rm
```
→ to remove files

```
git mv
```
→ to rename or move files & folders

```
git diff
```
→ to view changes

Demo

Undoing Changes: git restore and git commit --amend

Branching & Merging

Branching & Merging

Section Overview

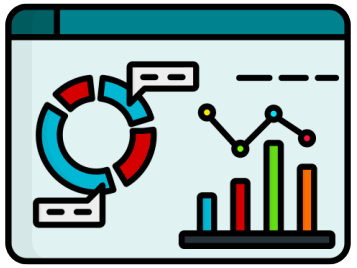
- → *Introduction to branches*
- → *Branch Operations*
- → *Merging branches and combining changes*
- → ***Demonstration:** Merging branches*
- → *Handling and resolving merge conflicts*
- → *Undoing changes with **git reset** and **git revert***

Understanding Branches

What is a Branch in Git?

Branch

without affecting the main codebase

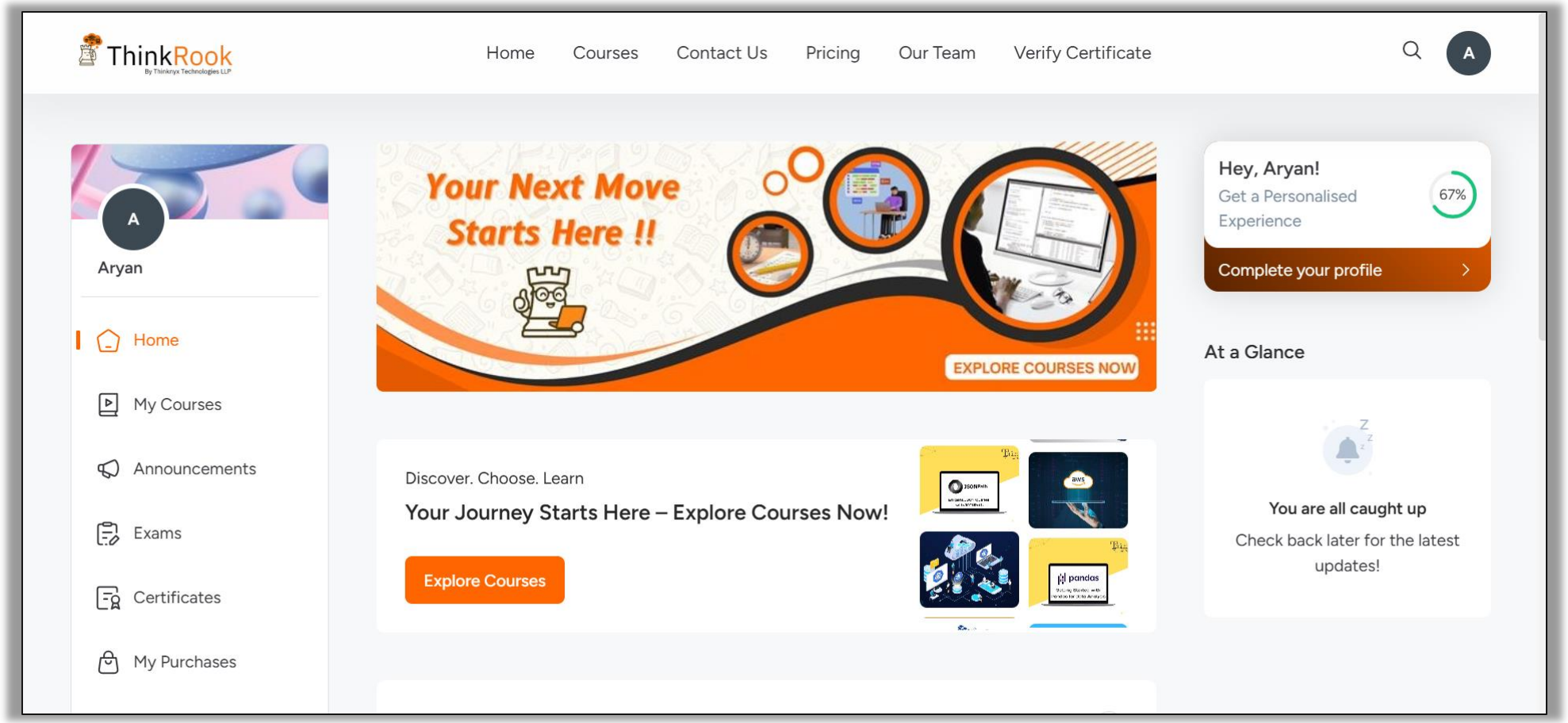


- ✓ New features
- ✓ Bug fixes
- ✓ Experiments

main/master Branch

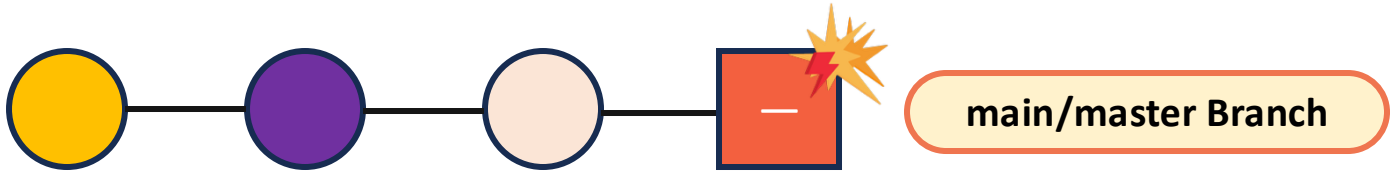
ThinkRook

By Thinknyx Technologies LLP



Dark Mode

Without Branching

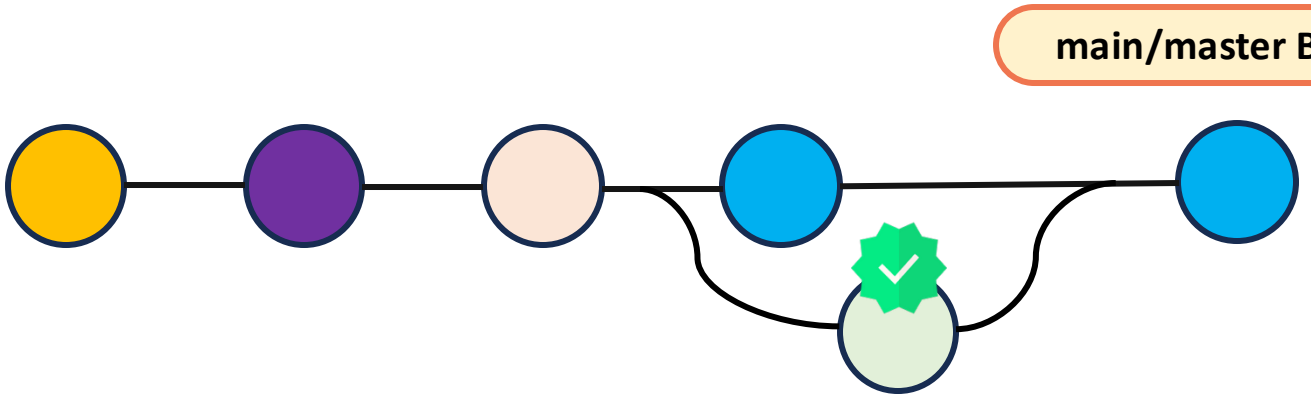


→ Edit the main code directly

→ If something breaks, the live site could go down

→ Introducing bugs before the feature is ready

With Branching



- ✓ Design tweaks
- ✓ Colour schemes

→ Create a new branch called **dark-mode-feature**

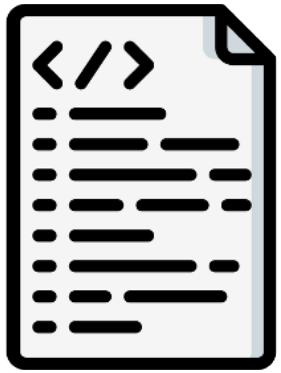
→ Make all our changes in dark-mode-feature branch

→ Test everything in this branch

→ Once verified, we **merge** it back into the main branch

Why we need Branches

Isolation of Work

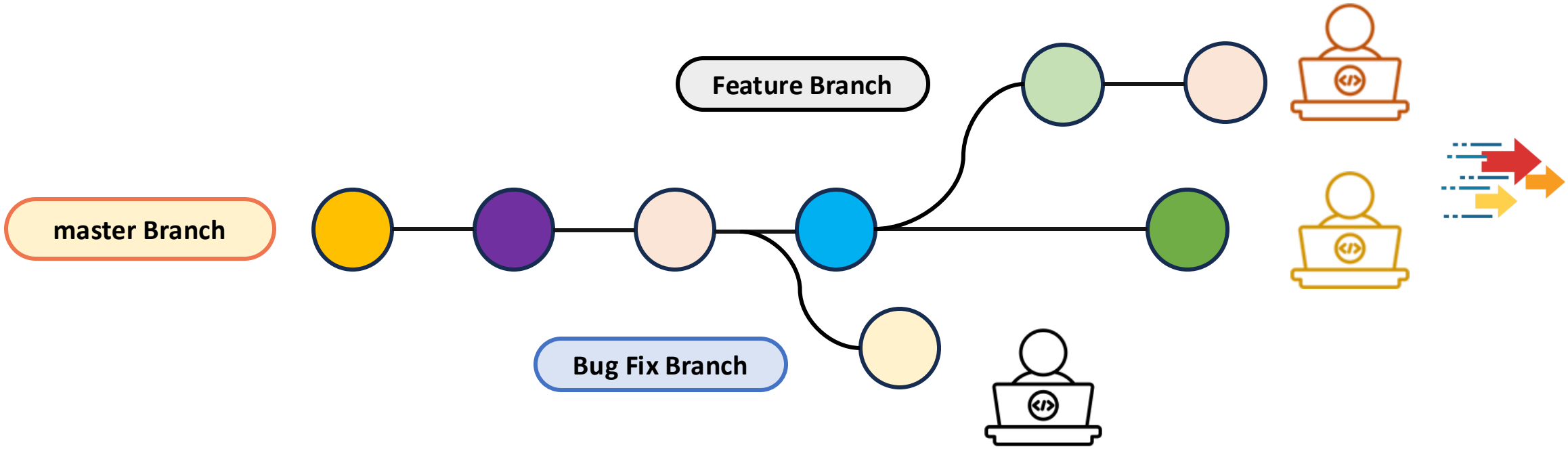


→ Changes won’t affect the main codebase until the code is ready

private workshop where we can build without worrying about breaking anything

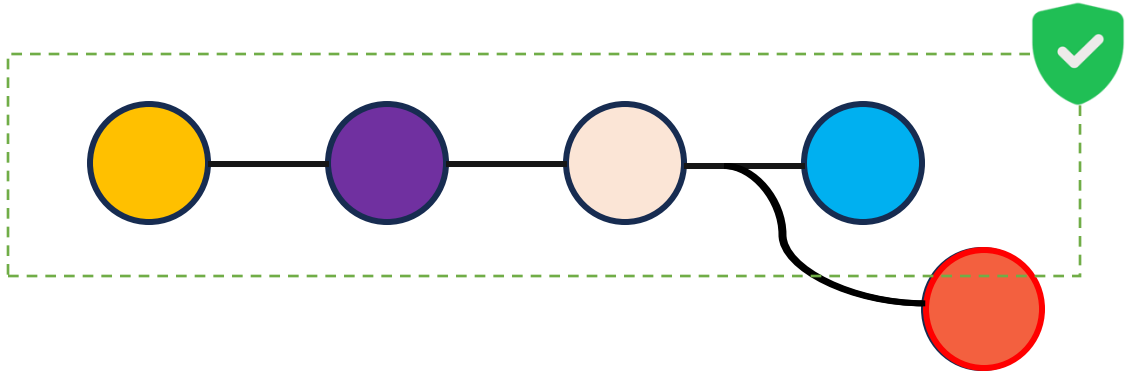
Why we need Branches

Parallel Development



Why we need Branches

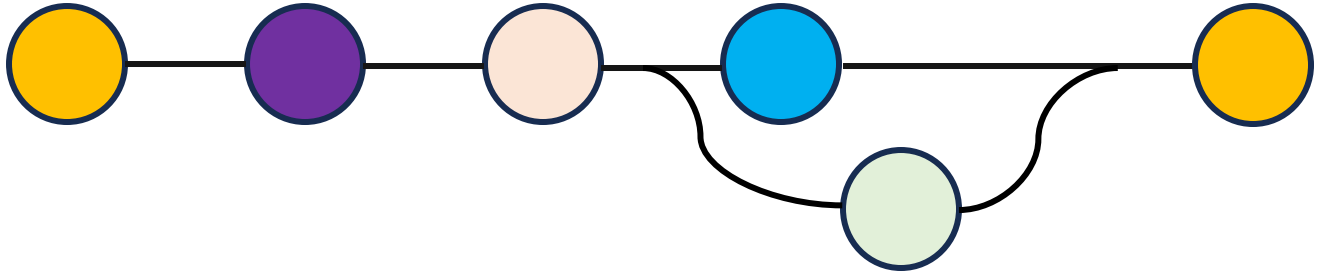
Safe Experimentation



Let us try out our ideas without consequences

Why we need Branches

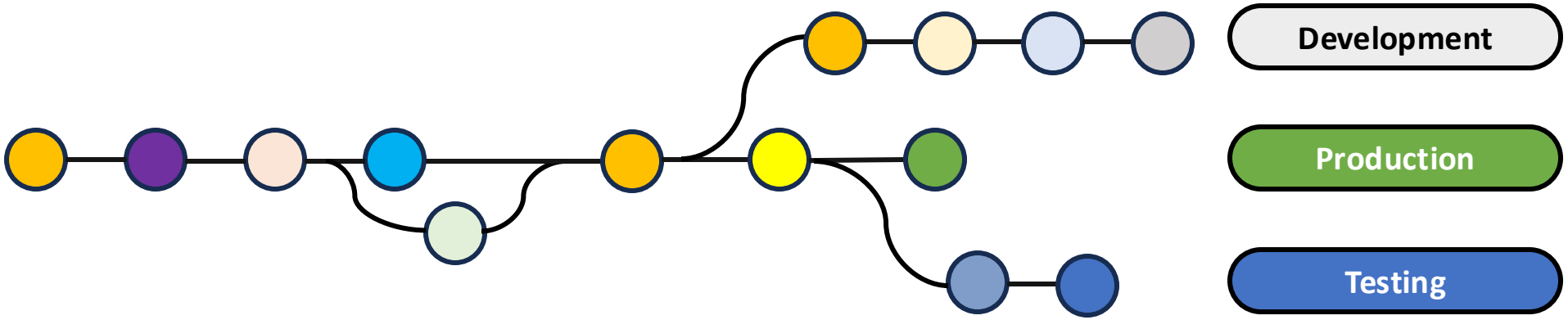
Code Review & Collaboration



- ✓ Catch bugs
- ✓ Improve code quality
- ✓ Ensure everyone agrees on the changes

Why we need Branches

Release Management

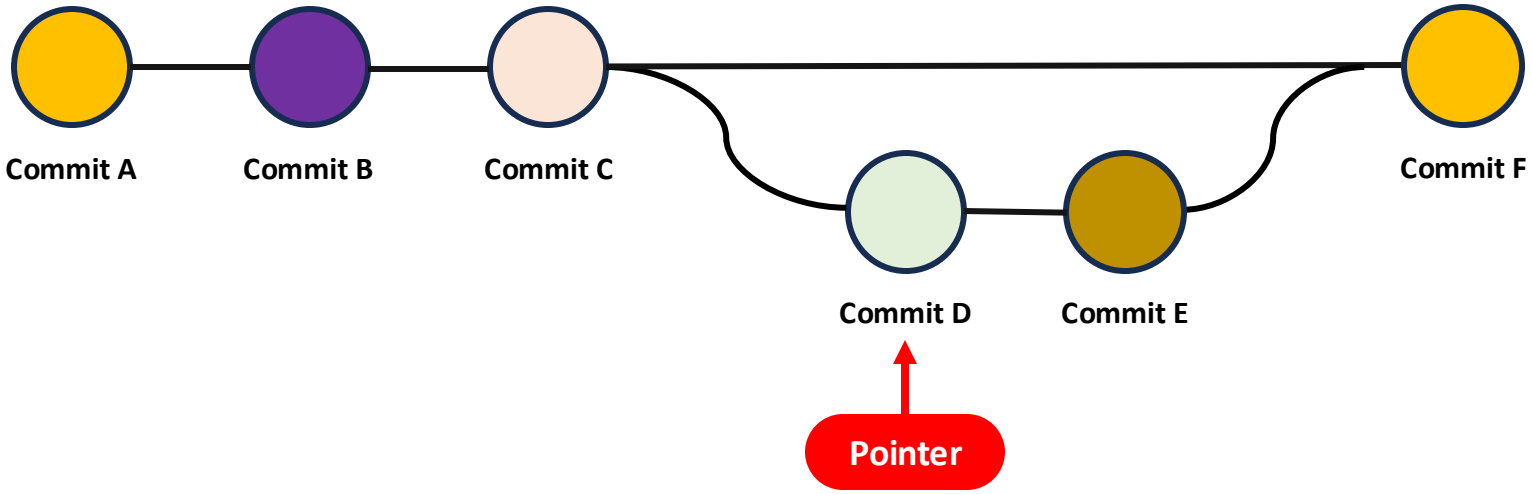


Makes easy to release stable versions
while building and testing new features in the background

How Branches Work Internally

→ Every commit in Git is like a snapshot of our project

→ Makes branches extremely lightweight in Git



HEAD Pointer

HEAD

```
root@thinkrook:~/git-demo# git log --oneline
80f0268 (HEAD -> master) updated notes.txt file
05e2c6d Moved log files to logs folder
50c6dd1 renamed notes1.txt
ea1d636 Removed notes2.txt
```

HEAD

- ✓ Tells Git where you are right now
- ✓ Points to the branch you are working on which was master in our case
- ✓ Points to the latest commit

```
root@thinkrook:~/git-demo# git log --oneline
80f0268 (HEAD -> master) updated notes.txt file
05e2c6d Moved log files to logs folder
50c6dd1 renamed notes1.txt
ea1d636 Removed notes2.txt
```

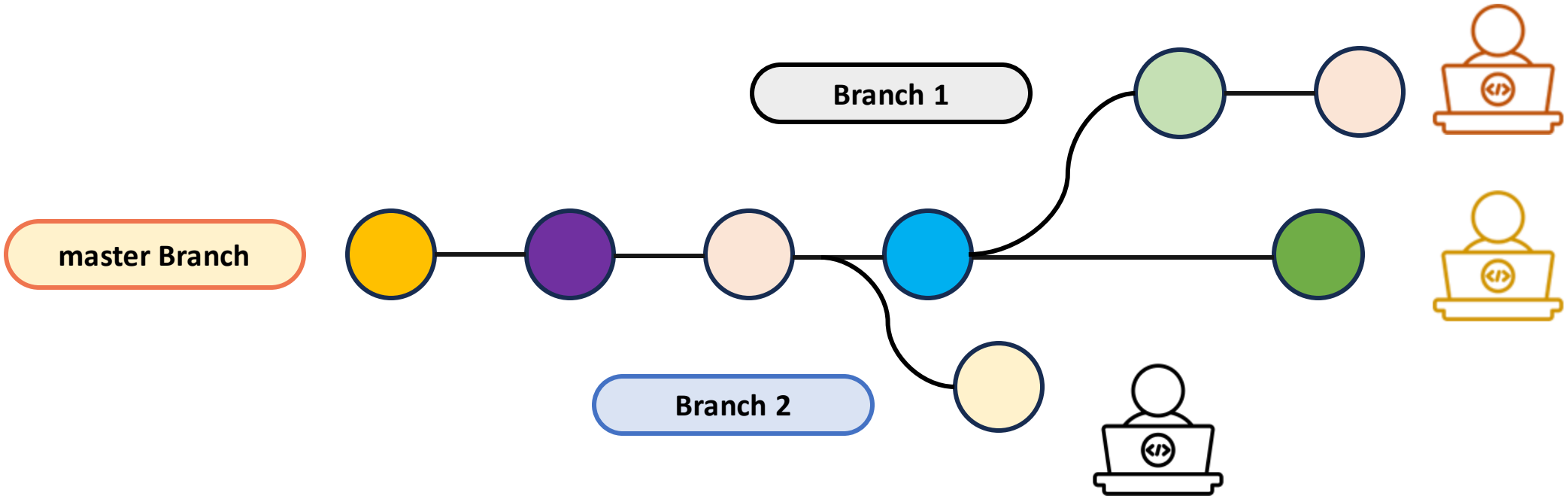
HEAD

→ Which commit your working directory is based on

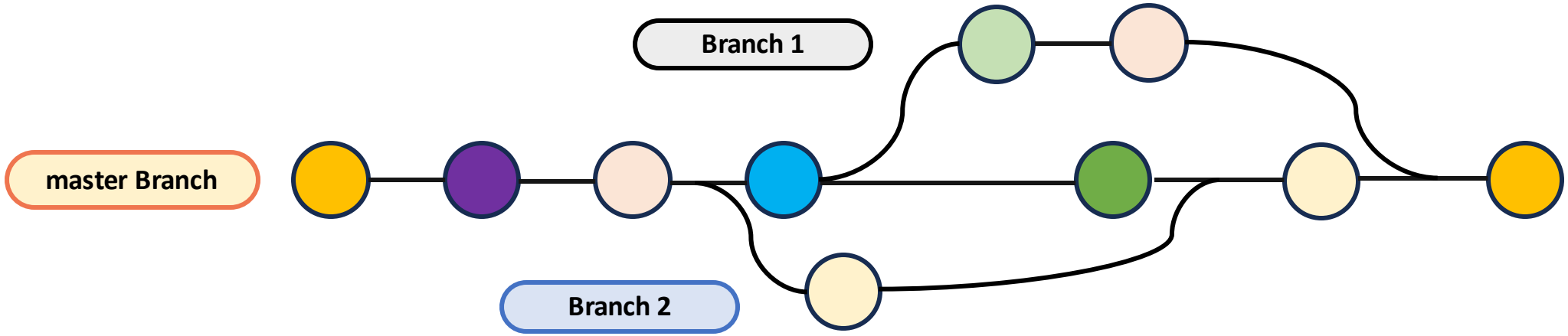
→ Where your next commit will be attached

Merging Branches in Git

Merging Branches in Git



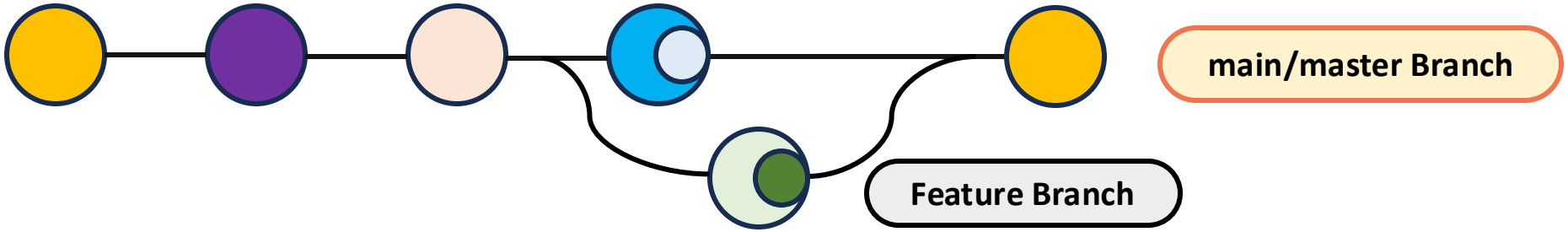
Merging Branches in Git

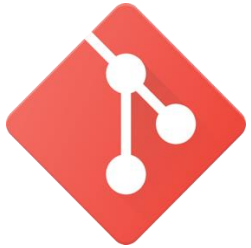


- ✓ Bring together different code changes
- ✓ Resolve conflicts
- ✓ Maintain a unified codebase

- ○ Essential for collaboration
- ○ Integration of new features
- ○ Bug fixes or improvements

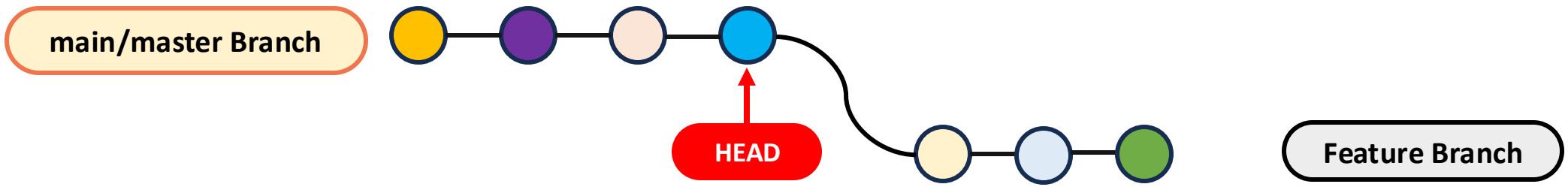
Merging Branches in Git



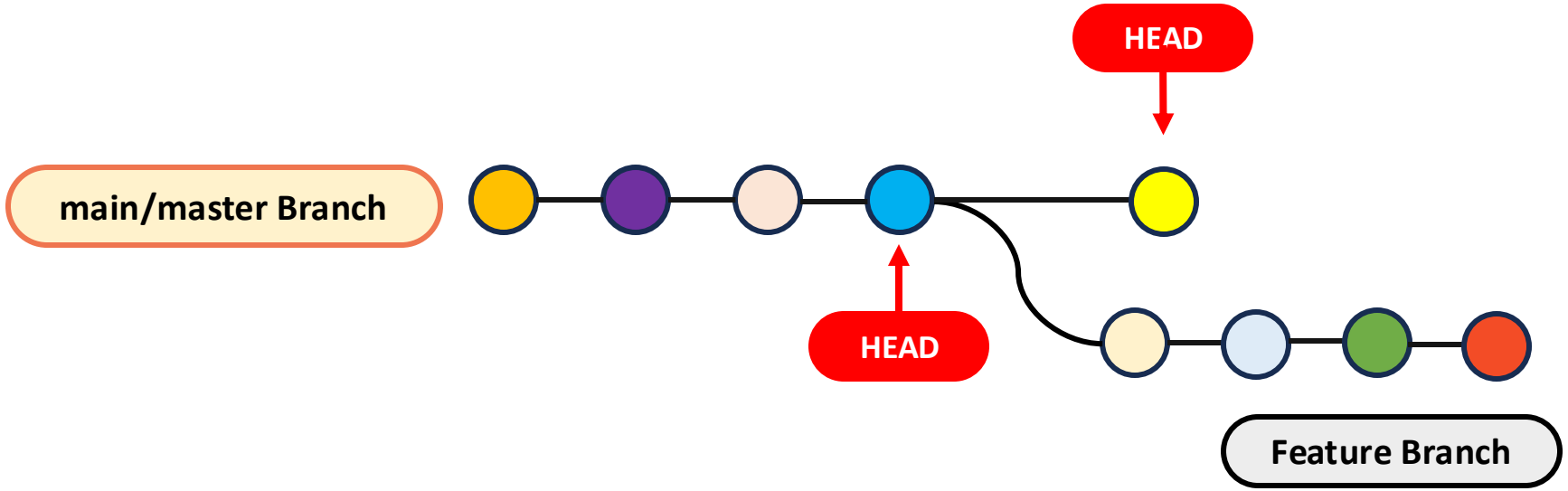


Fast-forward Merge

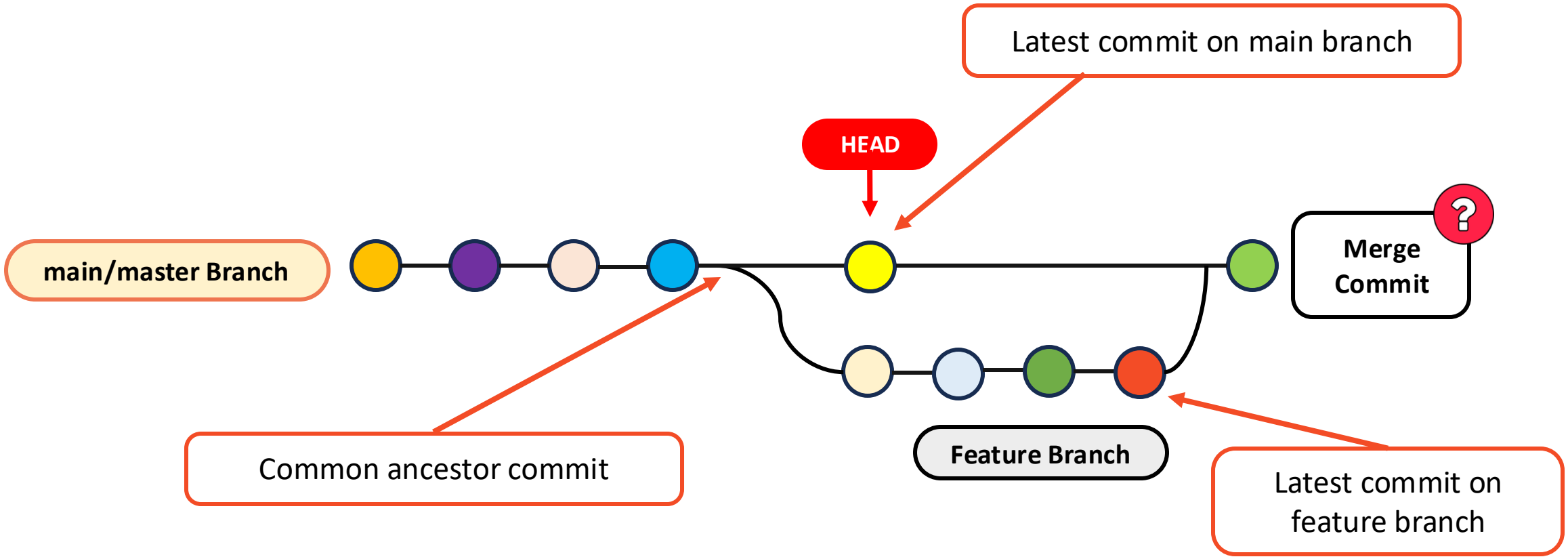
→ Simplest type of merge



Three-way Merge

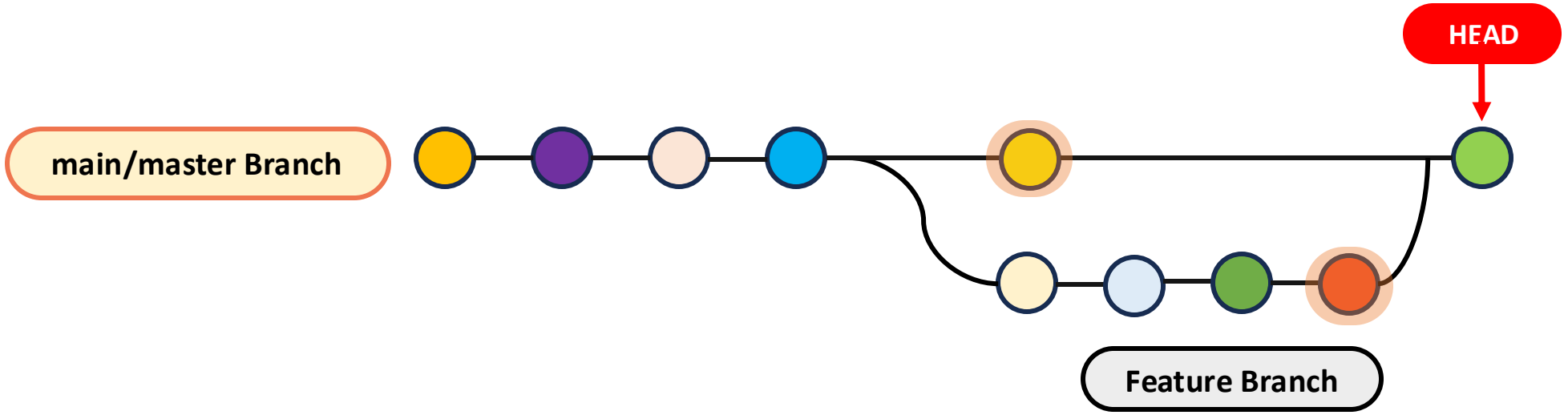


Three-way Merge

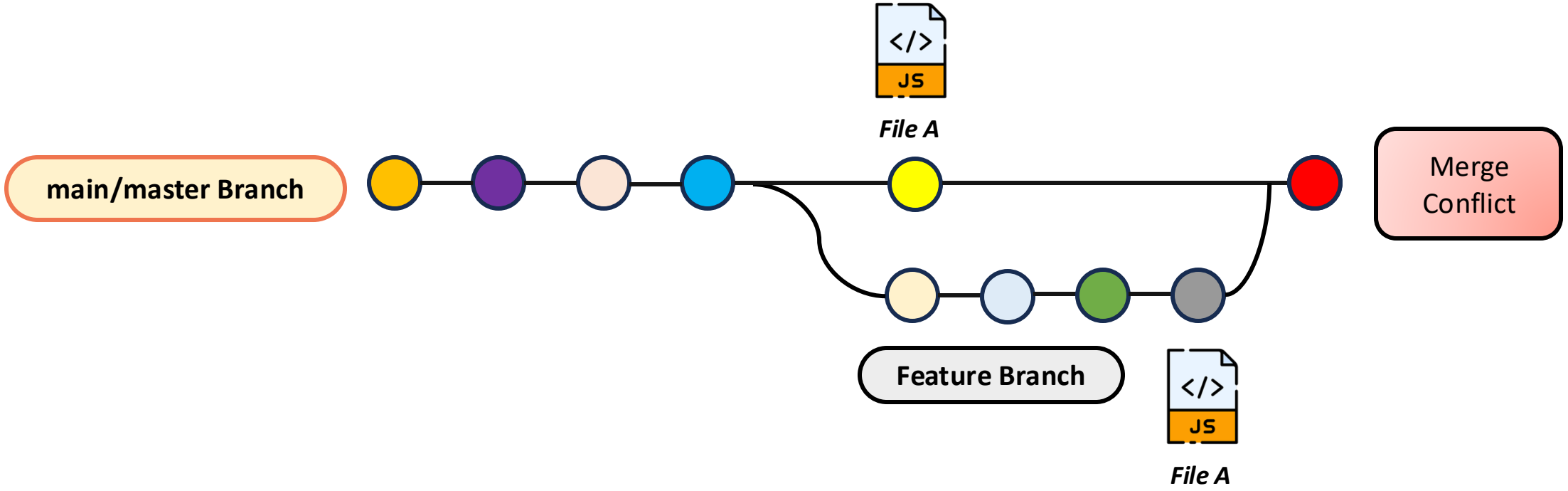


Merge Commit

Special kind of commit that has **two parents** instead of one



preserves history of both branches & shows clearly where they joined



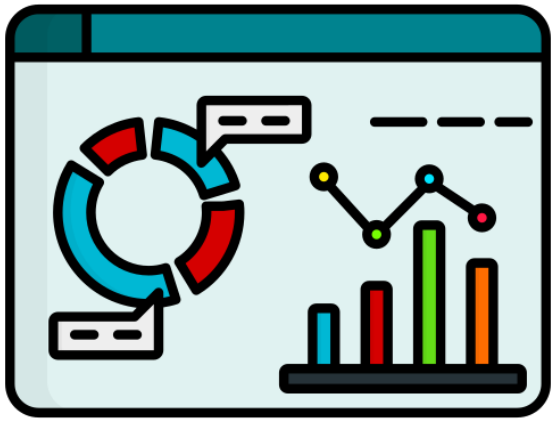
Fast-forward Merge

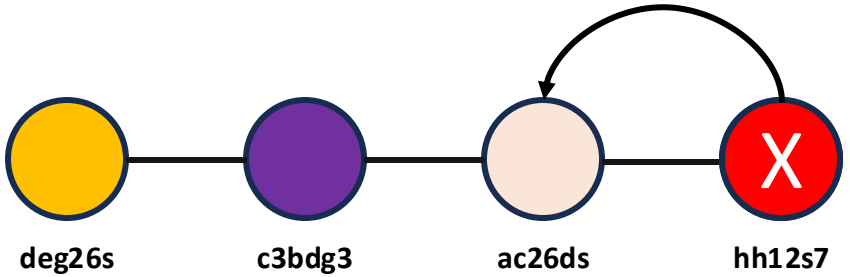
Three-way Merge

Helps you to keep track of your project's history & avoid confusion when collaborating with others

Undoing changes:
git reset** and **git revert

Undoing Changes





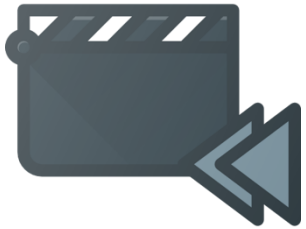
git reset

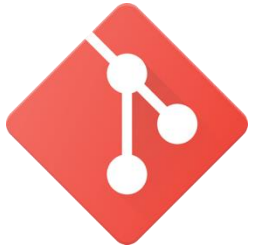
git revert

git reset

Moves your current branch pointer backwards in time

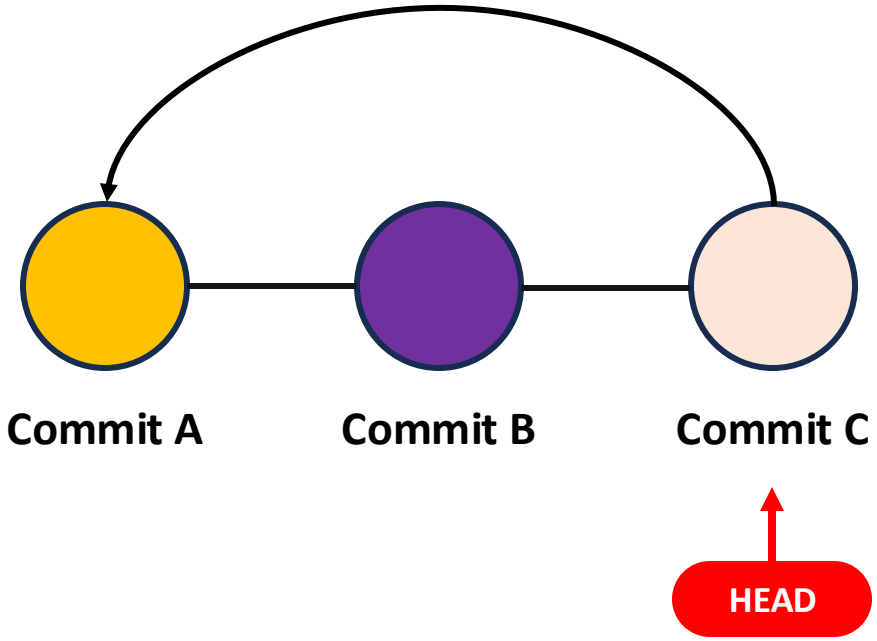
rewinding a movie



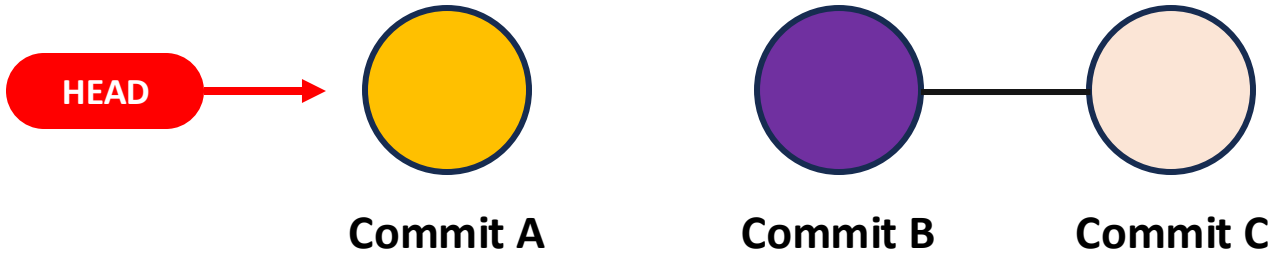


Forget about these recent commits, we want to go back to this earlier one

git reset

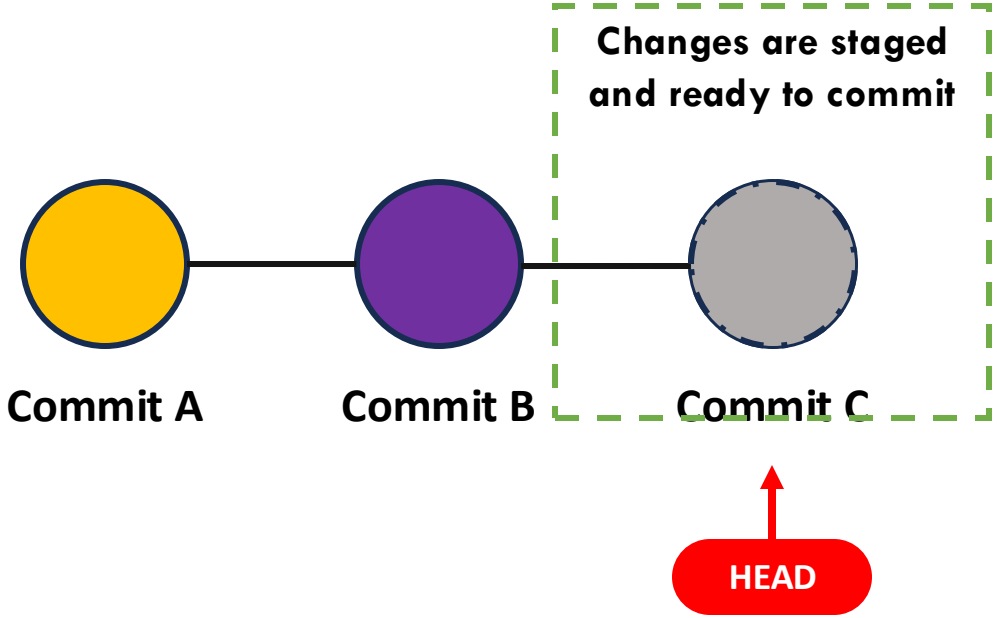


git reset



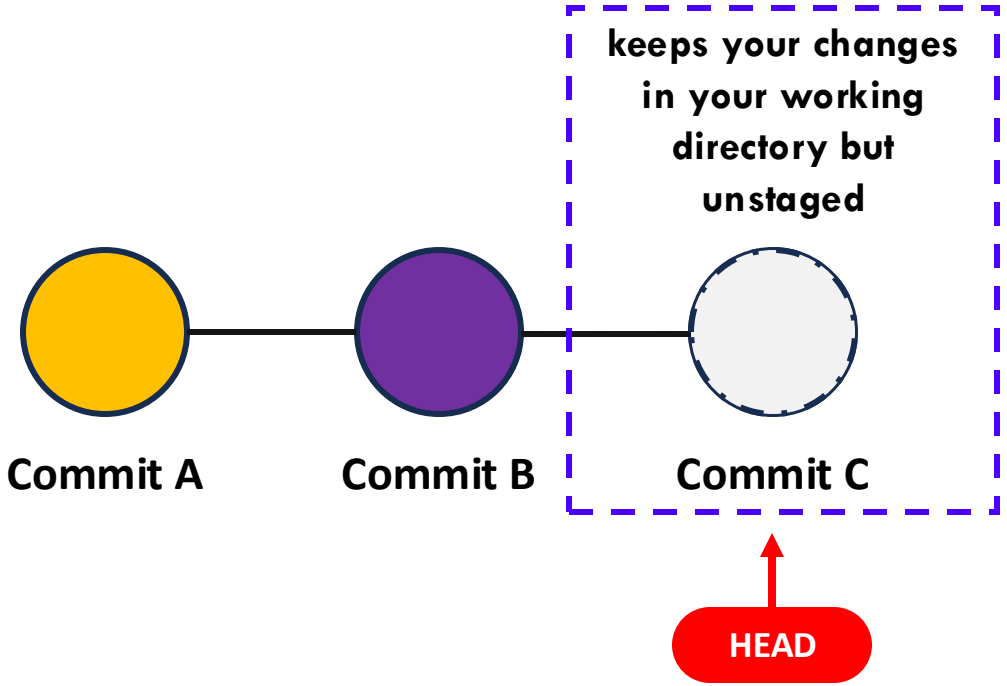
soft reset **mixed reset** **hard reset**

soft reset



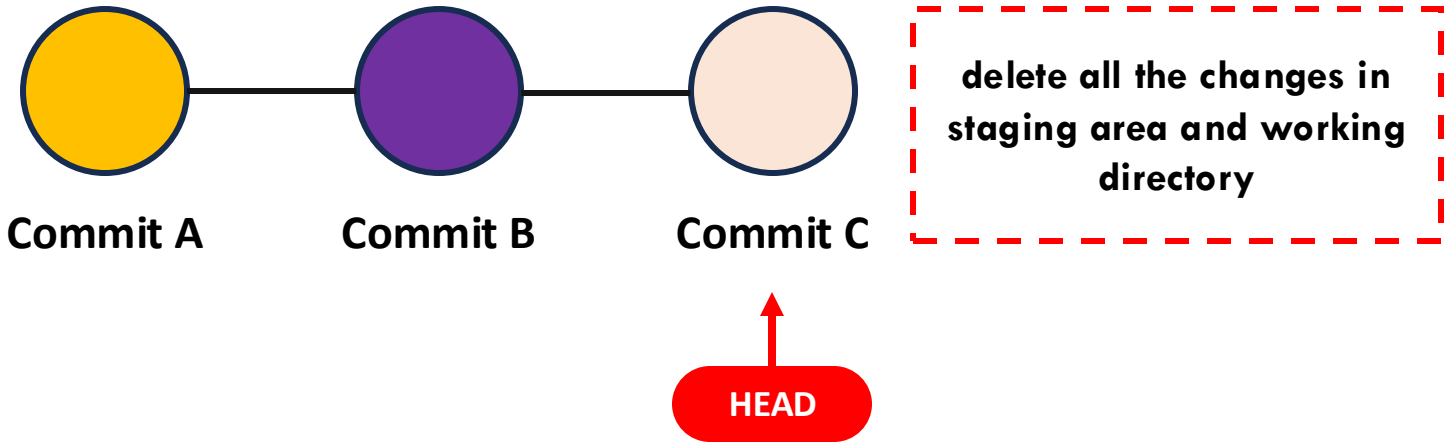
Useful if you want to rewrite history but still keep your work

mixed reset *Default*



Files are still there, but you need to add them again if you want to commit

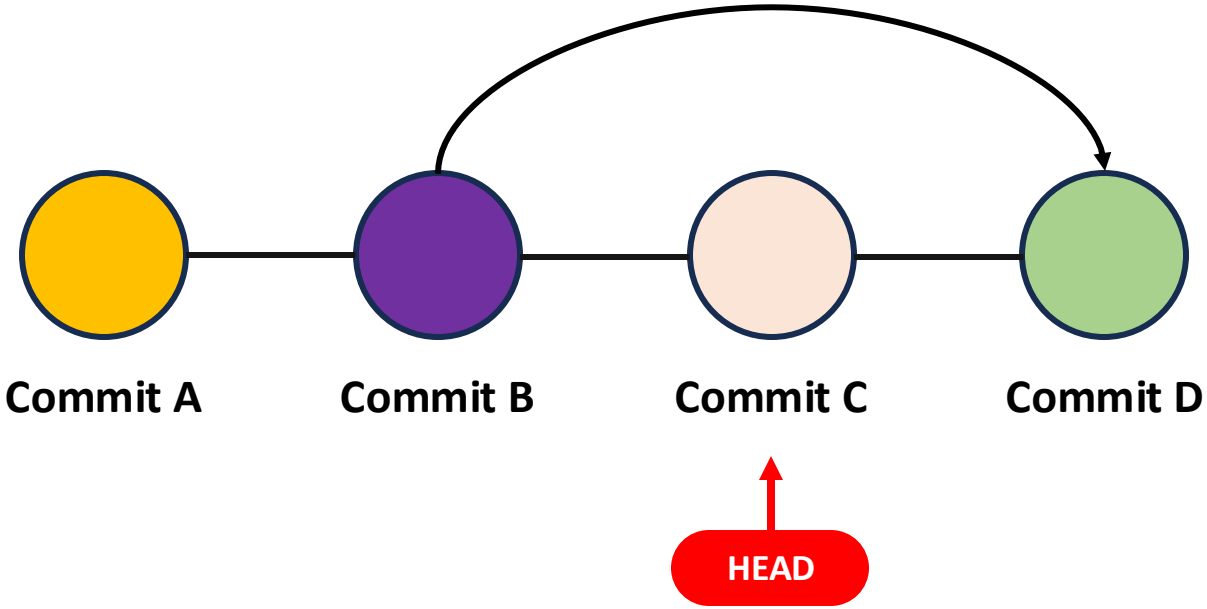
hard reset *most dangerous*



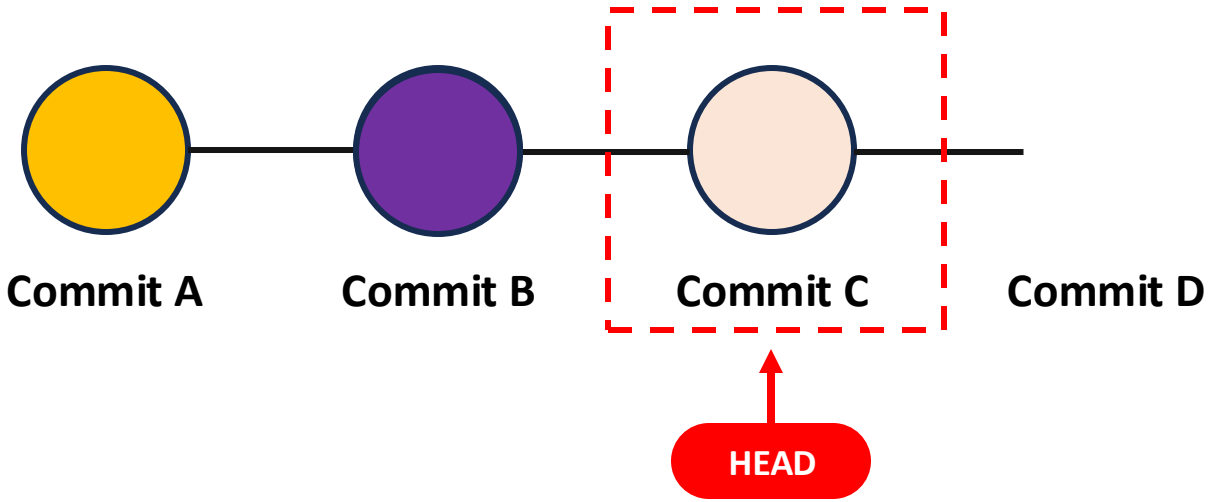
Be very careful with hard reset

git revert

Instead of rewinding history, **git revert** creates a brand-new commit that undoes the changes of a previous commit

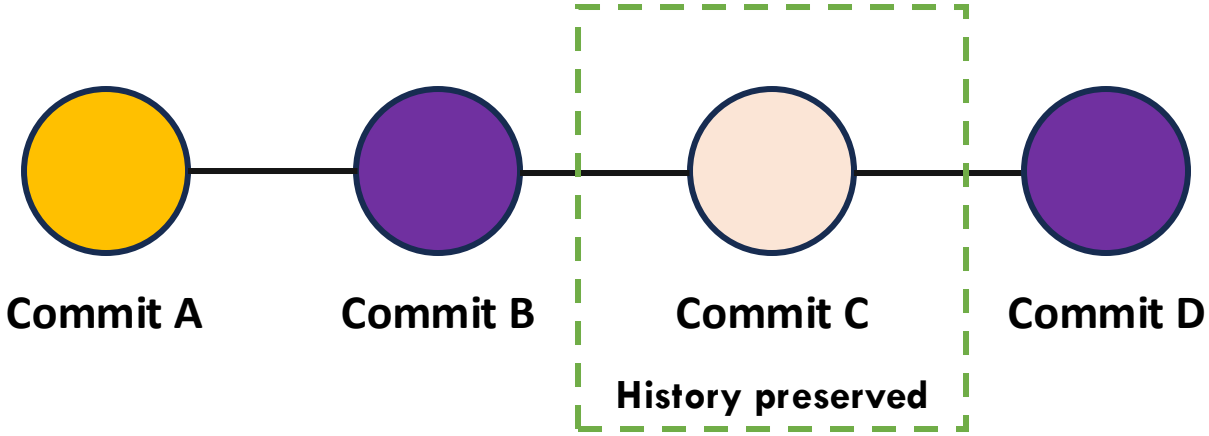


git revert




```
git revert
```

git revert



revert is often preferred in collaborative projects,
while **reset** is more common when you're working on your own

git reset

Rewinds history and can even erase commits if you're not careful

git revert

Keeps history intact but adds new commit to undo changes

GitHub Basics

GitHub Basics

Section Overview

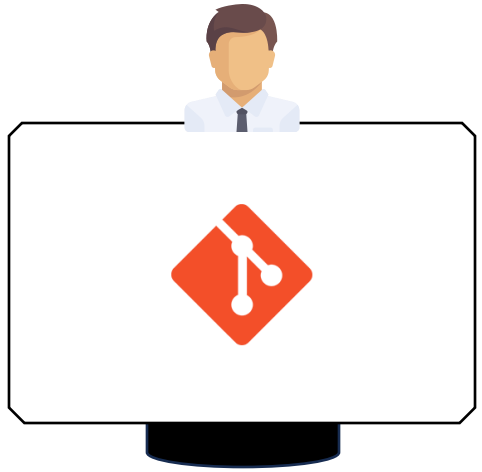
- → *Why remote repositories are essential*
- → *Introduction to GitHub*
- → *Creating and setting up your GitHub account*
- → *Create our first repository on GitHub*
- → *Connect your local Git setup to GitHub using different authentication method*

Why We Need Remote Repositories

Why We Need Remote Repositories

- History
- Branches
- Changes

Why We Need Remote Repositories

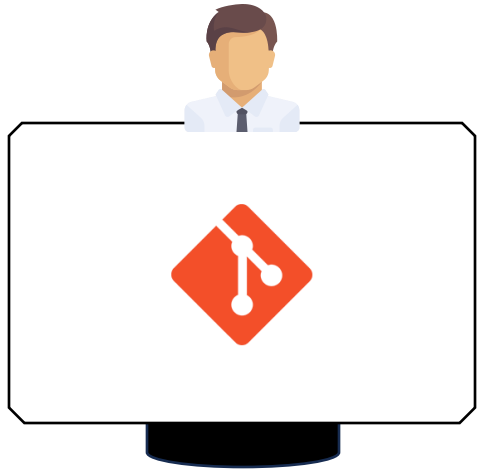


Limitations

Local Repository

Private and convenient, but nobody else can see it

Why We Need Remote Repositories



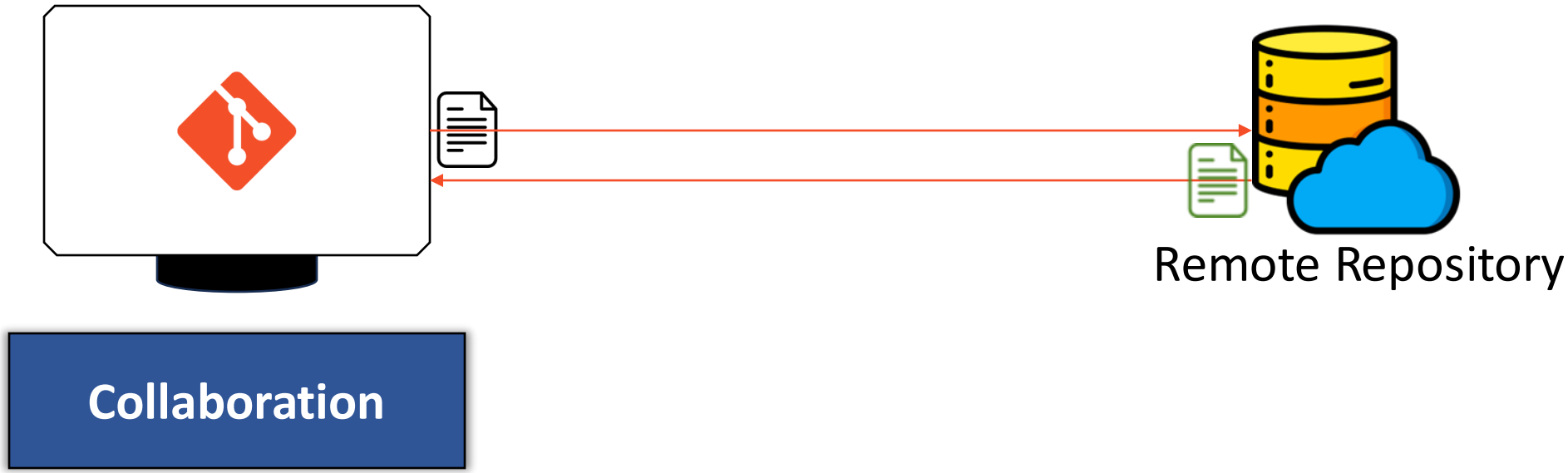
Limitations

Local Repository

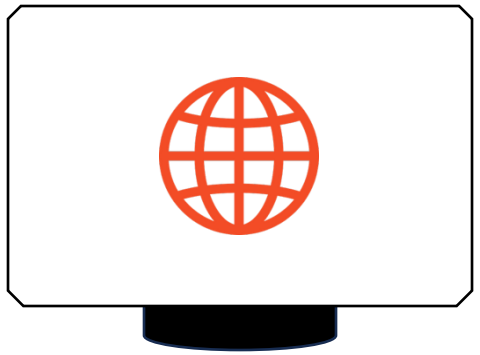
Remote Repository

Act as a shared space, usually on a server or in the cloud

Why We Need Remote Repositories



The Importance of Collaboration



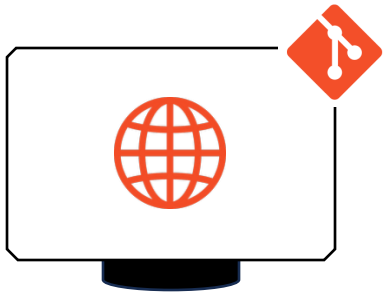
Homepage

Login System

Desiging

The Importance of Collaboration

Homepage



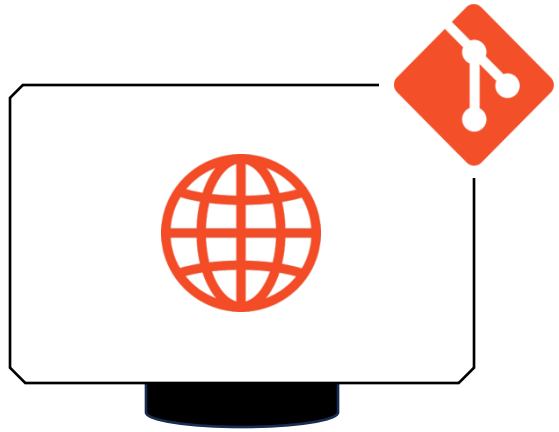
Login System

Styling

How would you combine your work?

- Messy
- Slow
- Prone to errors

The Importance of Collaboration



Homepage

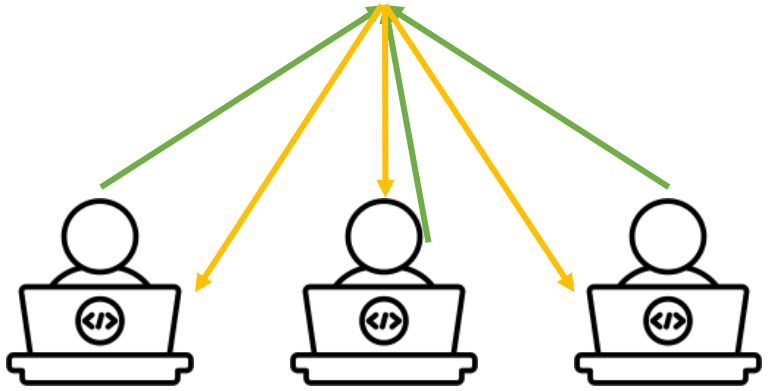
Login System

Styling

The Importance of Collaboration

Central Hub

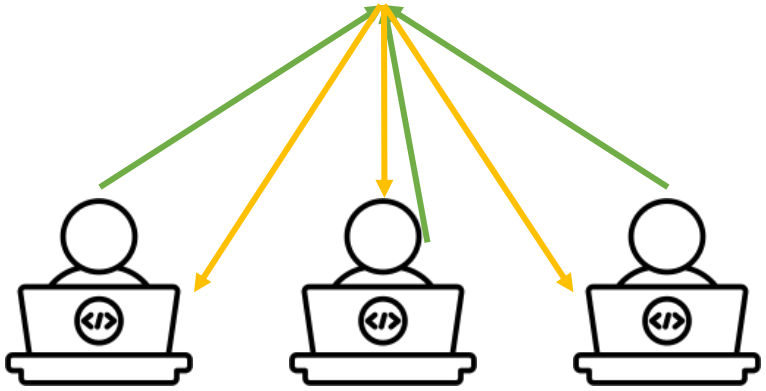
Remote Repository



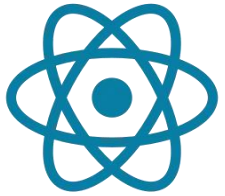
The Importance of Collaboration

Central Hub

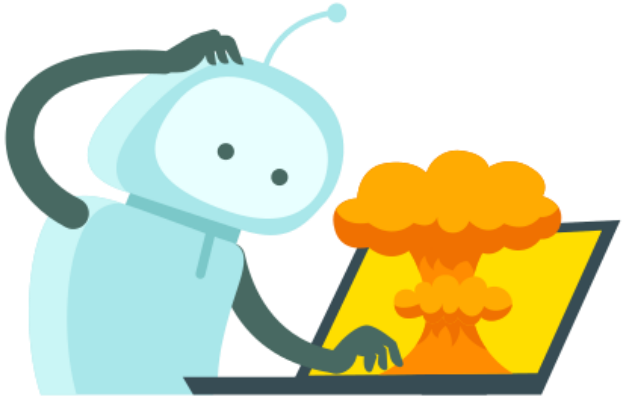
Remote Repository



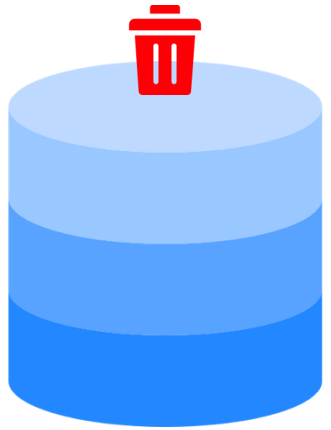
Open Source Projects



Why we need an Online Backup?



Laptop Crash



Local Repository

Why we need an Online Backup?

Remote Repository

Long-term Projects

Professional Work

Always have a backup waiting in the cloud

Benefits of Remote Repositories

Remote Repository

Hosting your projects on platforms like GitHub, you can build a portfolio that potential employers or clients can see

Benefits of Remote Repositories

Remote Repository

✓ Allow integration with powerful tools

Continuous Integration | **Issue Tracking** | **Deployment Pipelines**

✓ Make software development faster and more reliable

Getting Started with GitHub

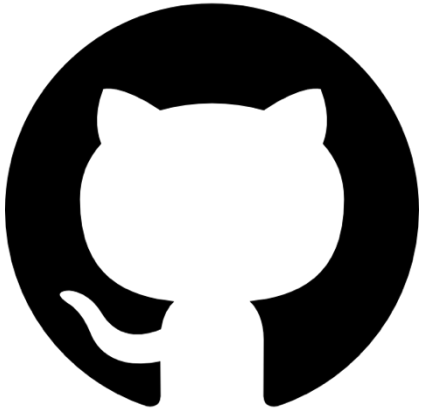
Getting Started with GitHub

Git

Version Control System

Getting Started with GitHub

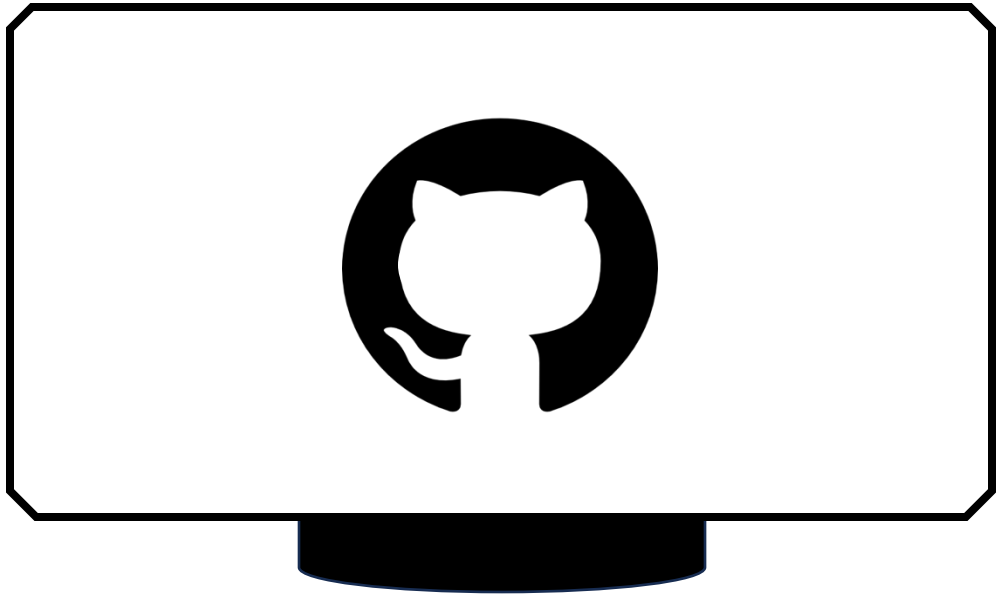
Git



GitHub

- Cloud-based platform for hosting Git repositories online
- Combines Git's version control with collaboration features

Getting Started with GitHub



- Share
- Collaborate
- Build Together

Git vs GitHub

Git

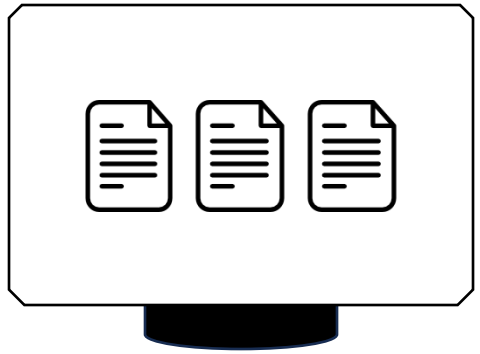
- Git is a tool
- Command-line version control system
- Can use it without the internet

GitHub

- GitHub is a service
- Host Git repositories
- Adds extra features like:
  - Collaboration
  - Issue tracking
  - Project boards
  - Pull & push requests

Git vs GitHub

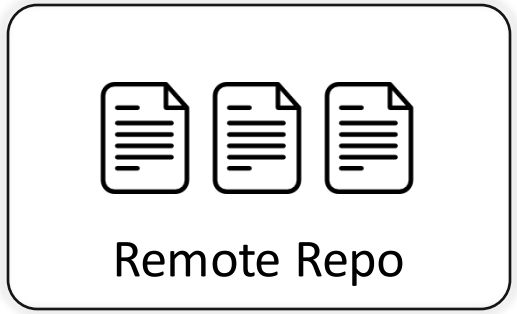
- Stays with you
- Holds your money



Provide complete control over your code, privately

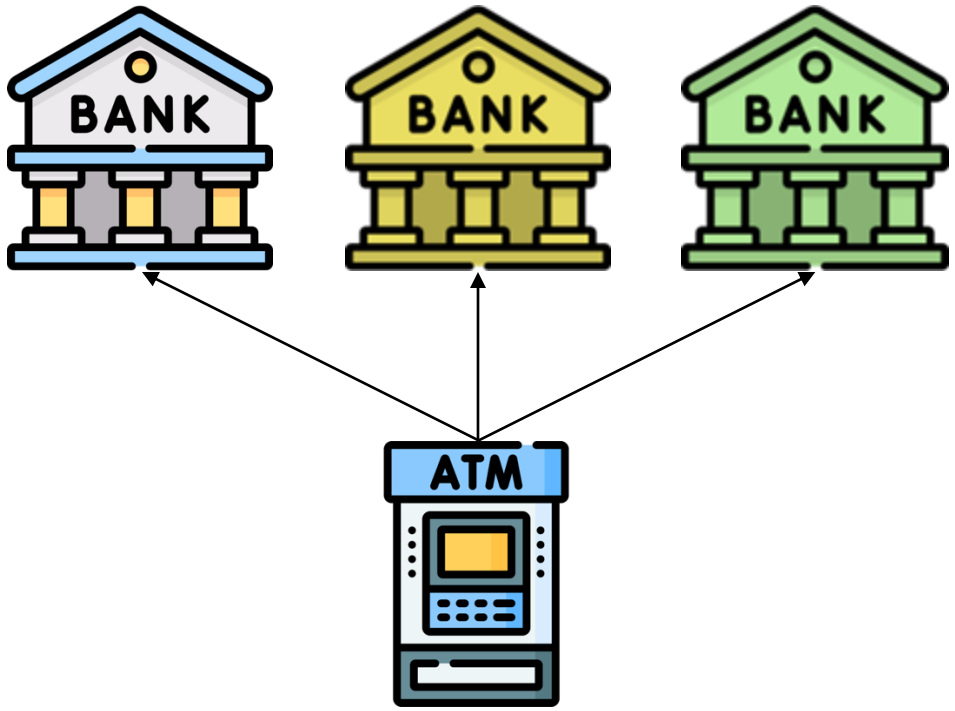
Git vs GitHub

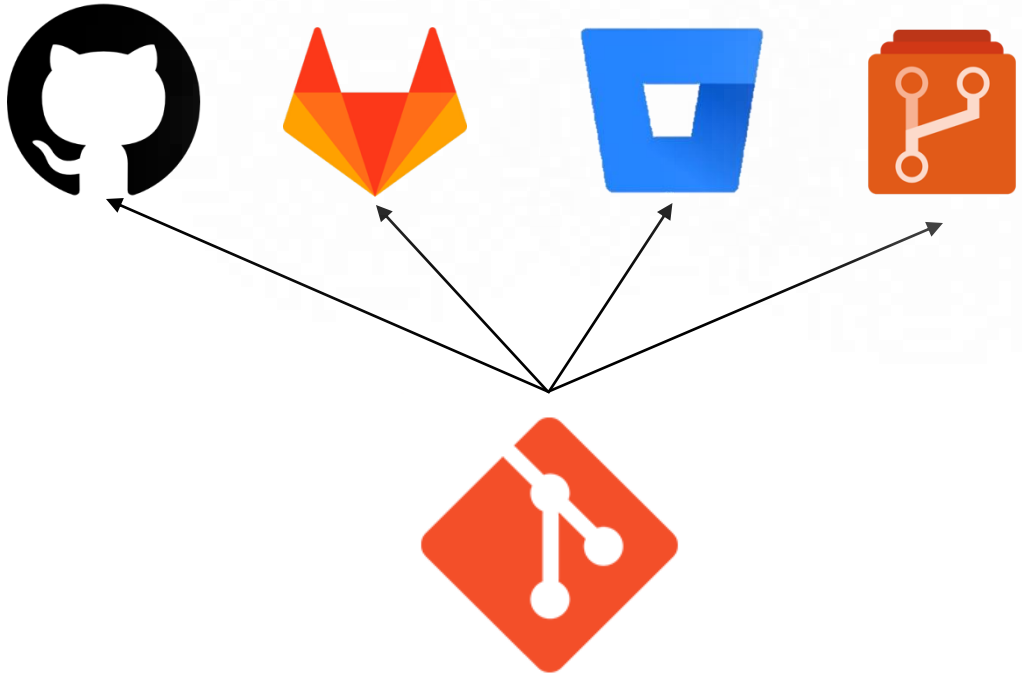
Secure & online space to store that money



- Access it from anywhere
- Share it with others
- Work together on the same project

Git vs GitHub





Git vs GitHub

Git

Manages and tracks your code

GitHub

Stores and shares that code with the world

Git vs GitHub

- Collaborative
- Organized
- Version Controlled

Git is the foundation, and GitHub builds on top of it

Collaboration

GitHub's Role in Open-Source and Collaboration

Invite Teammates

Review Code

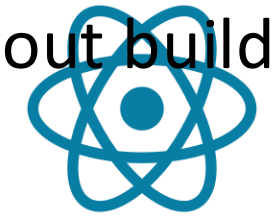
Issues & Tasks

Open-Source Projects

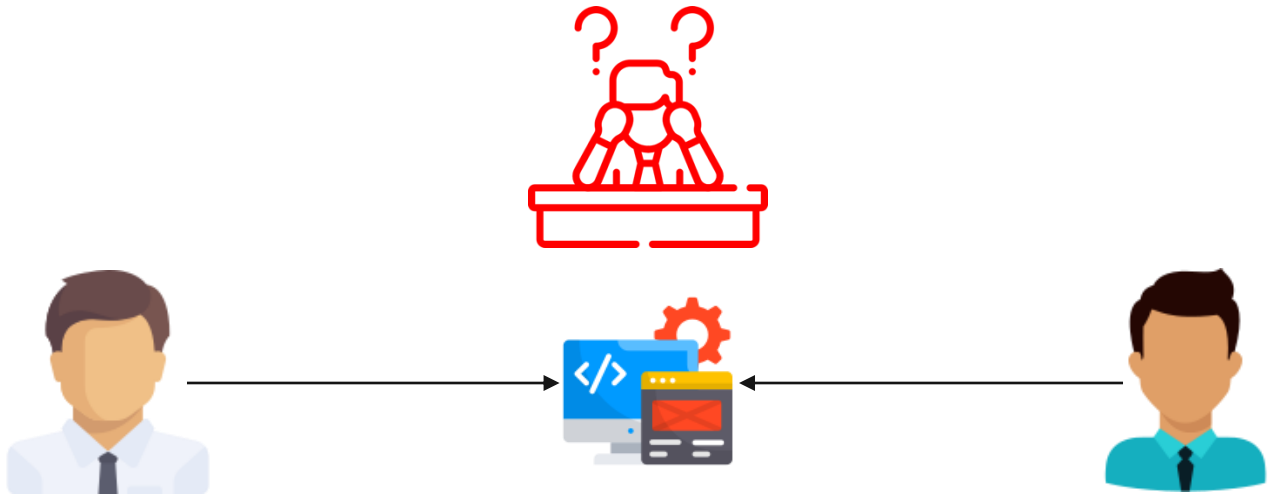
Collaboration

Open-Source Projects

It’s about building a community around code



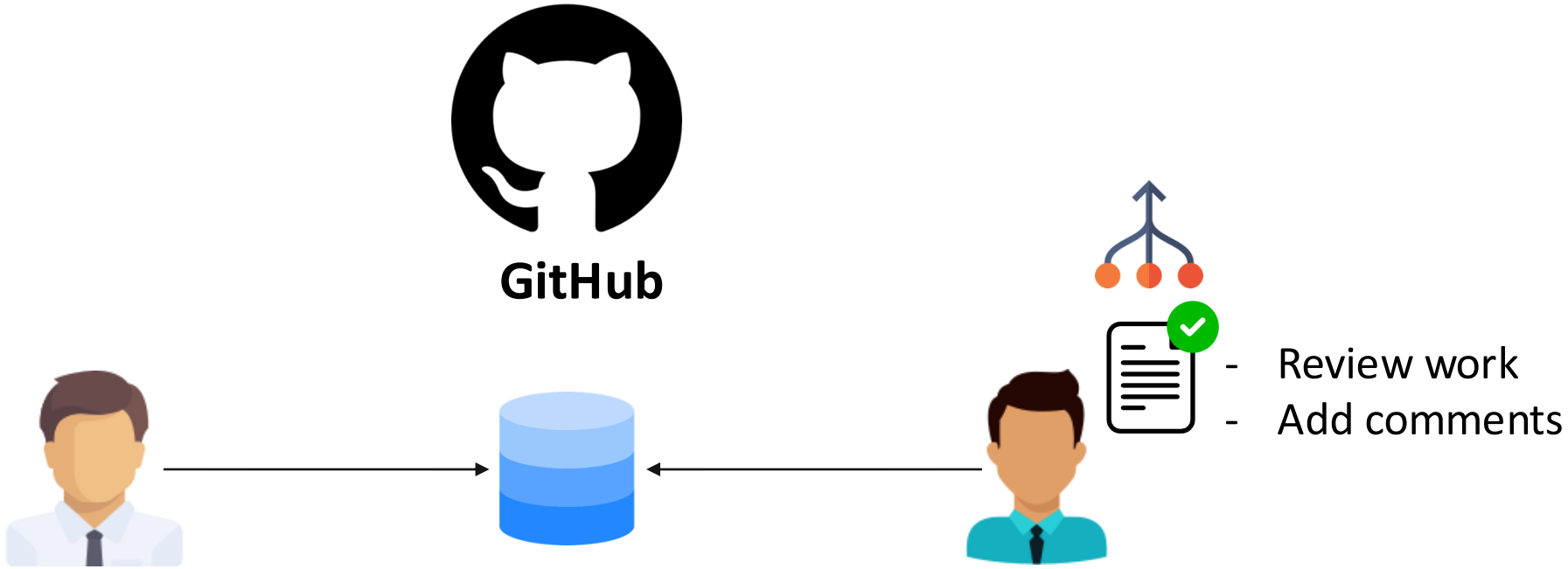
Collaboration in Action



Exchange files via email or messaging apps

GitHub

Collaboration in Action



- Create Branches
- Make Changes
- Submit Pull Requests

Collaboration in Action

No Confusion

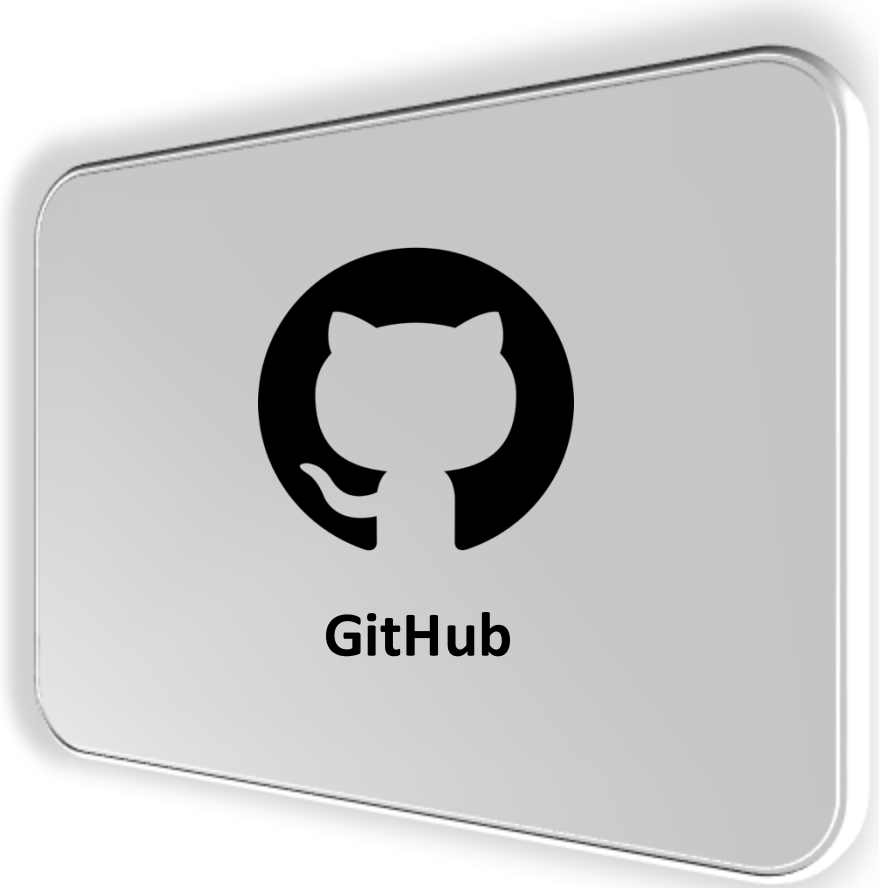
No Lost Files

Smooth Collaboration

Is GitHub the only option?

Why Choose GitHub?

Why Choose GitHub?



Community

Ecosystem

- Tools
- Services
- Workflows

GitHub

Largest community of developers

Collaborating	Learning	Contributing

Language

Social Network

- Shares
- Collaborates
- Builds together

go-to platform

- Beginners
- Professionals
- Open-Source Contributors

Demo

Setting up GitHub Account

Demo

Quickly Exploring GitHub

Demo

Creating Your First Repository on GitHub

Demo

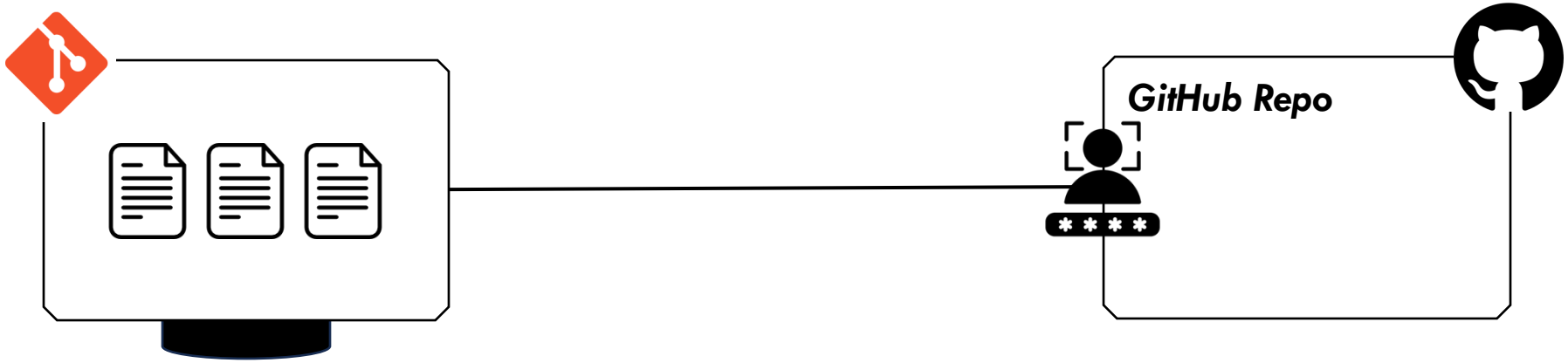
Connecting Local Git to GitHub

SSH vs HTTPS Authentication in GitHub

HTTPS Authentication

VS

SSH Authentication

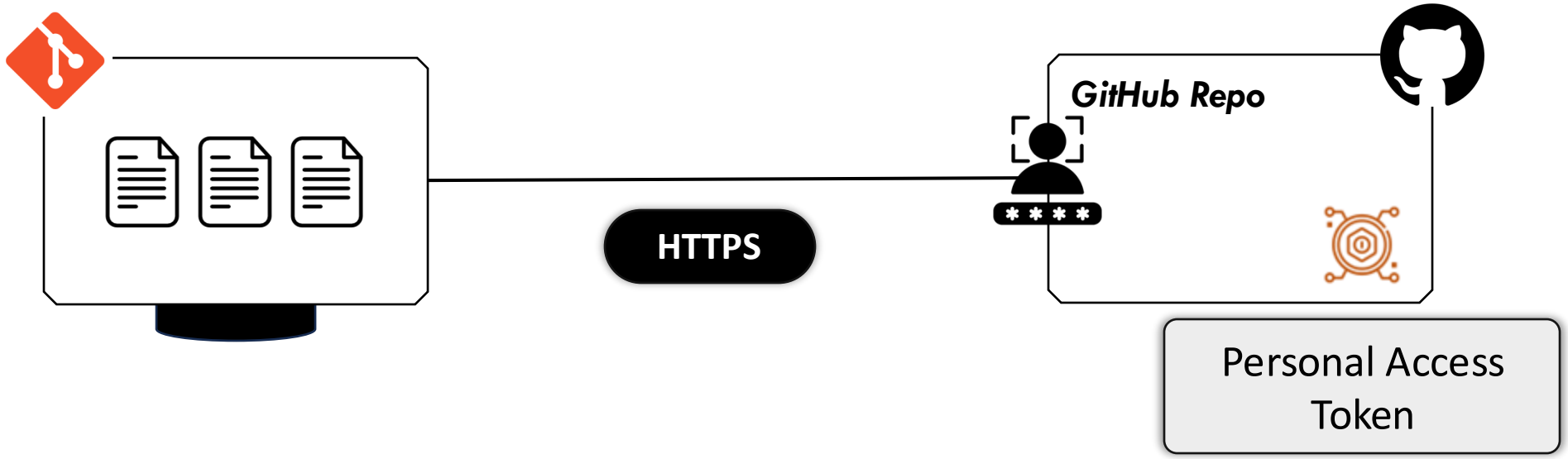


HTTPS

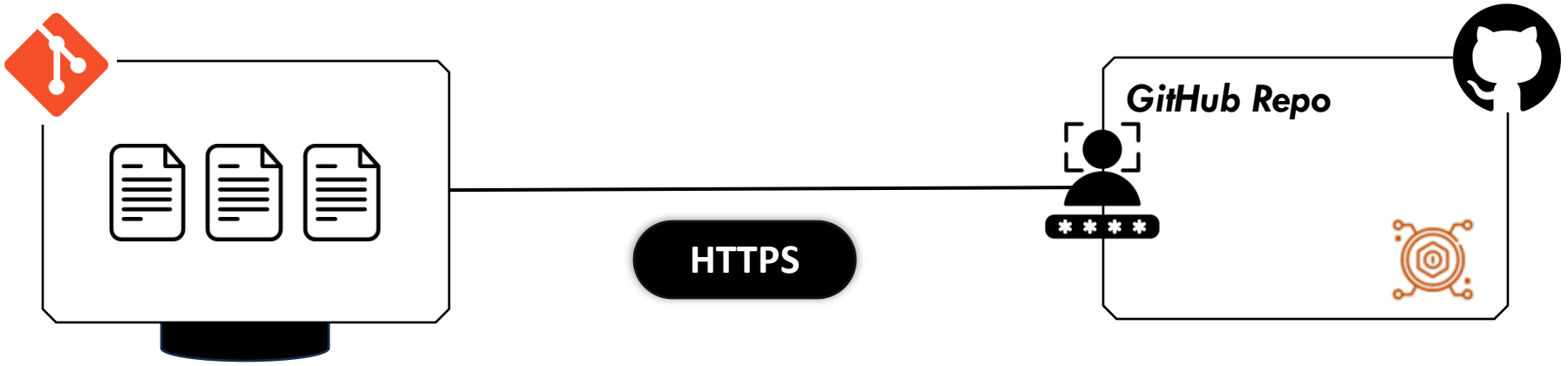
SSH

HTTPS Authentication

Username
Password



HTTPS Authentication

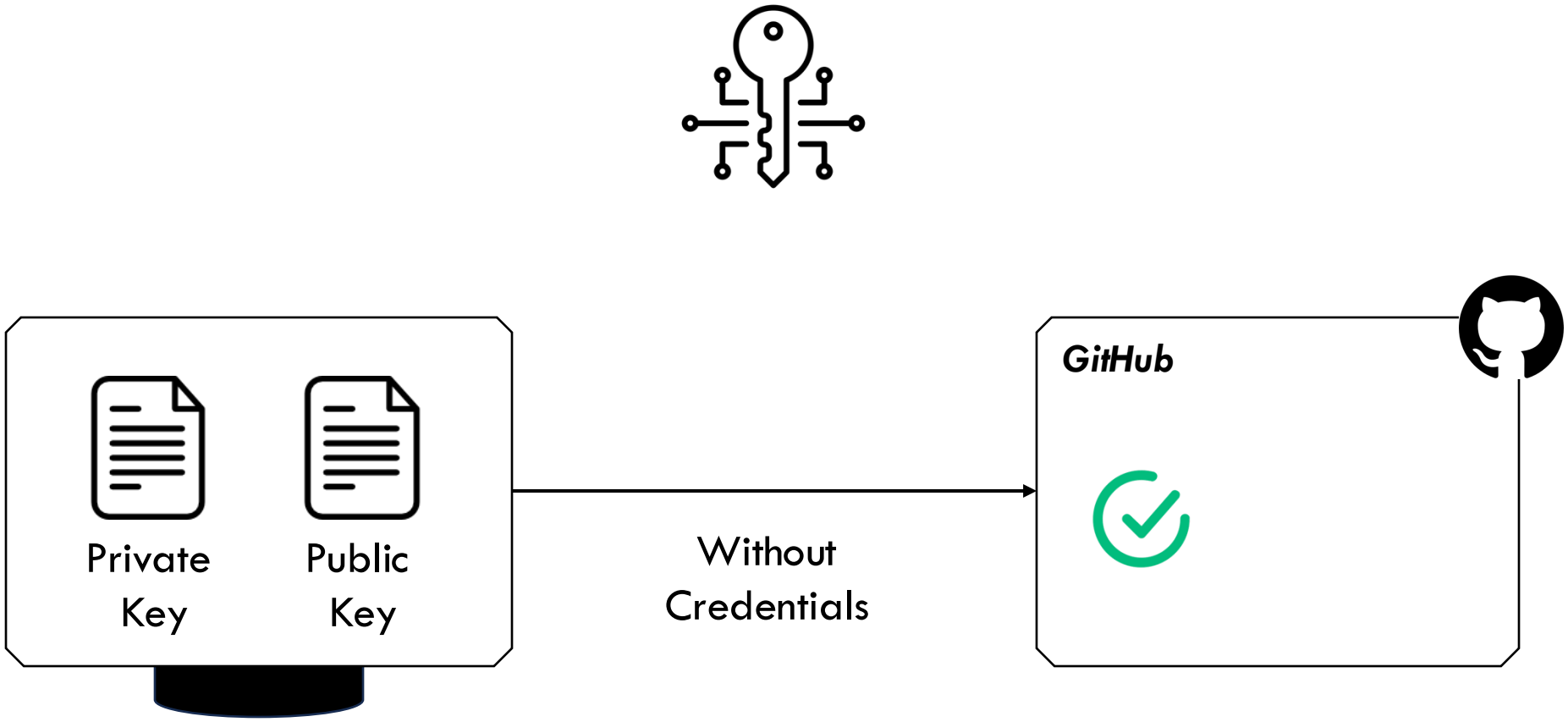


Personal Access Token

Password

- Easy to setup
- Works everywhere

SSH Authentication



HTTPS Authentication

- Easy for beginners
- Works well if you don’t use GitHub regularly

SSH Authentication

- Push and pull code regularly
- Saves you from entering credentials every time
- More secure since it’s based on cryptographic keys

Demo

SSH vs HTTPS Authentication in GitHub

Working with Remotes & Repositories

Working with Remotes & Repositories

Section Overview

- → *Add, Remove, and View Remote Repositories*
- → *Pushing & Pulling Basics*
- → *Difference between **git fetch** and **git pull***
- → ***Demonstration:** Cloning Repositories*

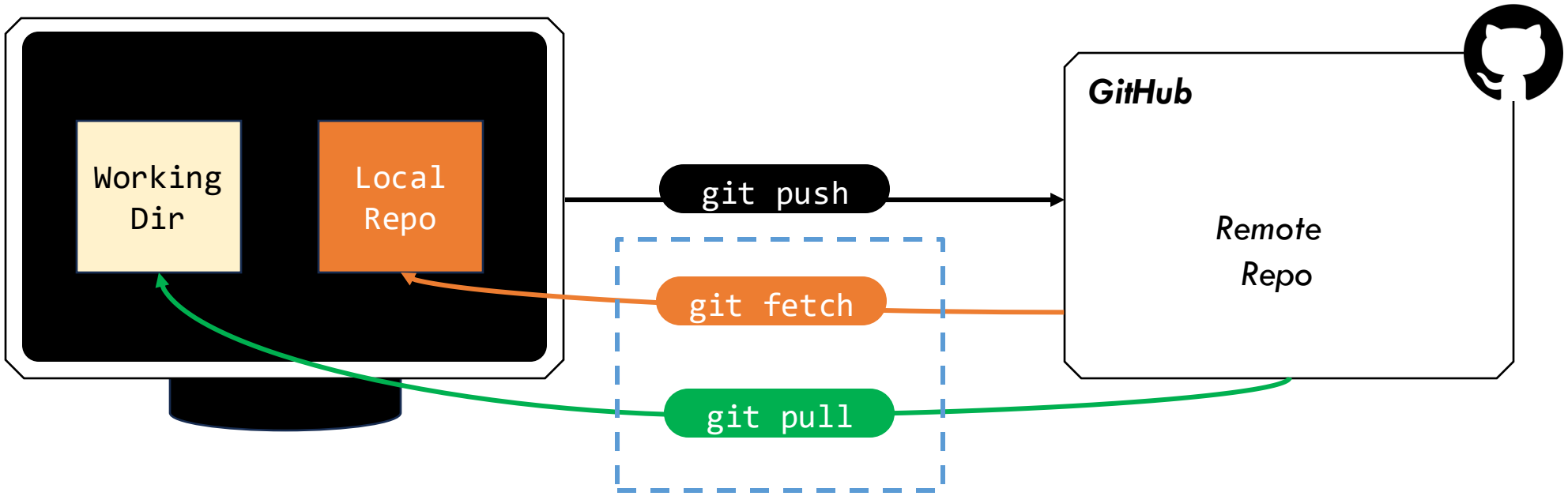
Demo

Adding, Removing & Viewing Remotes

Demo

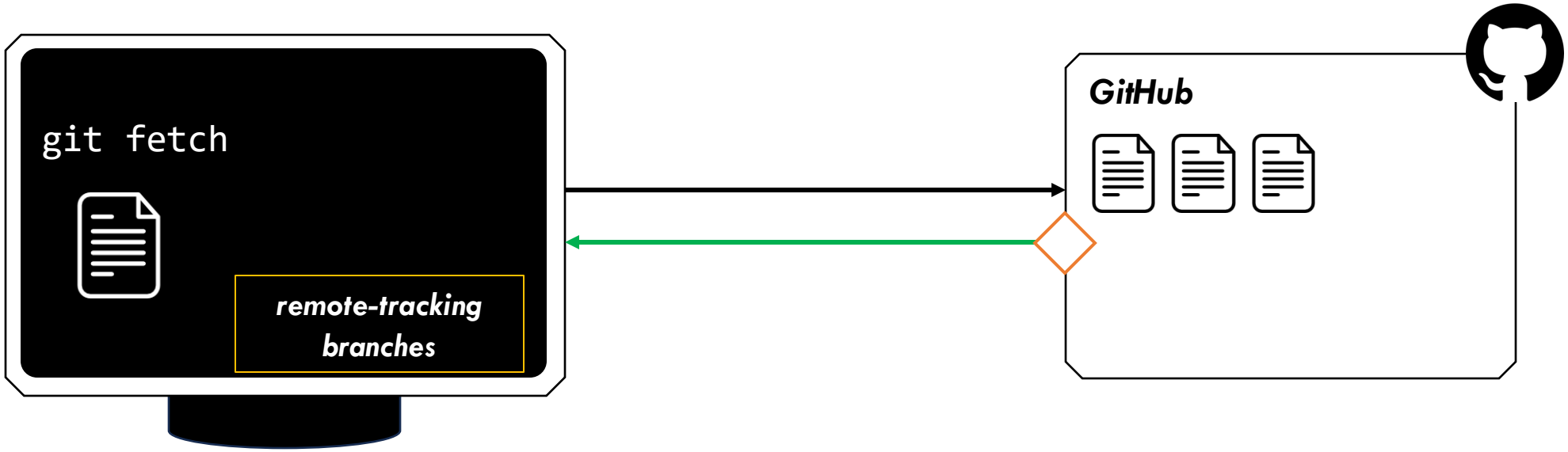
Pushing & Pulling Basics

*Understanding difference between **git fetch** & **git pull***



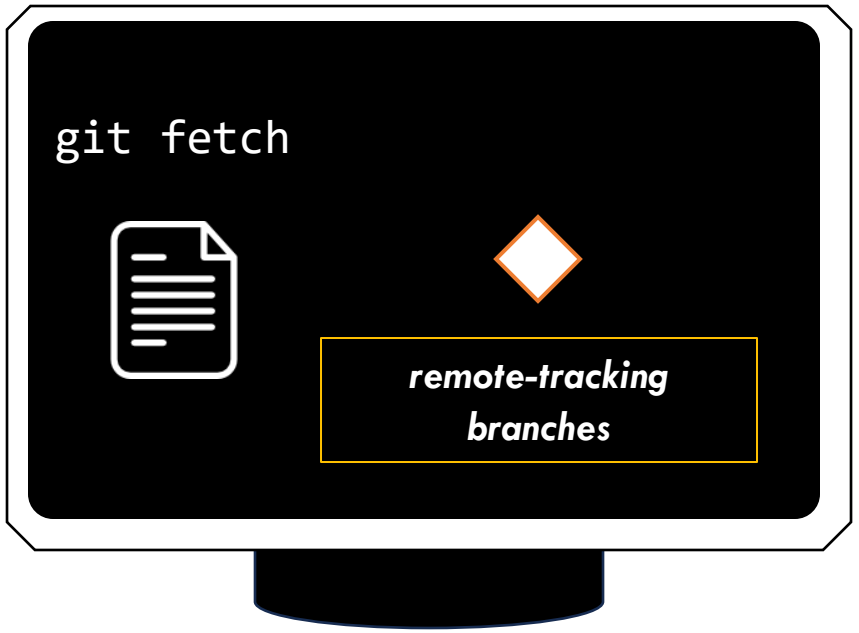
What's the difference between these two commands?

git fetch



Doesn't automatically merge them into your local branch

git fetch

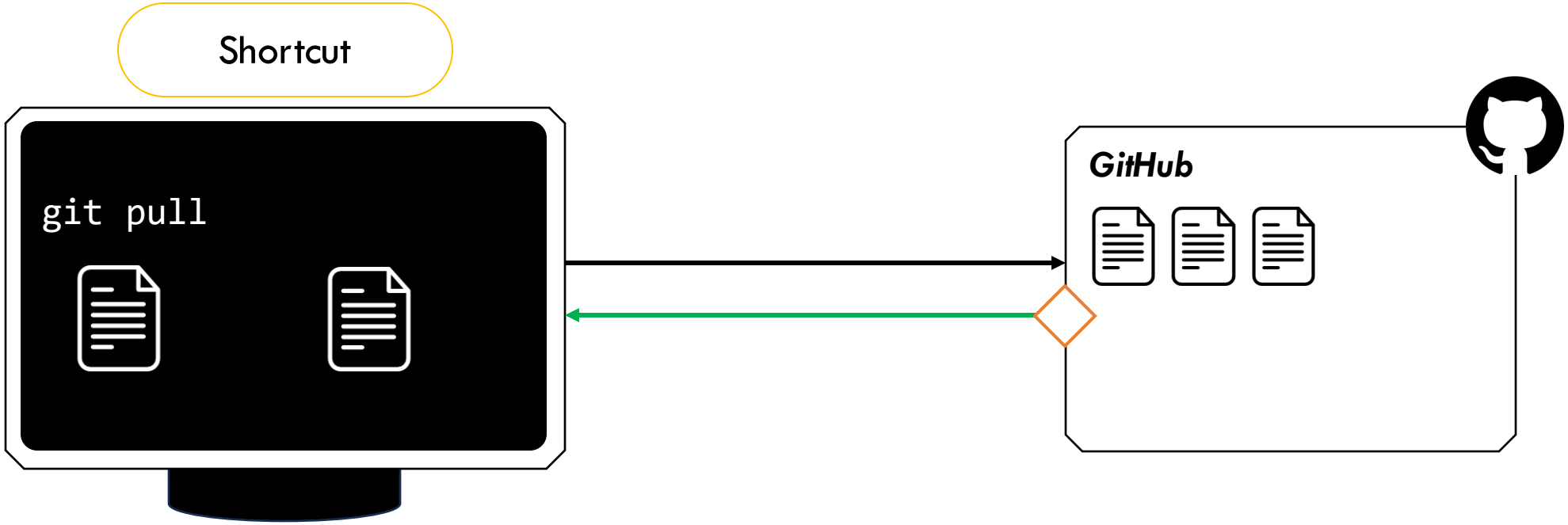


Review changes

✓ Merge changes

- More control
- Avoid conflicts in code

git pull



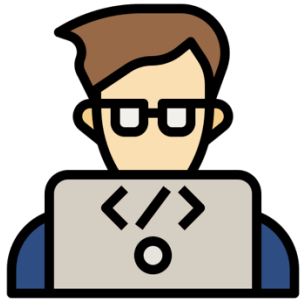
- Faster
- Less safe

`git fetch`

Review changes first before bringing them into your branch

`git pull`

Ready to fetch and merge in one go



Fetch

Review

Merge

`git fetch`

`git pull`

Demo

Cloning Repositories

Dealing with Everyday Situations in Git

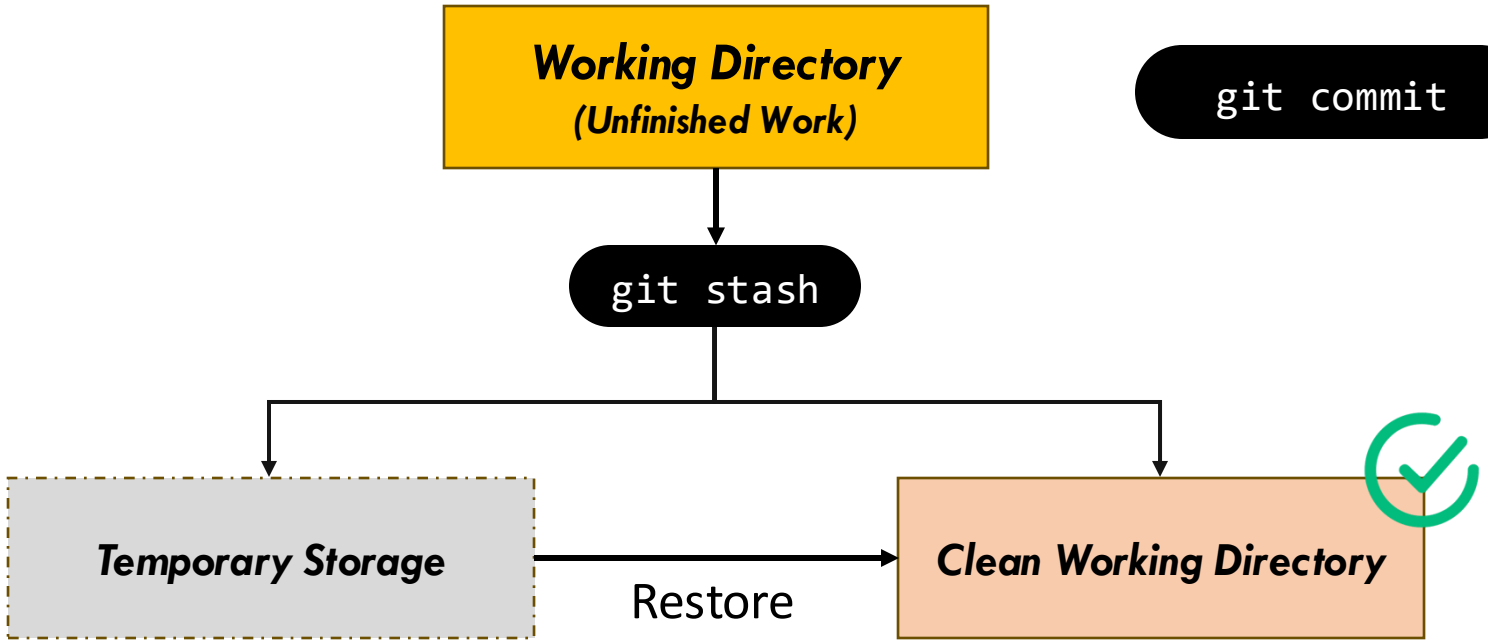
Dealing with Everyday Situations in Git

Section Overview

- → *Stashing Changes*
- → *Rebasing in Git*
- → ***Demonstration:*** *Rebasing*

Stashing Changes

Stashing



Useful when we're in the middle of something & suddenly need to switch branches or handle another task

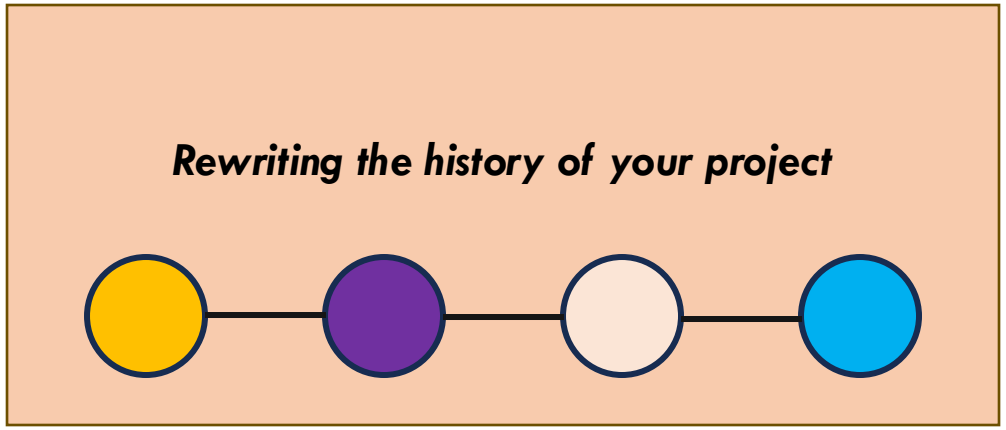
Demo

Stashing Changes

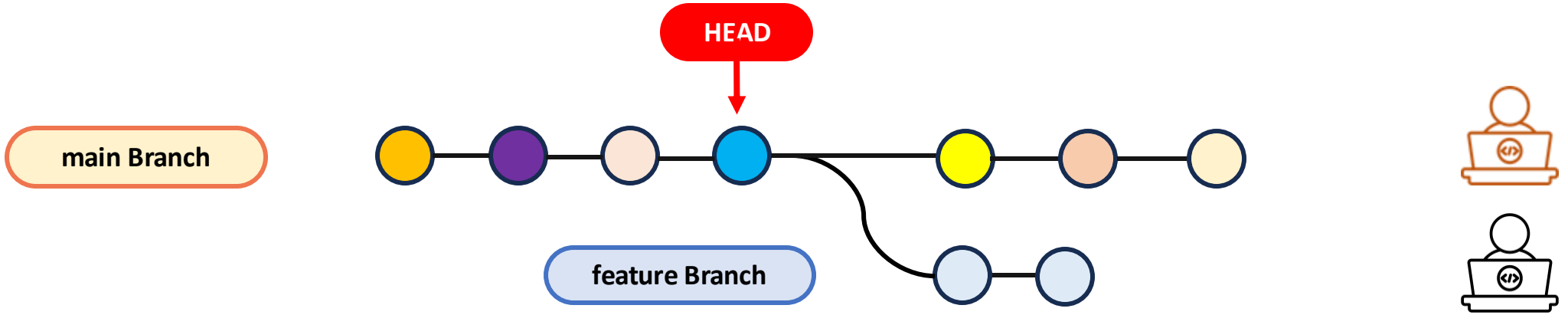
Rebasing in Git

Rebasing in Git

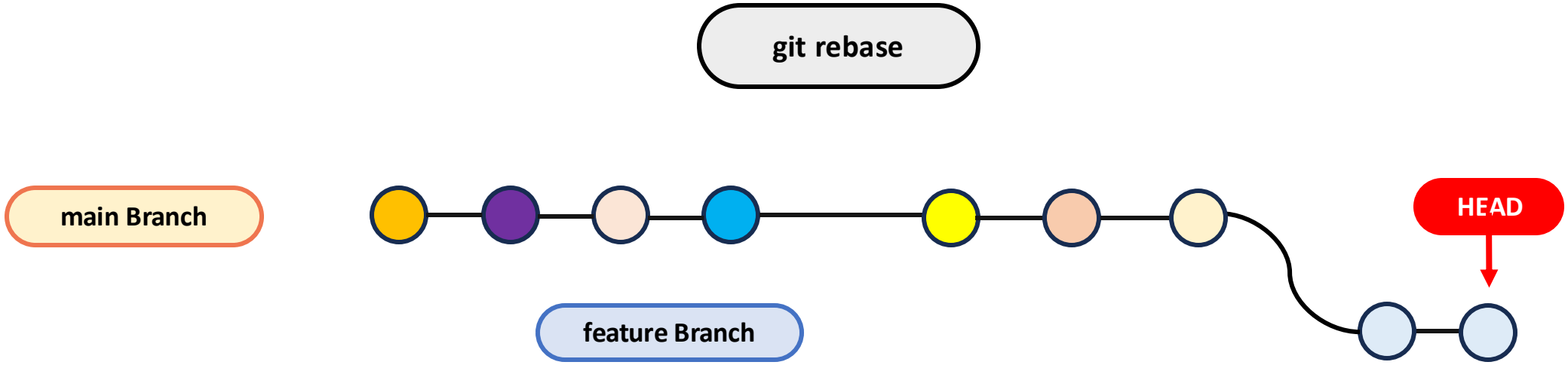
Rebasing is a way of moving or combining a series of commits onto a new base commit



Rebasing in Git

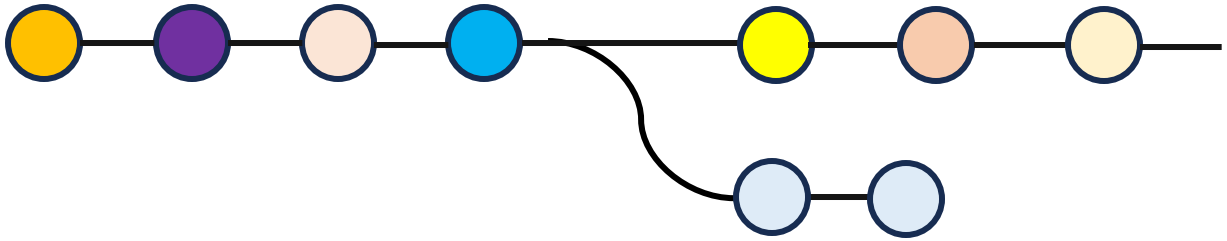


Rebasing in Git



Why Rebasing?

Maintain a cleaner and more linear project history in your Git project



Avoids:

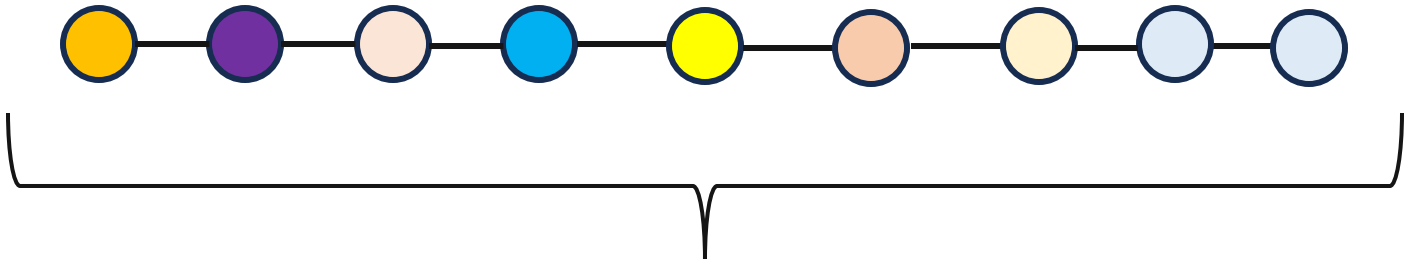
- *Unnecessary branching paths*
- *Extra merge commits*

Provides you neat, straight line of commits which is easier to read and understand

Smooth Collaboration

Why Rebasing?

No need to deal with extra merge commits



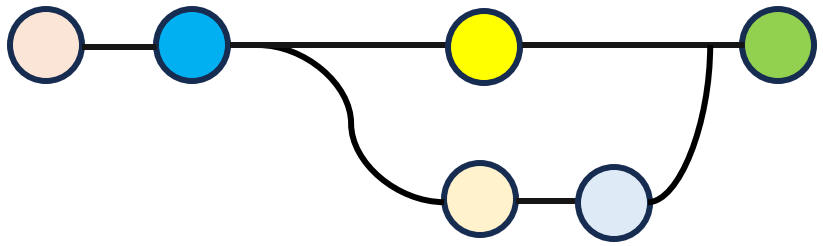
Developed in a direct sequence

Rebasing

Merging

Merging vs Rebasing

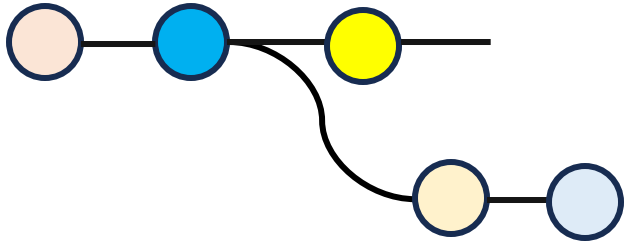
Merging



Merge Commit

- → Retains the complete history of both branches
- → History of when & where the two branches joined

Rebasing



- → Makes it look like your work started from the main branch
- → No extra merge commits, just a smooth line of commits

Merging vs Rebasing

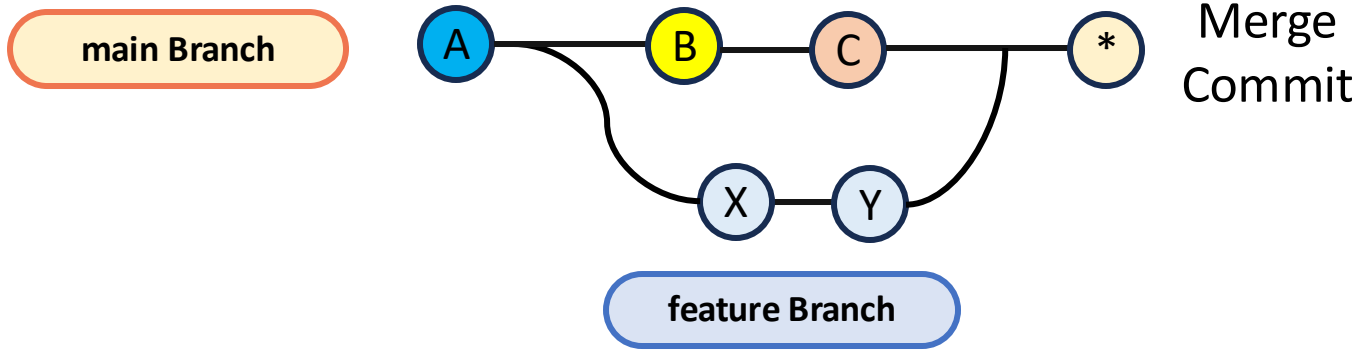
Merging

Preserve the complete history of how development happened, including all the merges

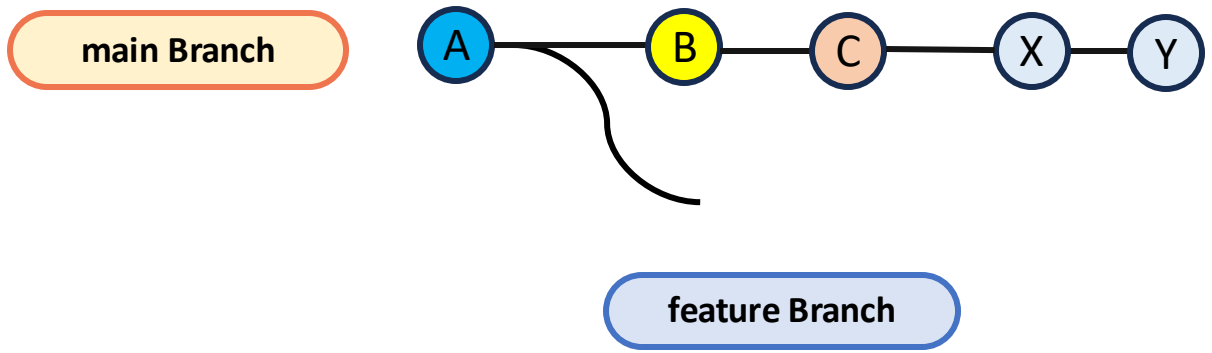
Rebasing

Simpler and cleaner history without the extra noise of merge commits

Merging



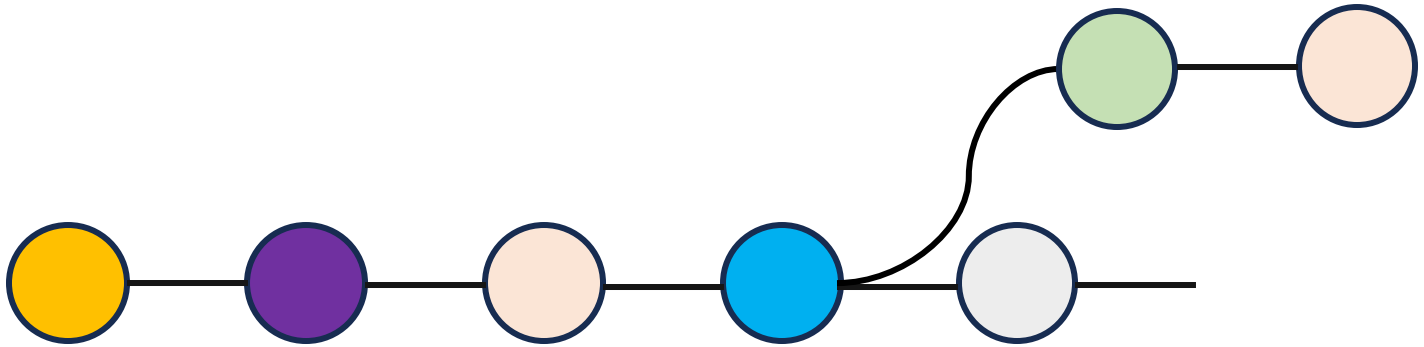
Rebasing



When To Avoid Rebasing?

Rebasing

Rewrites History



Confusion

Conflicts

When To Avoid Rebasing?

Rebasing

Only rebase branches that you haven’t shared with others

For shared branches like main or master

Merging

Demo

Rebasing in Git

Understanding GitHub Workflow and Collaboration

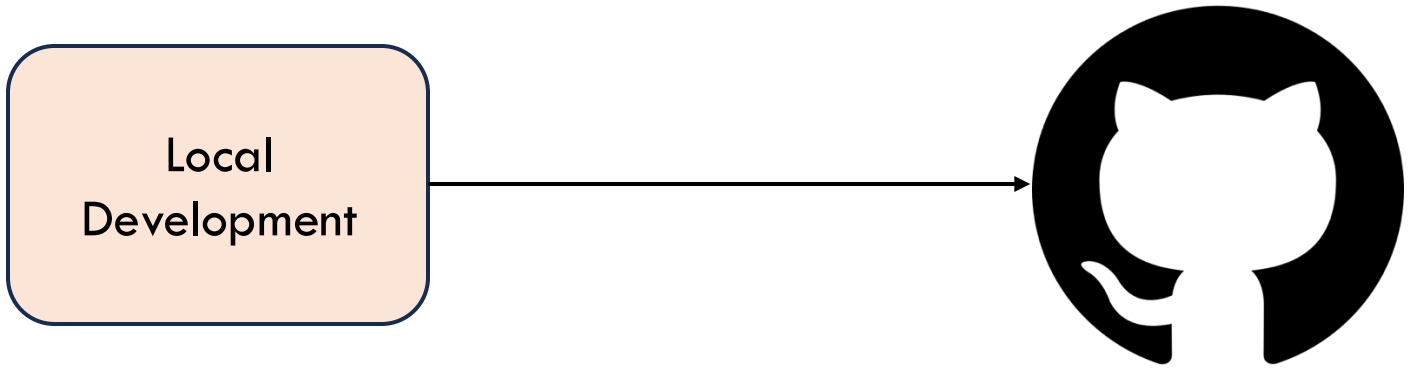
Understanding GitHub Workflow and Collaboration

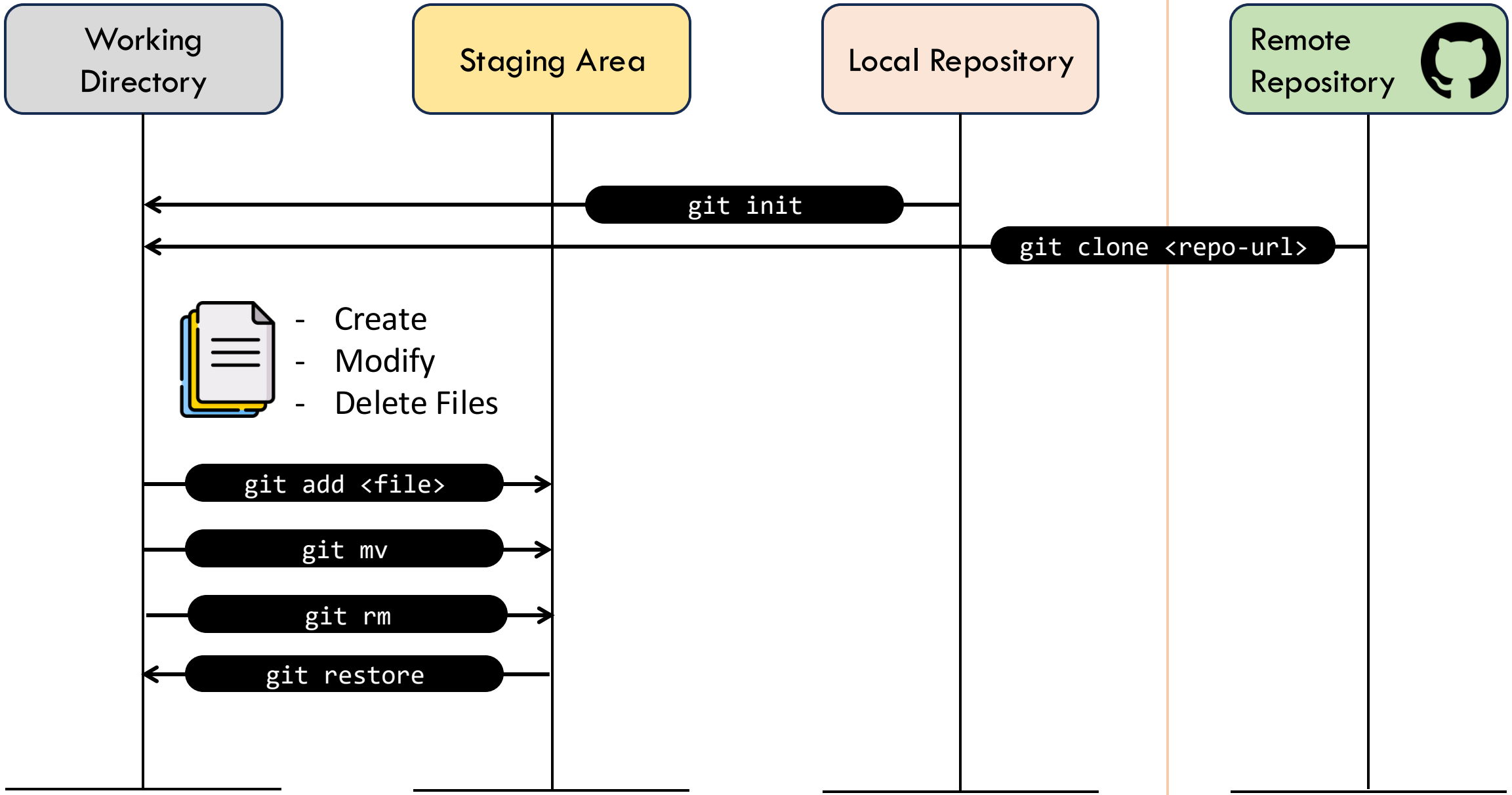
Section Overview

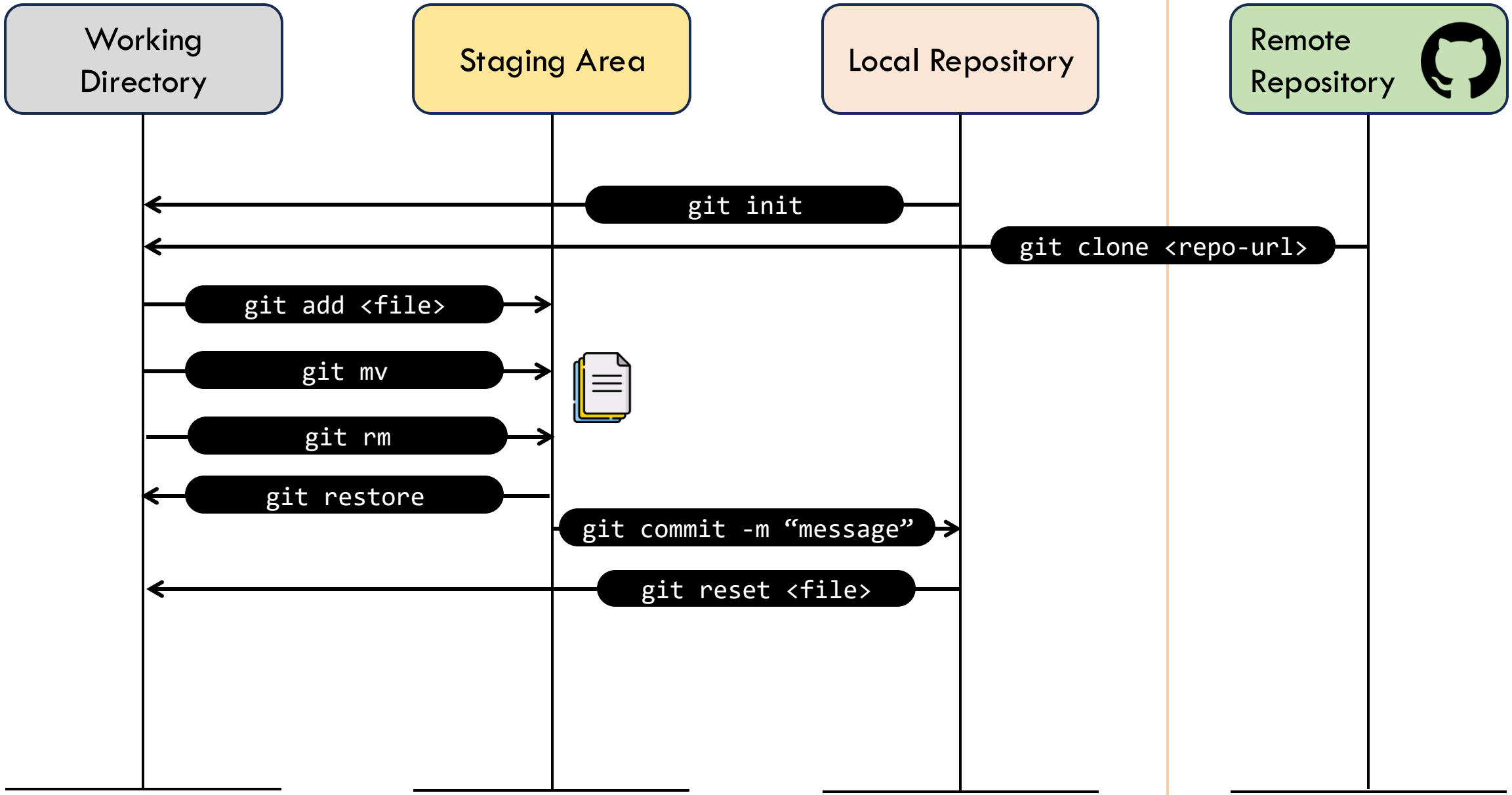
- → *Git and GitHub workflow*
- → ***Demonstration:*** *Project Walkthrough*
- → ***Demonstration:*** *Forking in GitHub*
- → ***Demonstration:*** *Opening Issues & Discussions*
- → ***Demonstration:*** *Creating & Managing Pull Requests*
- → ***Demonstration:*** *Repository Settings*

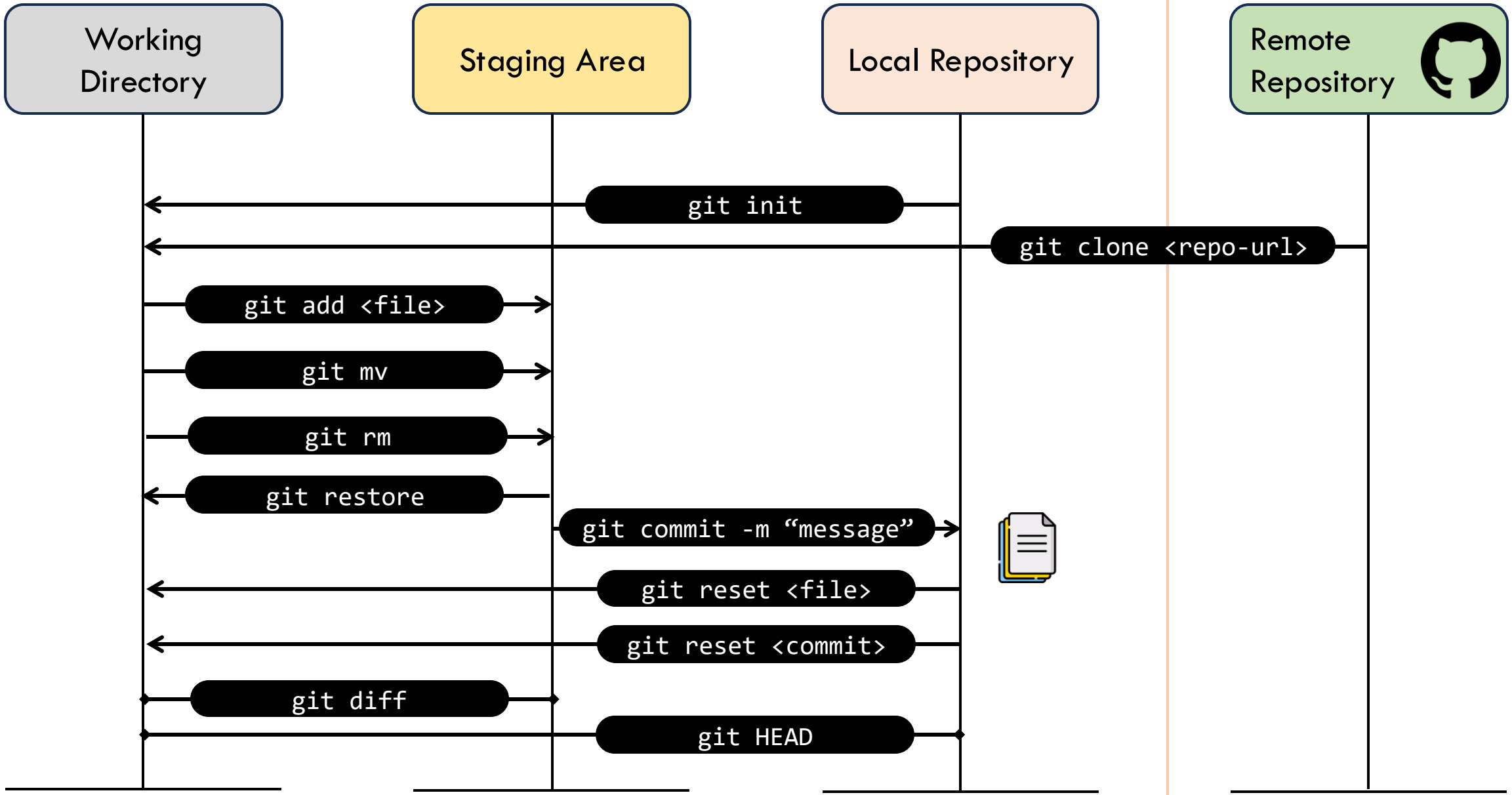
Git & GitHub Workflow

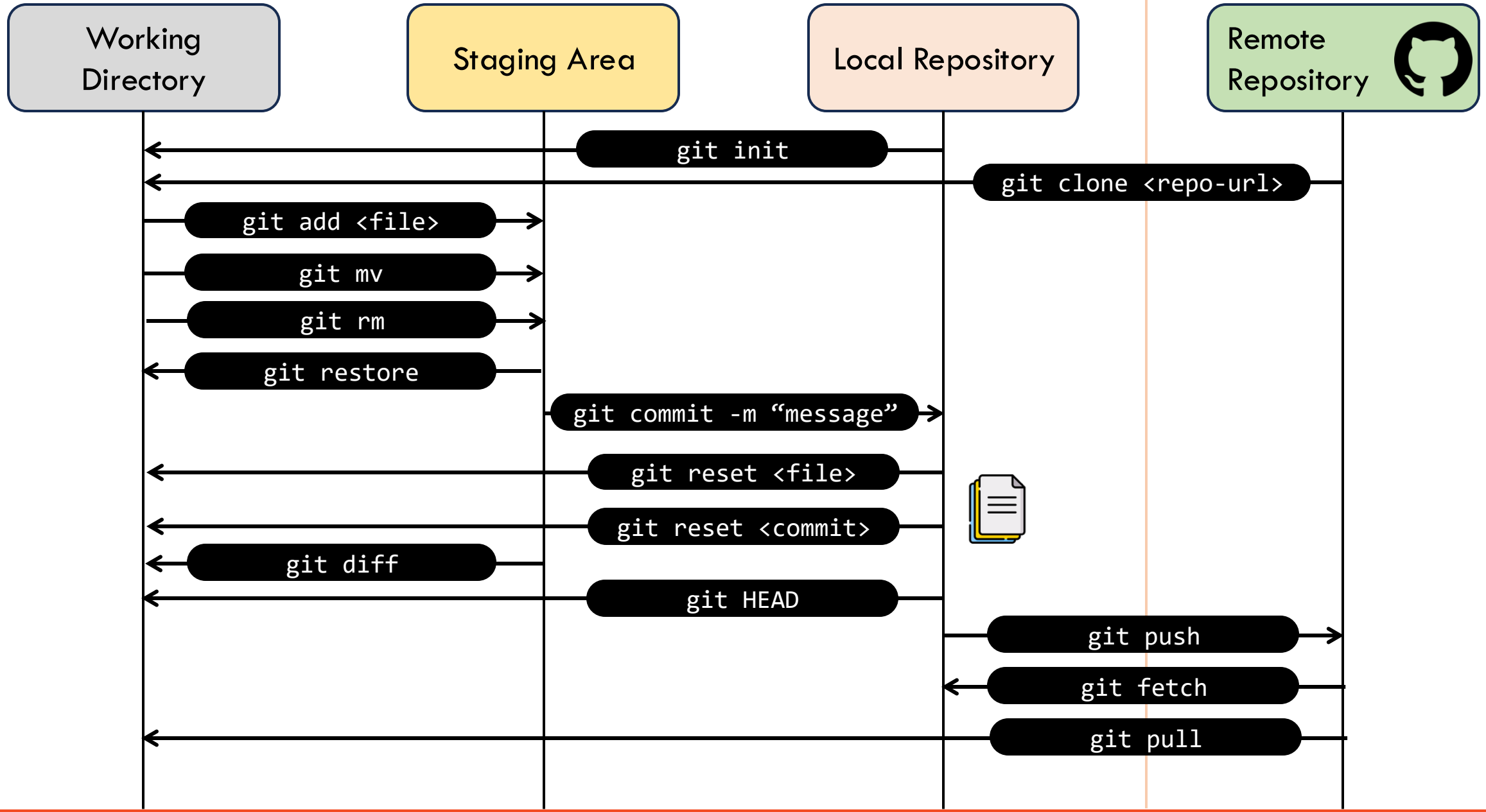
Git & GitHub Workflow

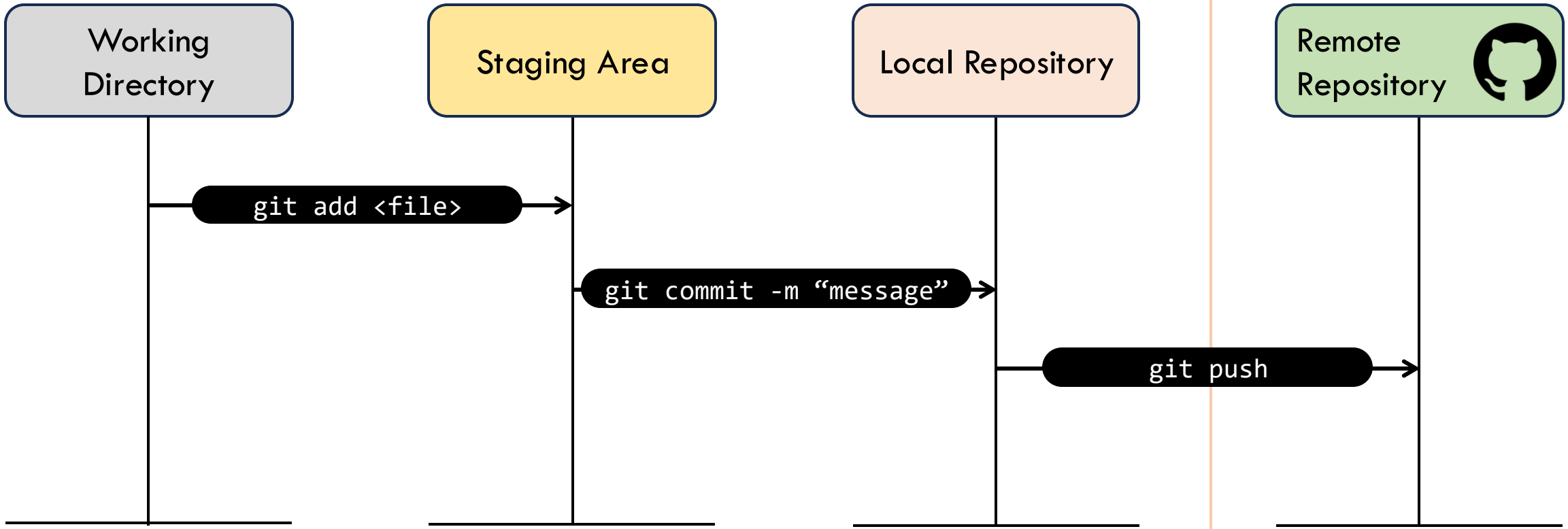












Demo

Project Walkthrough with Git & GitHub

Git & GitHub Project Workflow

Simple website as base project

Initialized Git in project folder

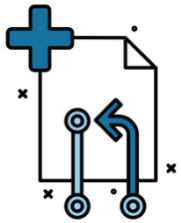
Staged & committed initial files

Created remote repo on GitHub

Linked local repo to remote

Pushed code to GitHub

Created and pushed README branch



Merged branch into main

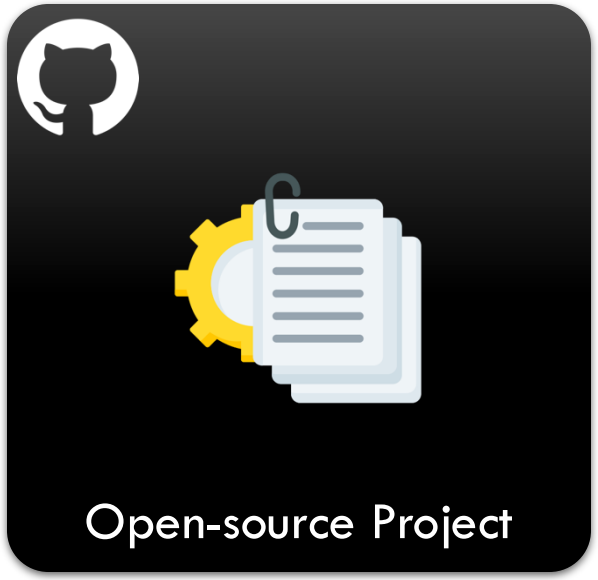
Verified changes on GitHub

Demo

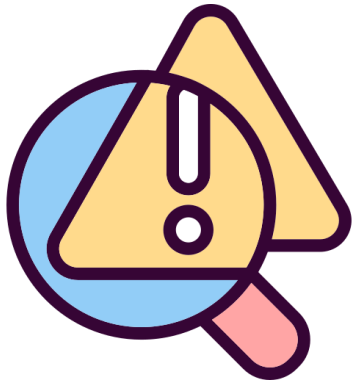
Forking in GitHub

Opening Issues & Discussions

Issues

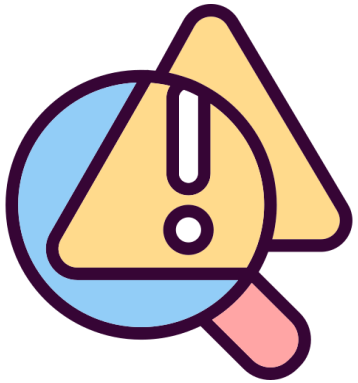


New Feature

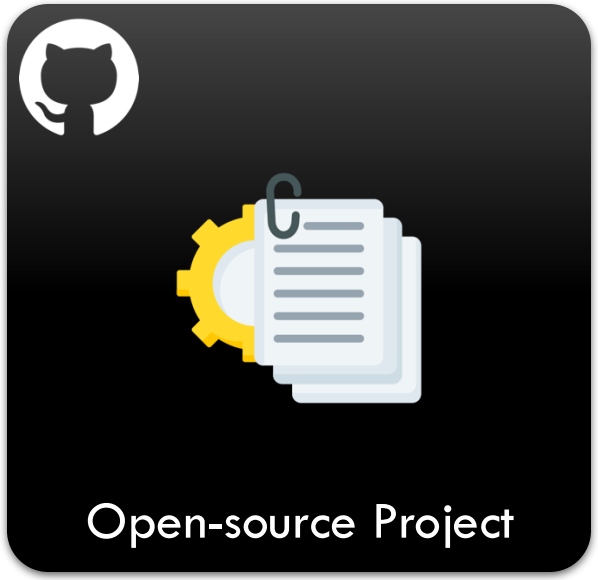


GitHub Issues

Issues

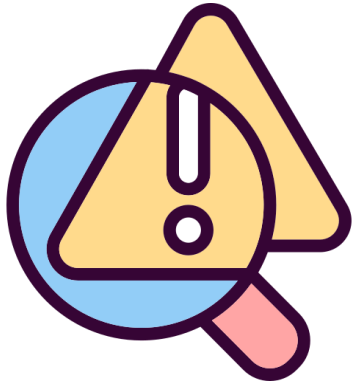


GitHub Issues



- **Bugs**
- **Typos**
- **Ideas**

Issues



GitHub Issues

To-do list

Bug Tracker

Demo

Opening Issues & Discussions

Demo

Creating & Managing Pull Requests

Demo

Repository Settings

Tagging and Versioning

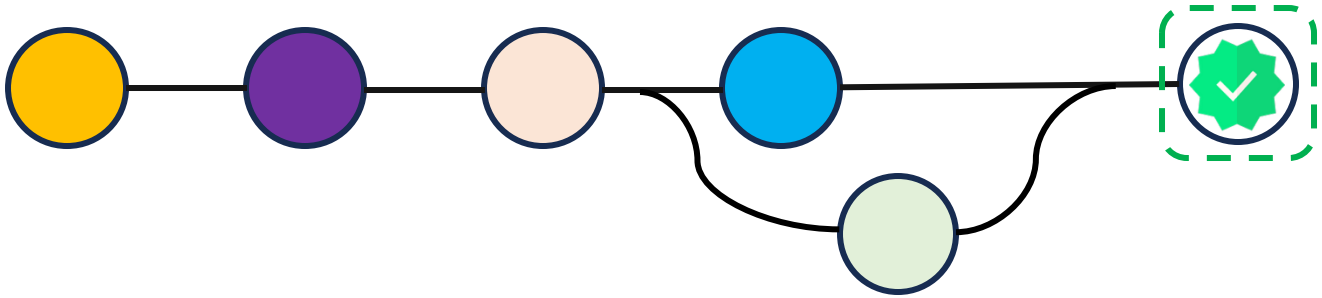
Tagging and Versioning

Section Overview

- → ***Demonstration:*** *Versioning a Project Release*
- → ***Demonstration:*** *Tags*

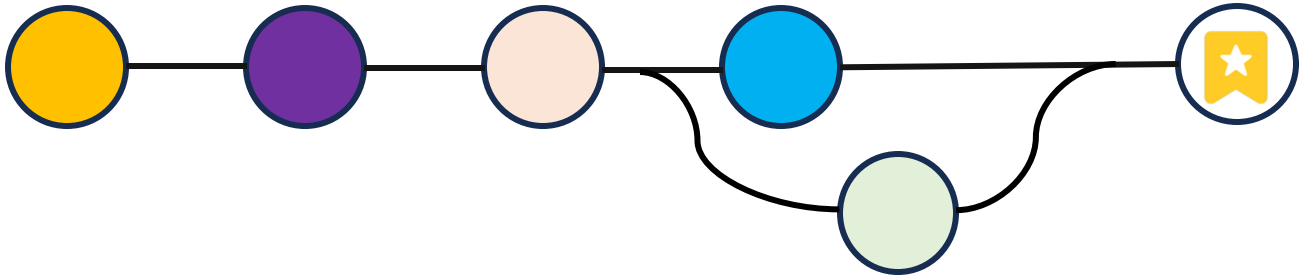
Versioning Project Release

Versioning Project Release



- ✓ Feature
- ✓ Delivering Stable Build
- ✓ Completing Task

Versioning Project Release



Versioning

- Placing a bookmark in the project's timeline to revisit that exact point later

- ✓ Tracking progress
- ✓ Sharing stable versions with others
- ✓ Packaging software for deployment

Versioning Project Release

Tags

Labels that point to a specific commit

Releases

Build on top of tags to provide additional details, notes, and downloadable assets

Demo

Versioning
Project Release

Tags

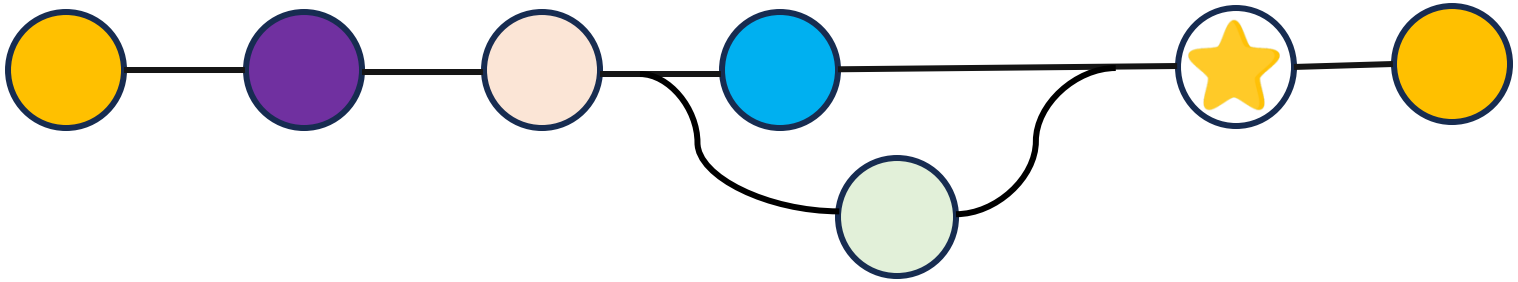
Tags

Releases 

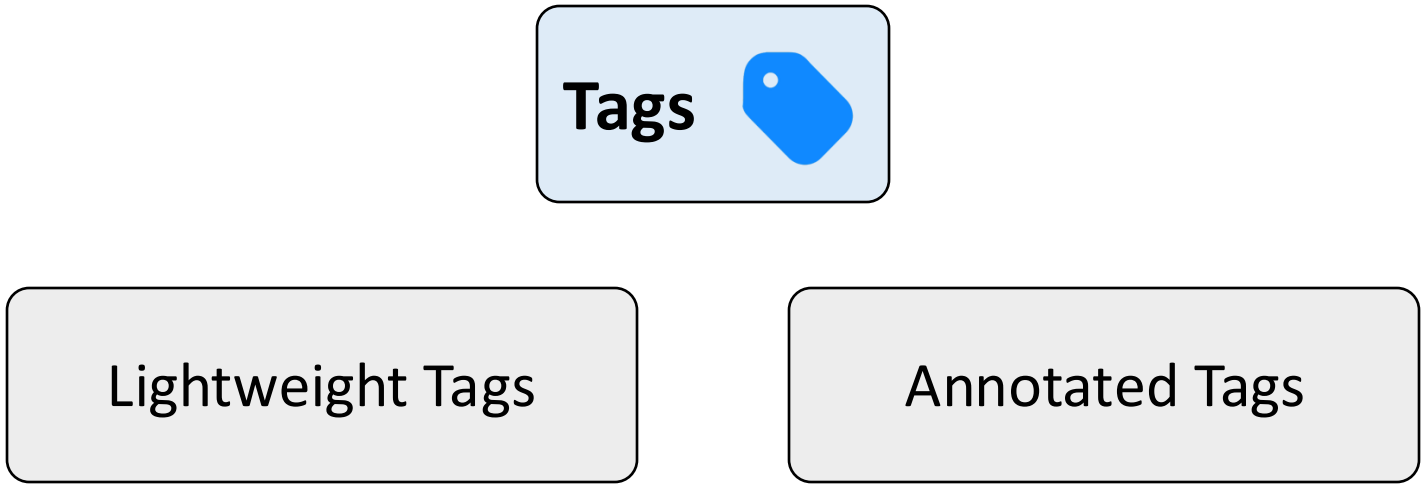
Tags

Tags

Permanent label you attach to a commit



Tags



Demo

Tags – Marking Versions

GitHub Productivity Tools

GitHub Productivity Tools

Section Overview

- → GitHub Projects
- → GitHub Wikis
- → GitHub Actions and Pages

GitHub Projects

Demo

GitHub Projects

GitHub Projects

Real-world Projects

To Do

In Progress

Done

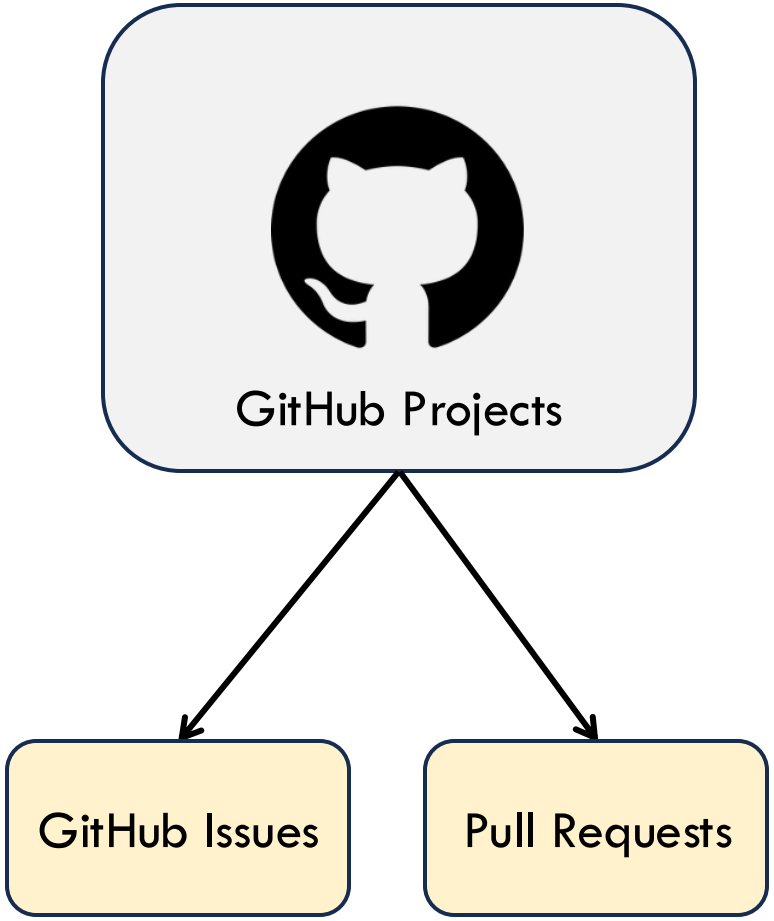
- ✓ What's pending
- ✓ What's actively being worked on
- ✓ What's completed

Built-in project management tool

GitHub Projects

Customizable Board

- Manage tasks
- Track progress
- Organize the work



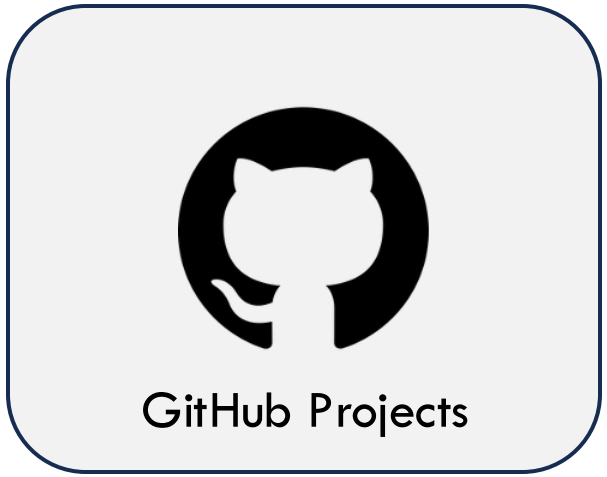
Trello

Jira

- ✓ Plan Sprints
- ✓ Track Bugs
- ✓ Organize Features

Demo

GitHub Wikis



GitHub Wikis

Dedicated space inside our repository where we can create and organize documentation

GitHub Wikis

Knowledge base for the project

README.md

- Detailed Guides
- Tutorials
- FAQs
- Technical Specifications

Support Markdown formatting

- ✓ Add Headings
- ✓ Links
- ✓ Images
- ✓ Code Snippets
- ✓ Embed Content

→ Helps you to create a clear documentation for the growth of any open-source project

→ Makes repository more welcoming to new contributors

Demo

GitHub Actions, Pages, Security and Insights

GitHub Actions

GitHub Pages

GitHub Security

GitHub Actions

Automation & CI/CD Feature

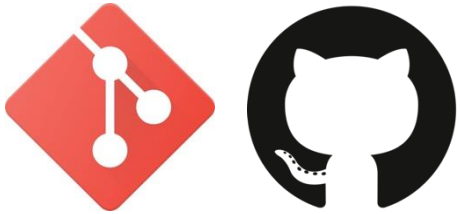
Allows to define workflows that run automatically whenever something happens in the repository

GitHub Actions

- ✓ Run tests whenever code is pushed
- ✓ Build our project automatically
- ✓ Deploy our application

GitHub Actions

Instead of manually testing or deploying, define an automated workflow once, & GitHub takes care of it



Git & GitHub Bootcamp: Build, Track & Collaborate

By – Thinknyx Technologies LLP

Course Conclusion

- ✓ *Understood version control concepts and explored how Git and GitHub streamline collaboration*
- ✓ *Differences between centralized and distributed version control systems*
- ✓ *Installed and configured Git across various operating systems and learned how Git tracks changes in the .git folder*
- ✓ *Practiced key Git operations such as committing, branching, merging, resetting, and reverting changes*
- ✓ *Connected local repositories to GitHub and collaborated using pull requests, issues, and discussions*
- ✓ *Hands-on experience with real-world workflows, on forking, resolving merge conflicts, managing releases, and using GitHub tools like Projects, Wikis, Actions, and Pages*

Follow us on:

- @thinknyx
- @thinknyx
- @thinknyx-technologies
- @thinknyx
- @thinknyx-technologies