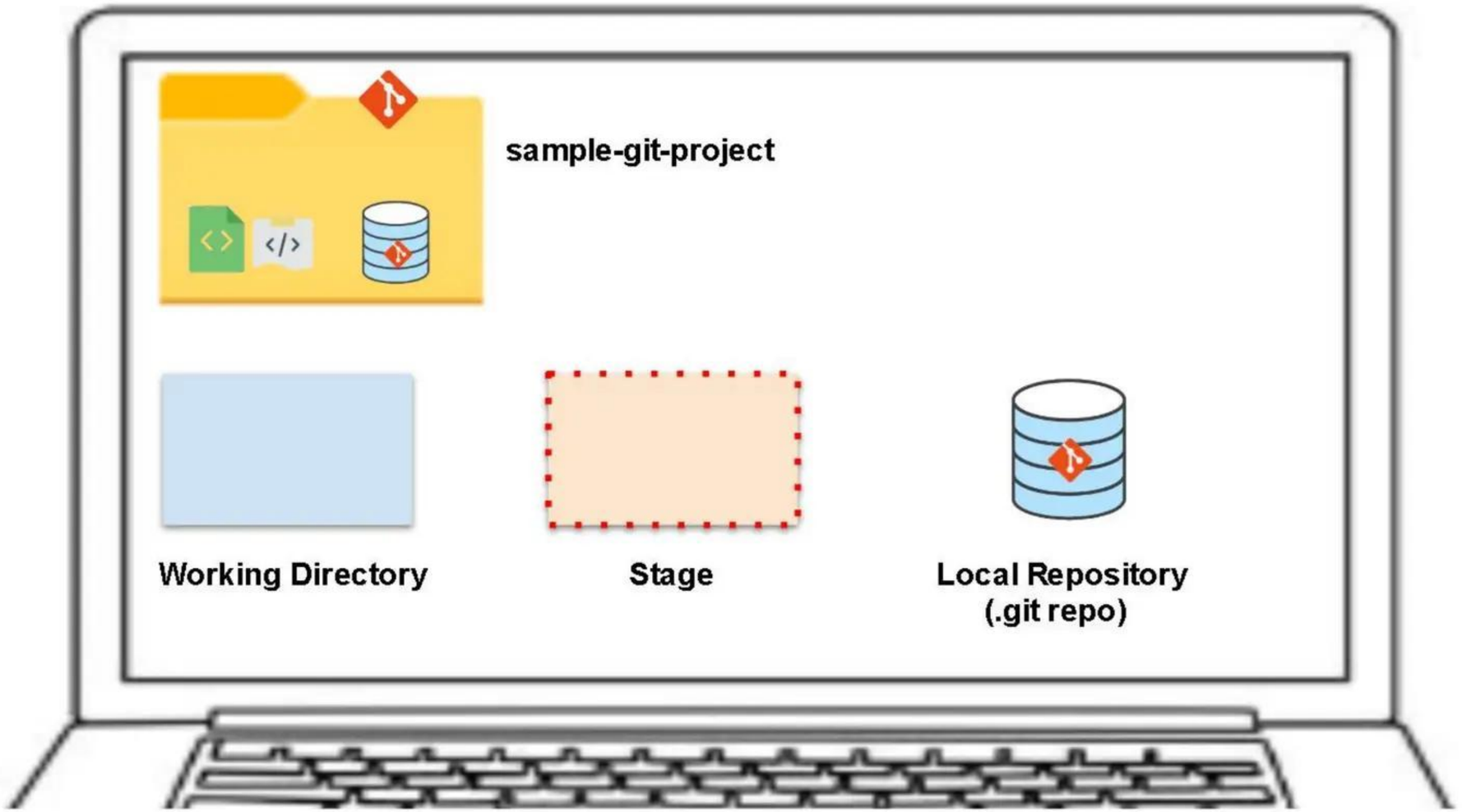
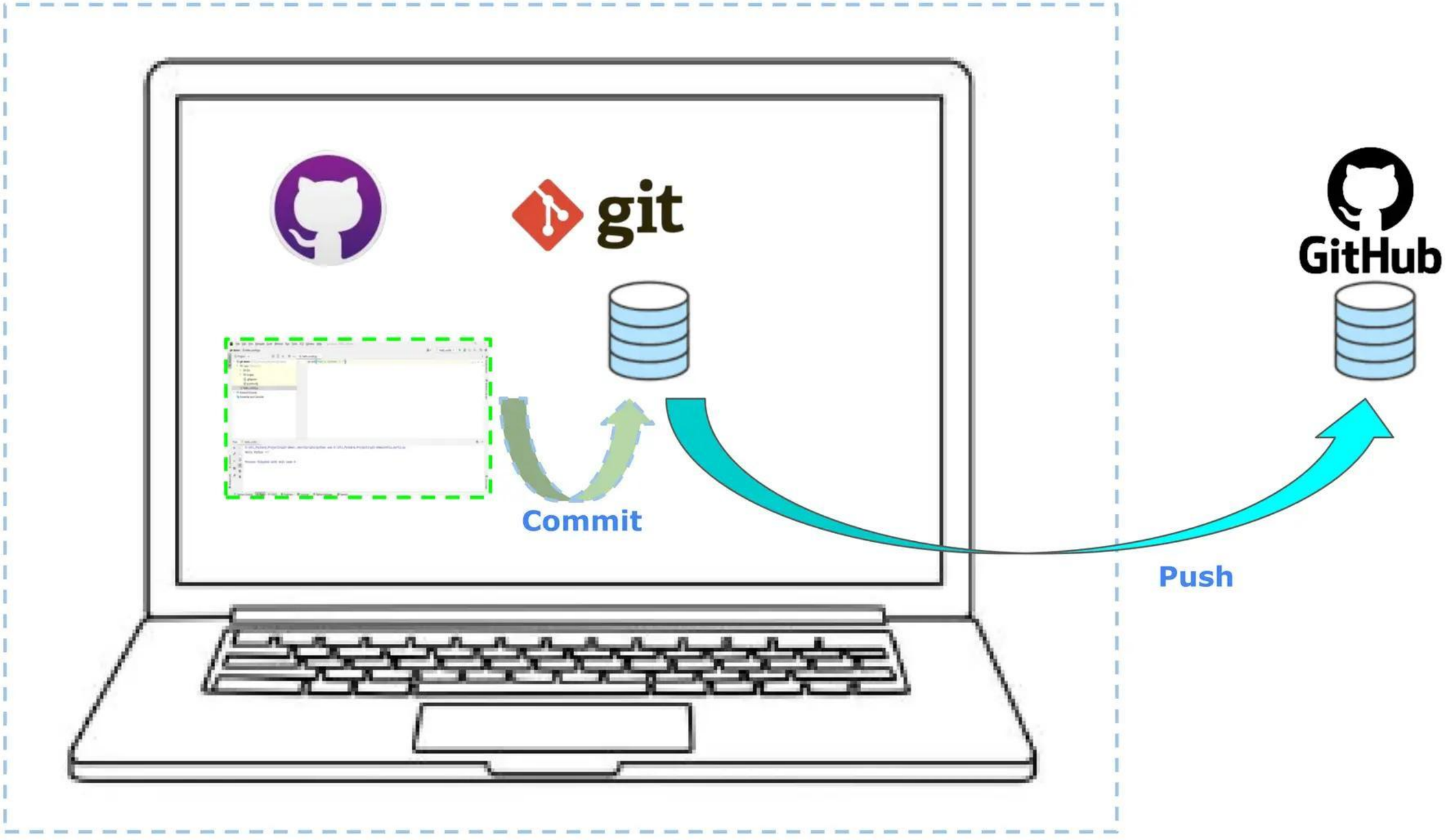
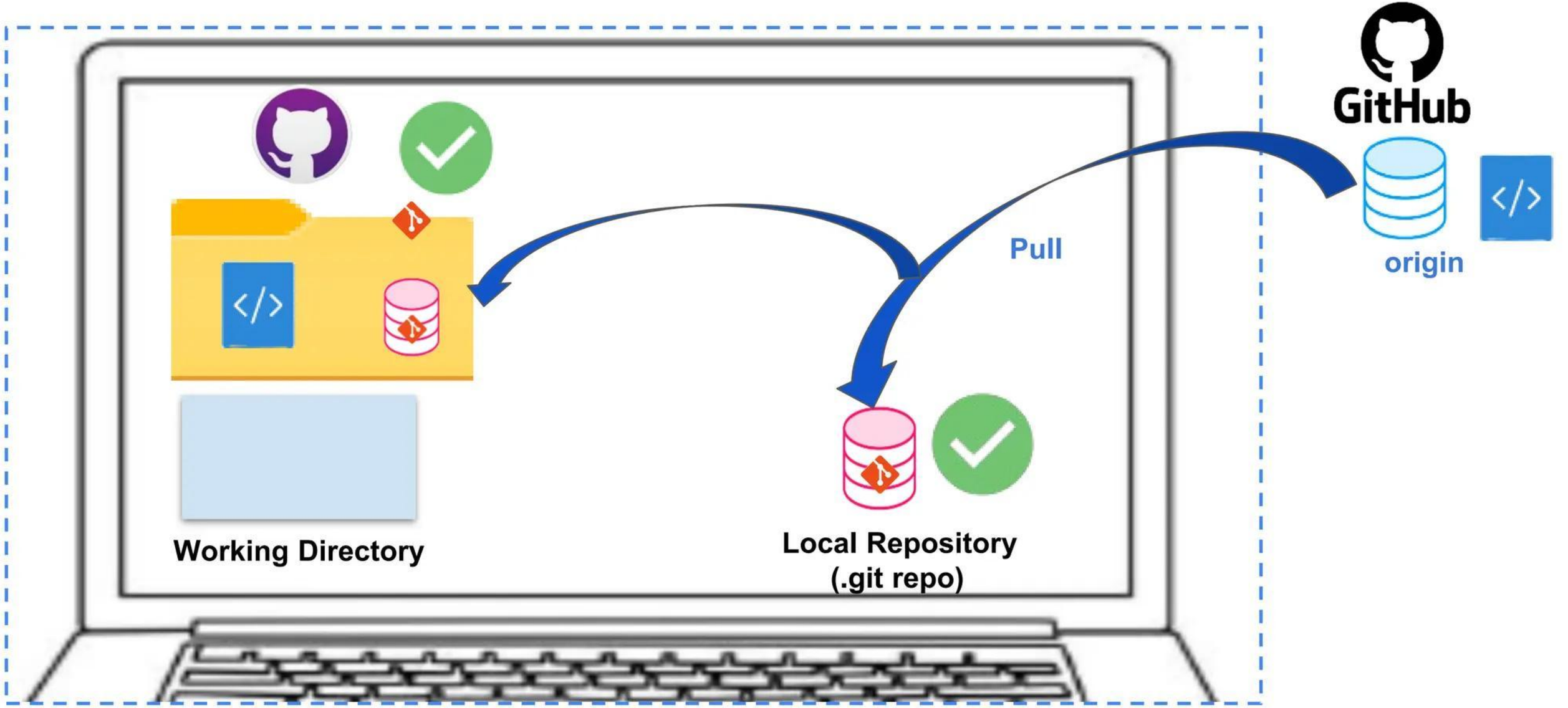
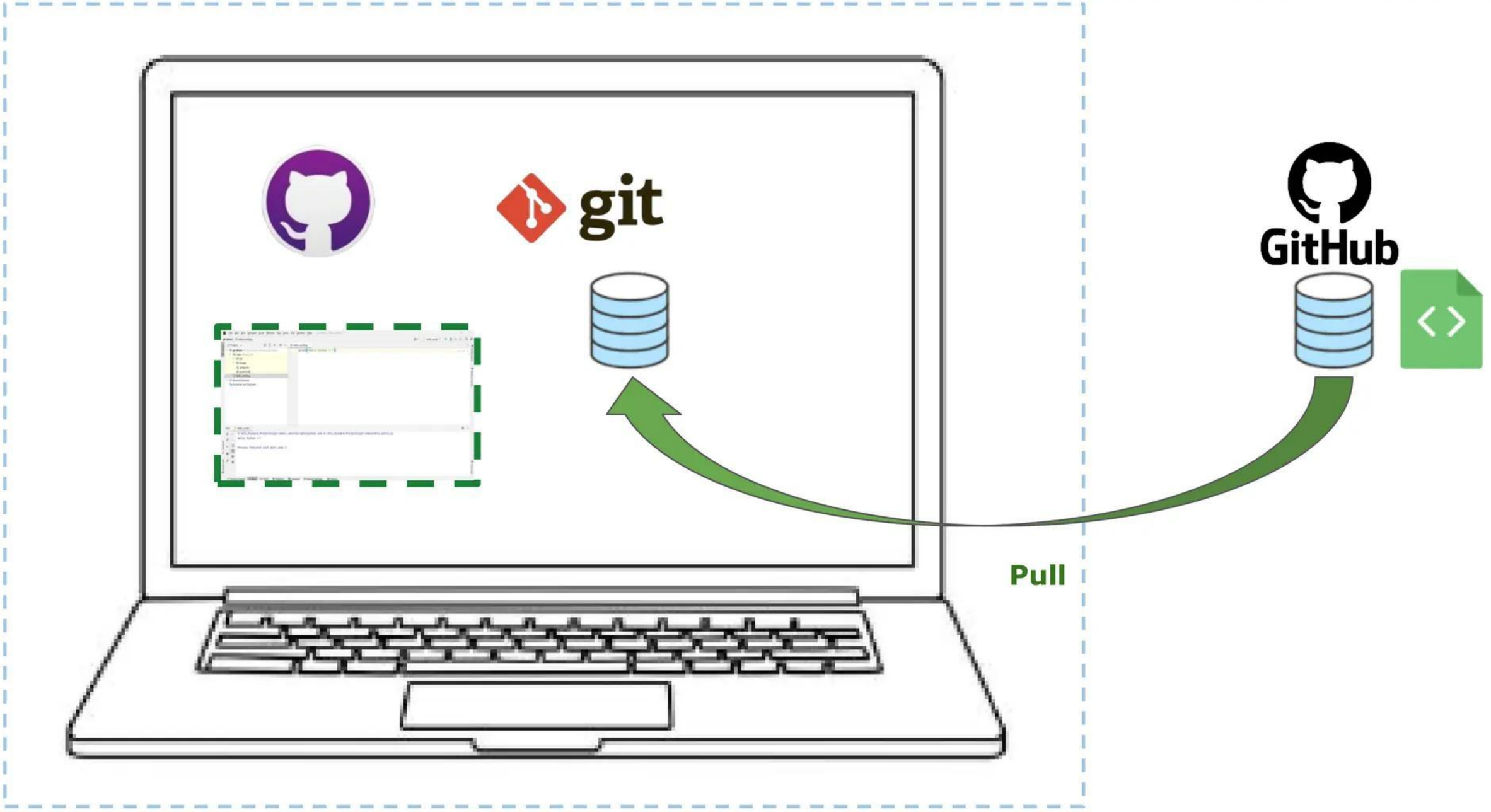

sample-git-project
Working Directory
Stage
Local Repository
(.git repo)

GitHub


git
GitHub
Commit
Push


GitHub
origin
Pull
Working Directory
Local Repository
(.git repo)


git
GitHub
Pull

# Stash

Tags are objects used to label specific points in the commit history of a repository

Frequently used to mark software release versions (v1.0,v1.1,v2.0 etc.)

Provide a way to reference specific commits

Two Types

- **Lightweight** and **Annotated**

# Clone

**Creates a local copy of the entire repository**

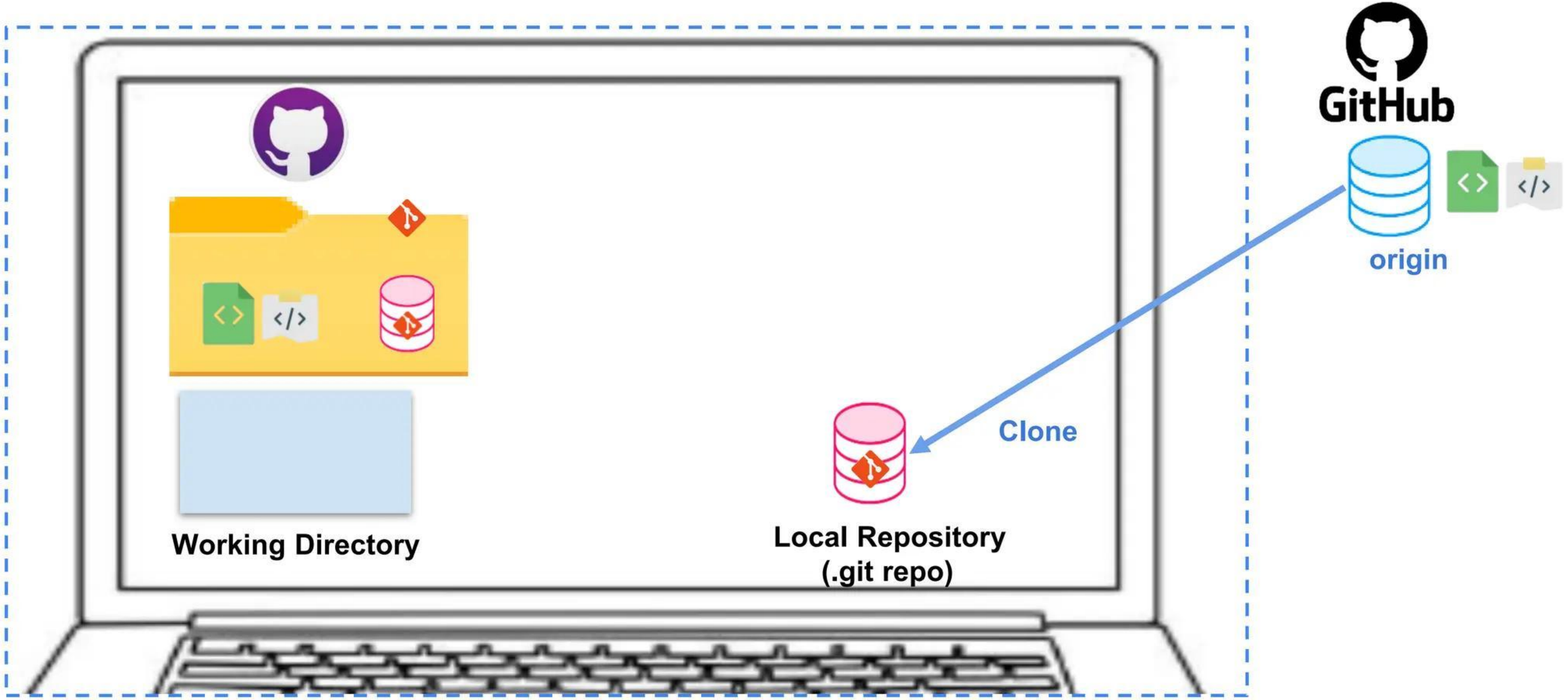
- Files ,Commit History ,Branches and Tags

**Cloning is done from an existing remote repository**

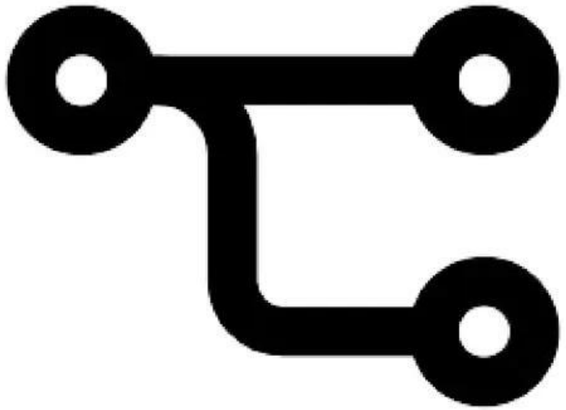
- Usually called **origin**
- **Permissions** are required

**Repo URL required**

- Can be obtained from the hosting platform


GitHub
origin
Clone
Working Directory
Local Repository
(.git repo)

# Fork



Copy of a Repository

Does not impact the original repository which is commonly referred as **upstream repo**

Used for contributing code to a repository where you are not an authorized contributor

Commonly used in open source projects

Changes can be merged back into original repository using pull requests

# Clone vs Fork

Both are used to create a copy of an existing repository

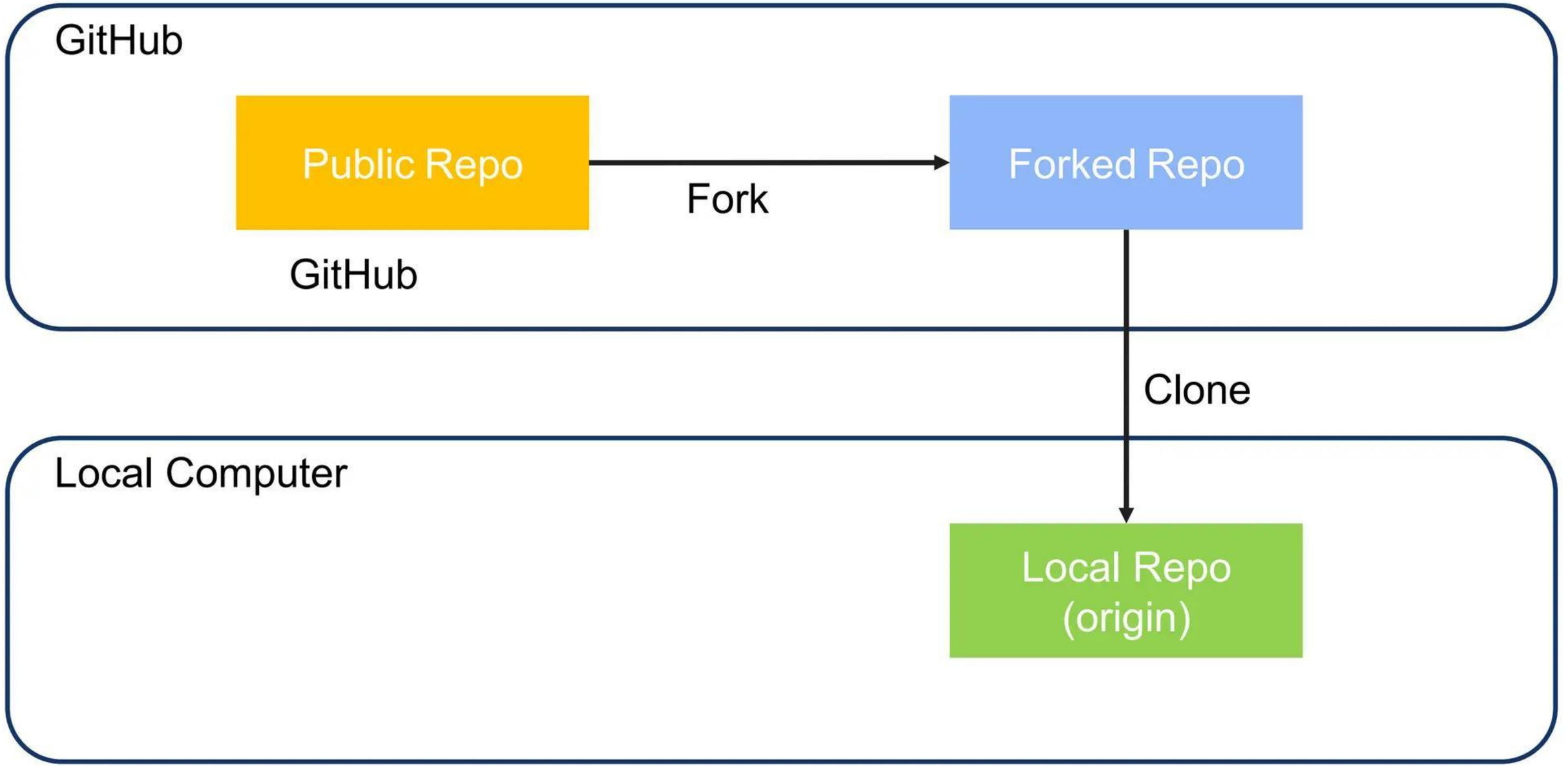
Use **Clone** to make a local copy of an existing repository

Use **Fork** to make a copy of the repository on GitHub

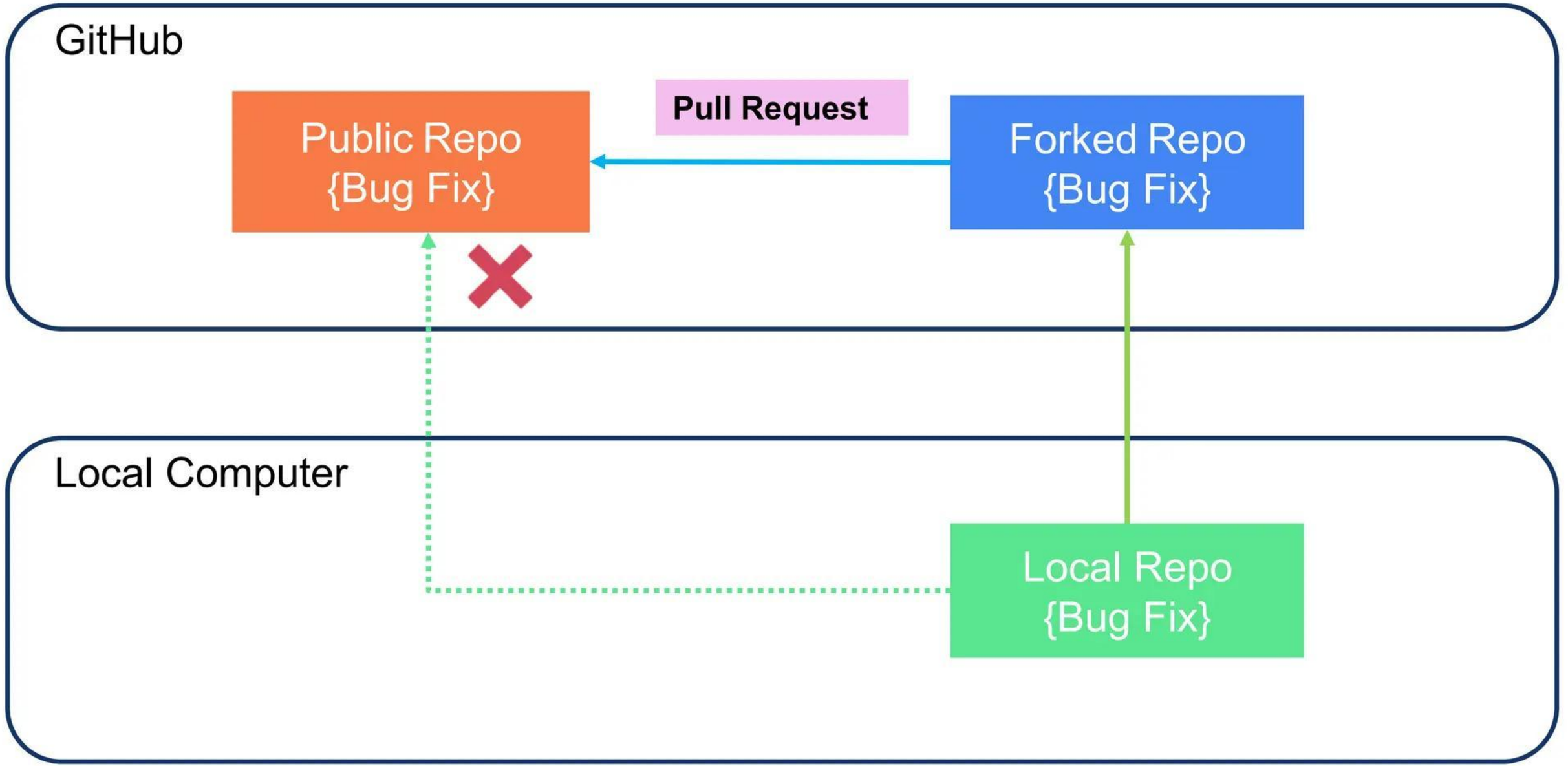
On GitHub, once you create a Fork of a repository ,you have your own **public copy** of the repository in GitHub.

Post Fork creation , to work with the code, **you need a local copy so you have to make Clone of the Fork**. Once you clone, you have a local repo on your computer which is configured with a remote repo named Origin on GitHub.

# Fork Overview



# Fork Overview



# Tags


Tags are objects used to mark or label specific commit in the commit history of a repository so that you can refer them in future

For example, you can tag a commit that corresponds to a release version, instead of creating a branch to capture a release snapshot.

Frequently used to mark software release versions (v1.0,v1.1,v2.0 etc.)

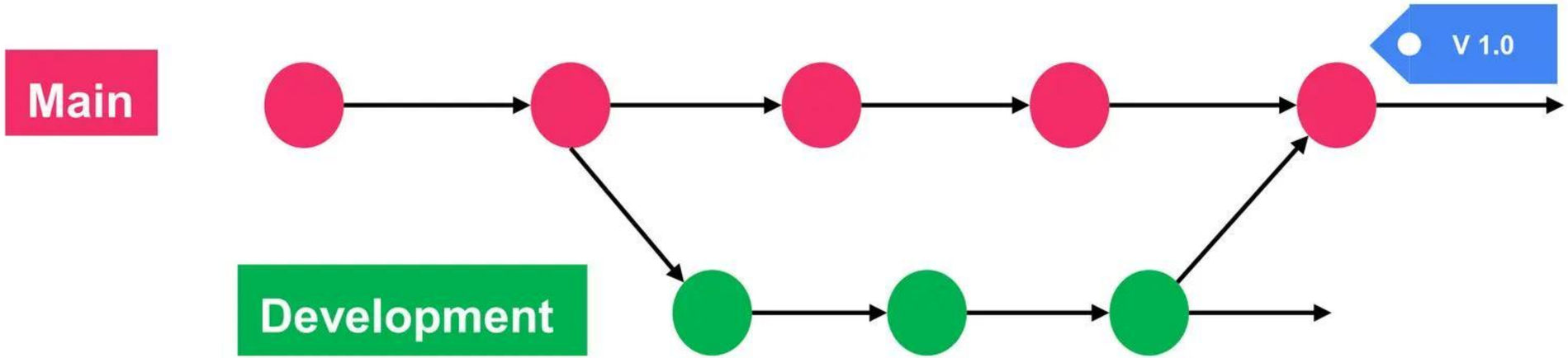
Two Types

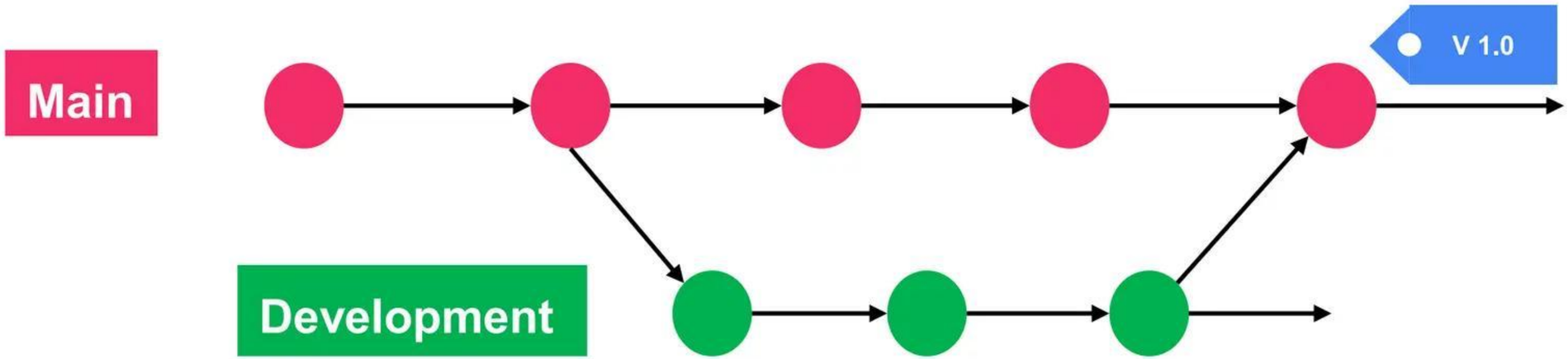
- **Lightweight** and **Annotated**

# Types of Tags

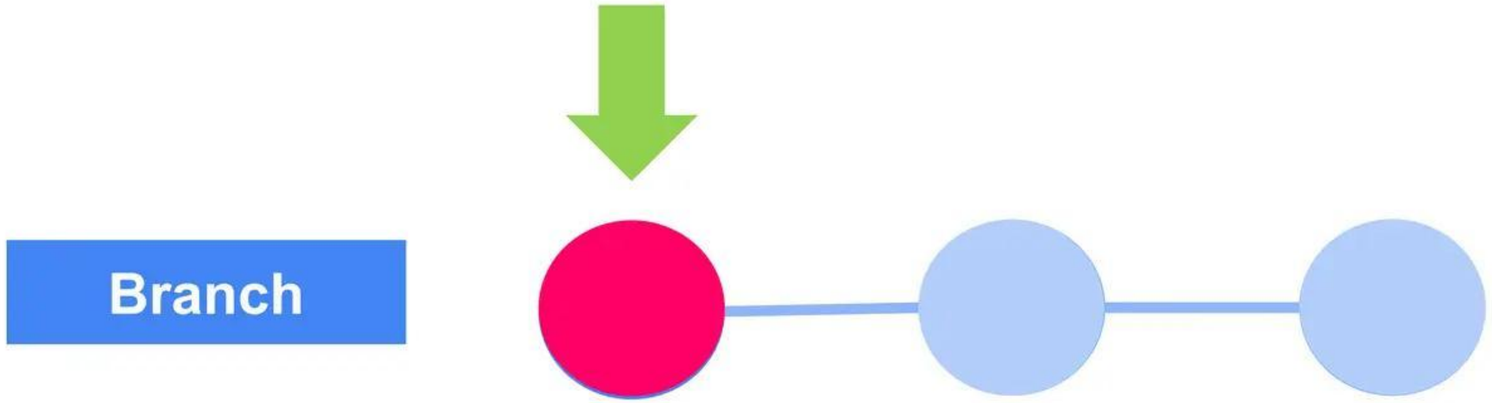
- **Lightweight Tags :** Points to a specific commit, includes tag name

- **Annotated Tags :** Points to a specific commit and includes metadata such as tag name ,tagger name ,message ,date

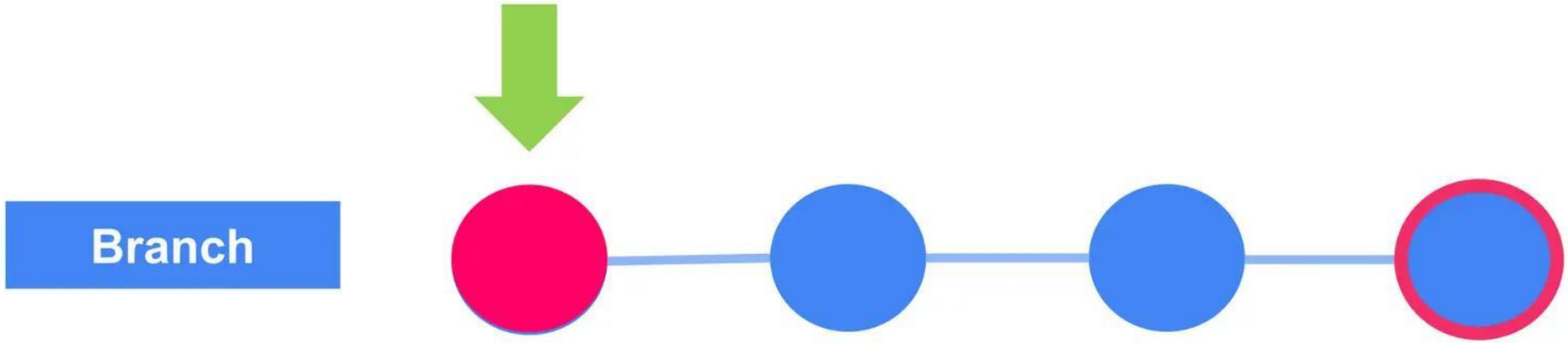



Main
V 1.0
Development

# Resetting to commit in GitHub Desktop



# Reverting a commit in GitHub Desktop



# Discarding changes in GitHub Desktop

If you have uncommitted changes that you don't want to keep, you can discard the changes.

Discarded files are moved to Trash/Recycle Bin

You can discard all uncommitted changes in one or more files

You can discard changes in one or more lines

# Undoing a commit in GitHub Desktop

Restores the changes from a commit to working directory,
so you can make further changes before re-committing.

Useful if you made a mistake in the changes you included.

**Not possible** if you have already pushed the commit to the **remote** repository.

# Resetting to commit in GitHub Desktop

Reset to Commit allows for discarding more than one commit at a time

Allows set your local history back to your latest pushed commit/non-local commit

It resets the HEAD of the repo to the commit you selected, keeping all the changes in the working directory and in the commits in between.

# Amending a commit in GitHub Desktop

It is way to modify the **most recent commit** you have made in your current branch

Helpful in scenarios like edit the commit message or including/excluding changes in the commit

Replaces the previous commit with **a new commit** to the current branch

Ideally you should only amend a commit that you haven't pushed to the remote repository.

To amend a commit that has been pushed to the remote repository, you will need to use a force push to overwrite the commit history in the remote repository.

# Reordering commits in GitHub Desktop

Reordering allows to alter a branch’s commit history to provide a more meaningful progression of commits

To reorder commits which are already pushed to remote branch ,force push is required.

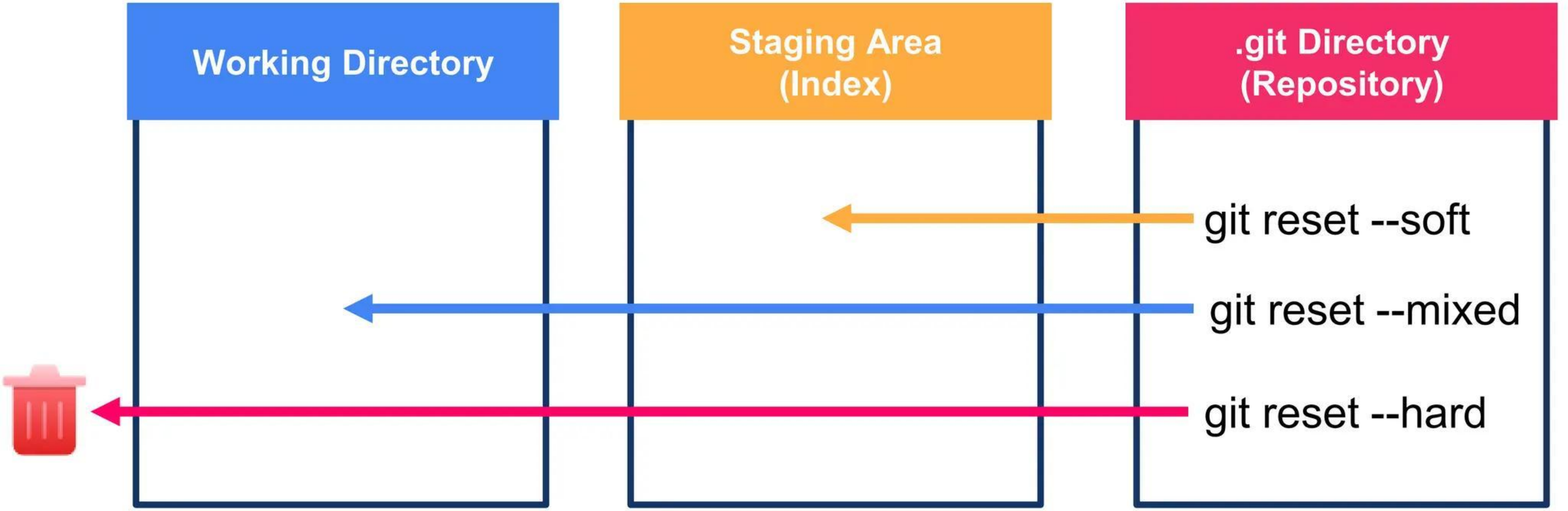
# Reverting a commit in GitHub Desktop

You can revert a specific commit to remove its changes from your branch

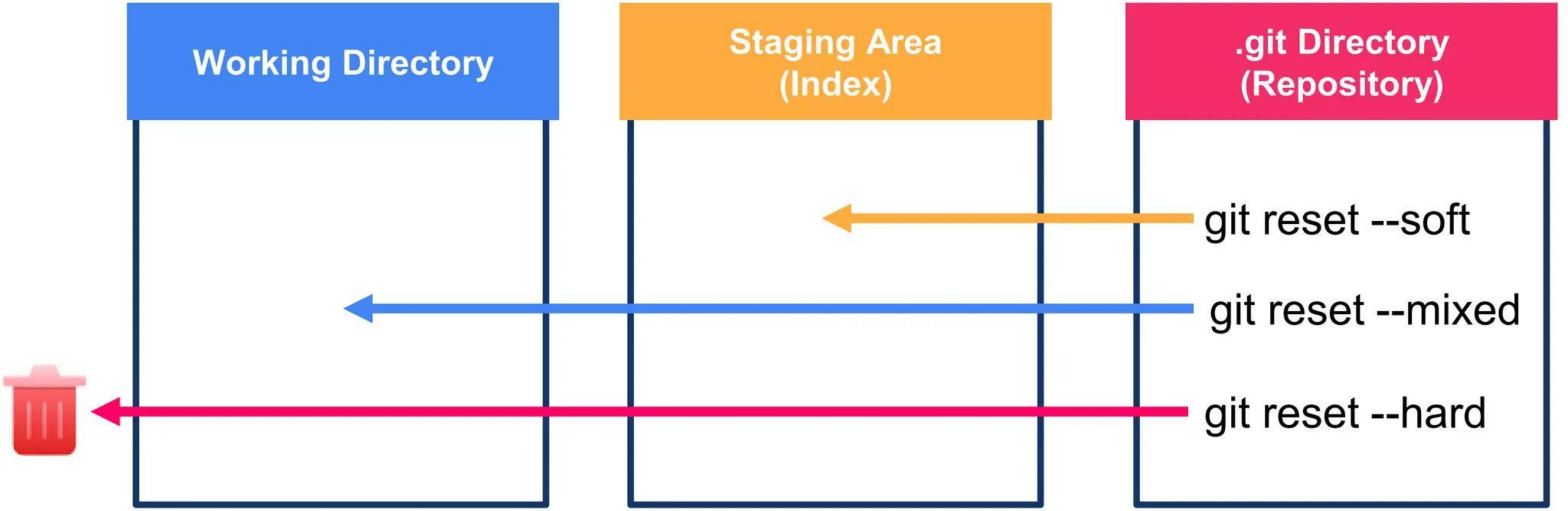
When you revert to a previous commit, the revert is also a commit. The original commit also remains in the repository's history.

When you revert multiple commits, it's best to revert in order from newest to oldest. If you revert commits in a different order, you may see merge conflicts.

# Three options of Git Reset



# Three options of Git Reset



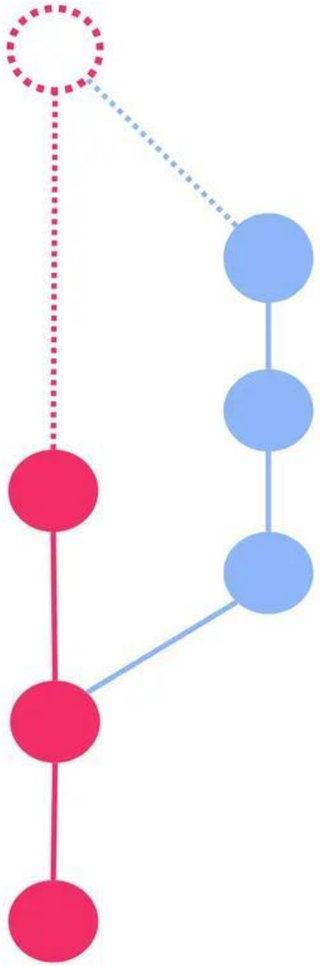
# Cherry Pick

Cherry pick enables us to choose a commit or specific commits from one branch and apply it to another branch

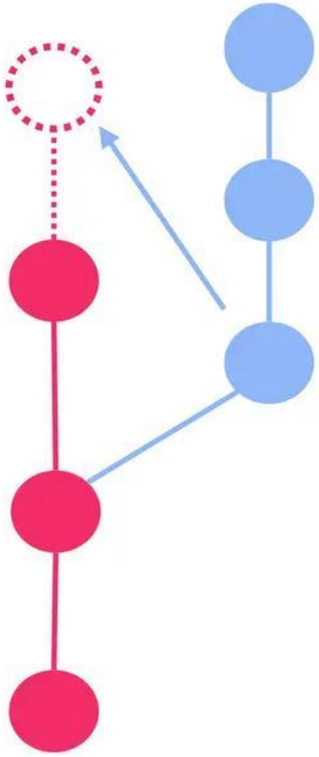
Creates copy of a commit – conceptually similar to copy and paste

Cherry pick is advanced feature

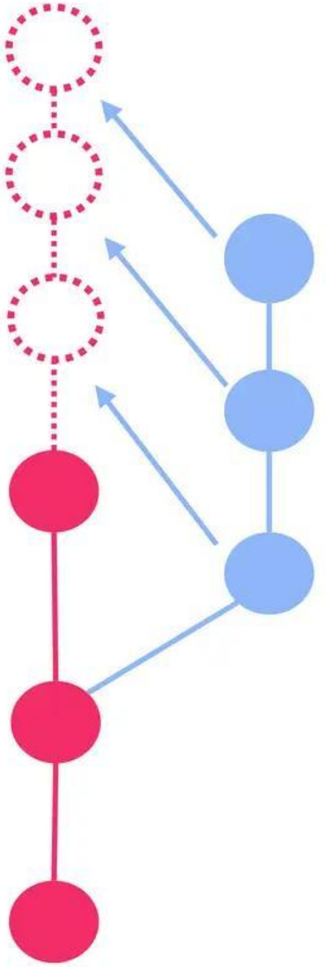
- Should not be considered as an alternative of merge
- Can cause confusion



MERGE

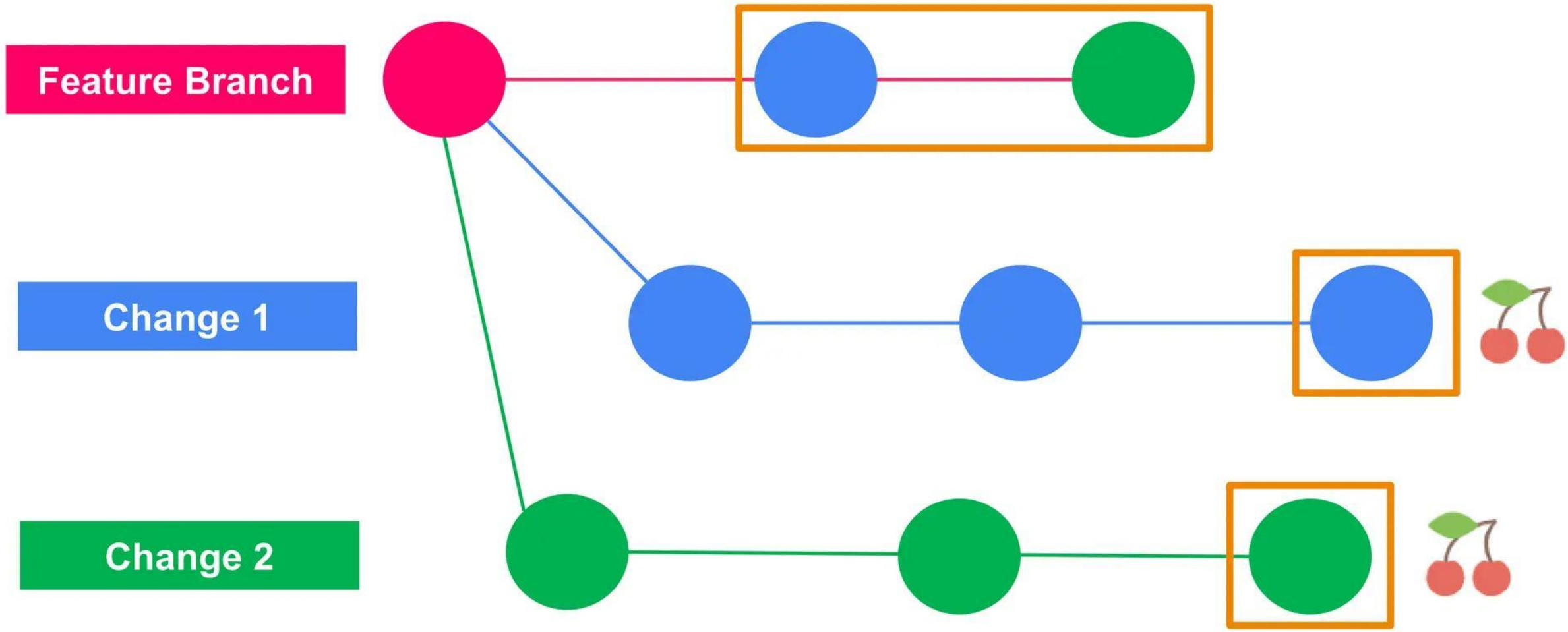


CHERRY PICK

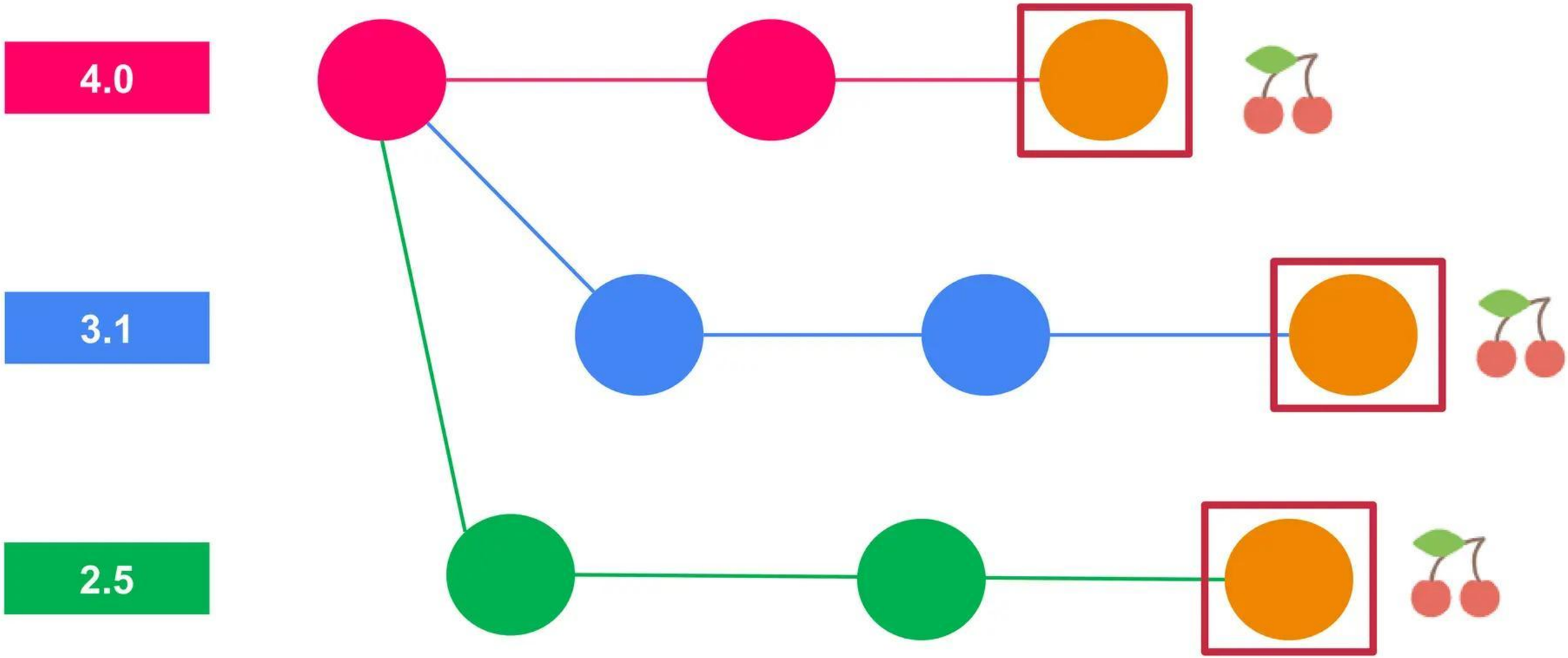


REBASE

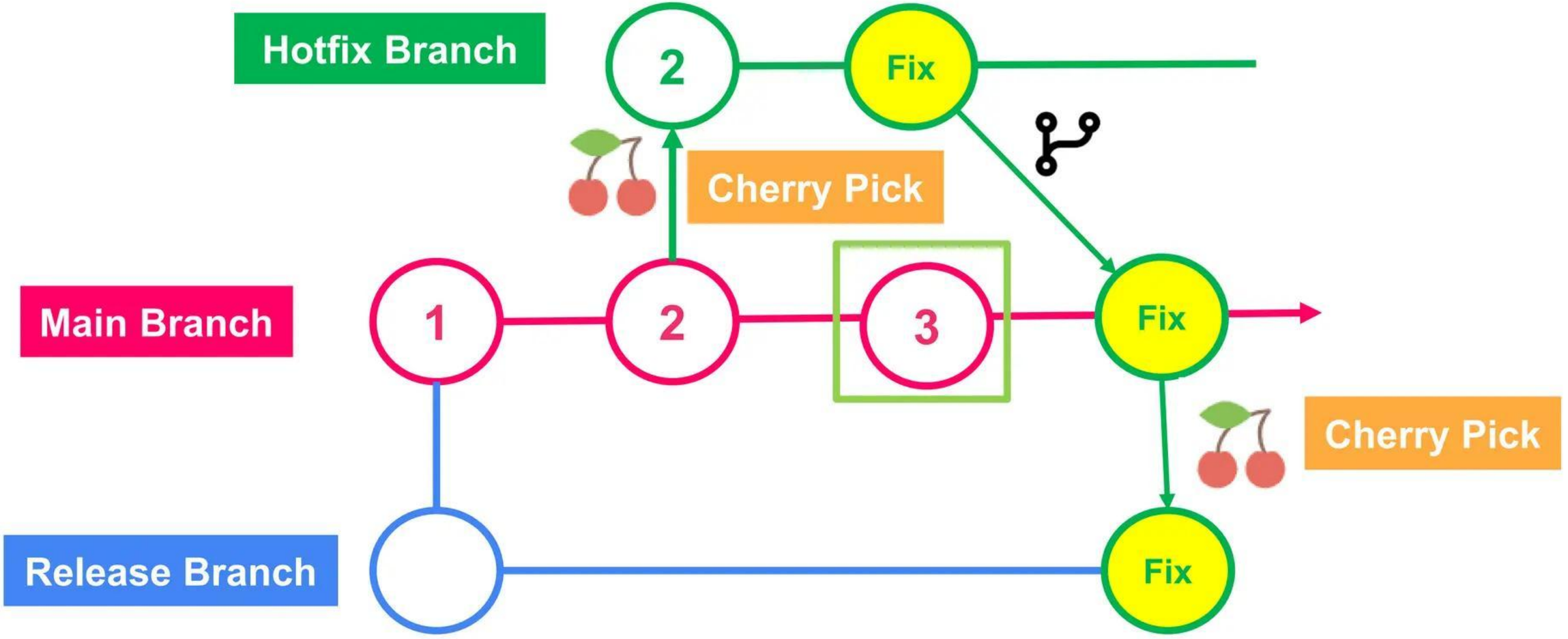
# Cherry pick to a Feature Branch



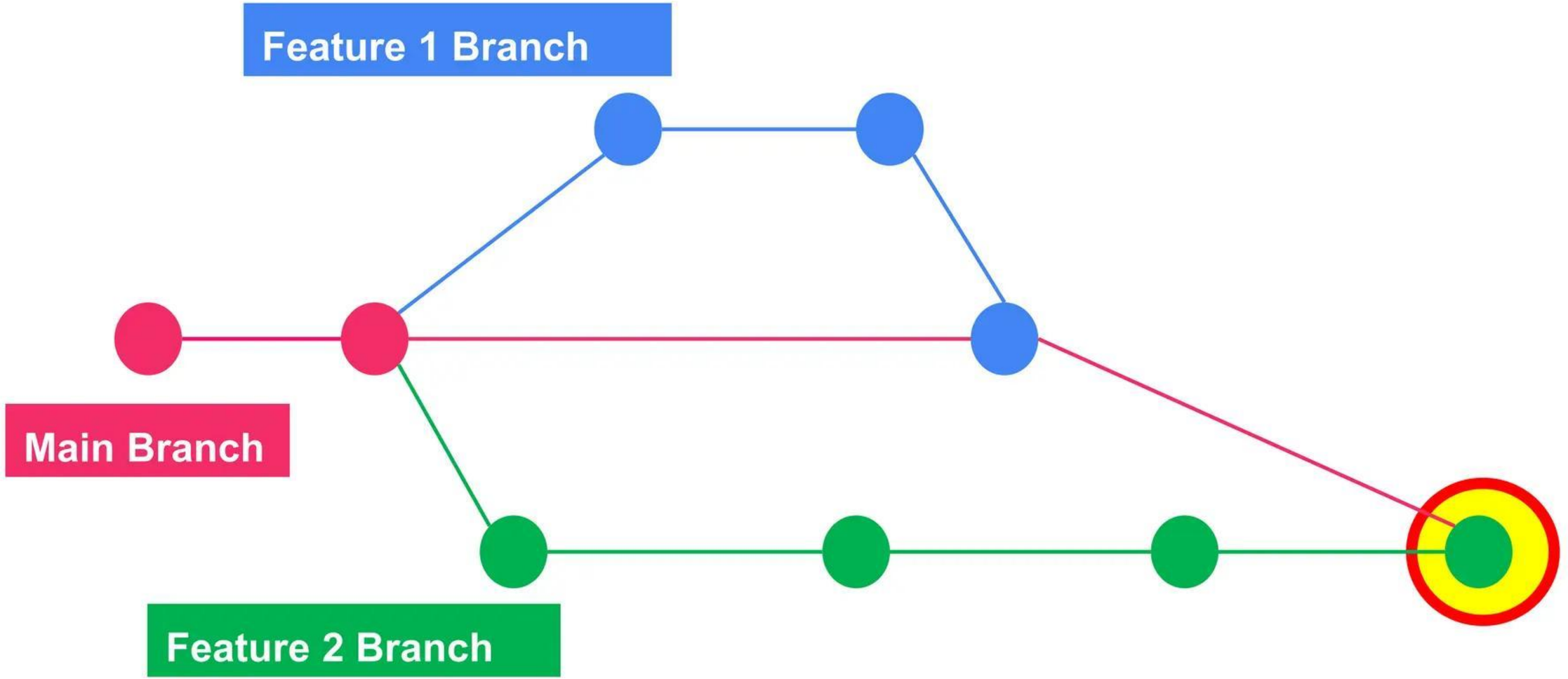
# Cherry-pick to Version Branches



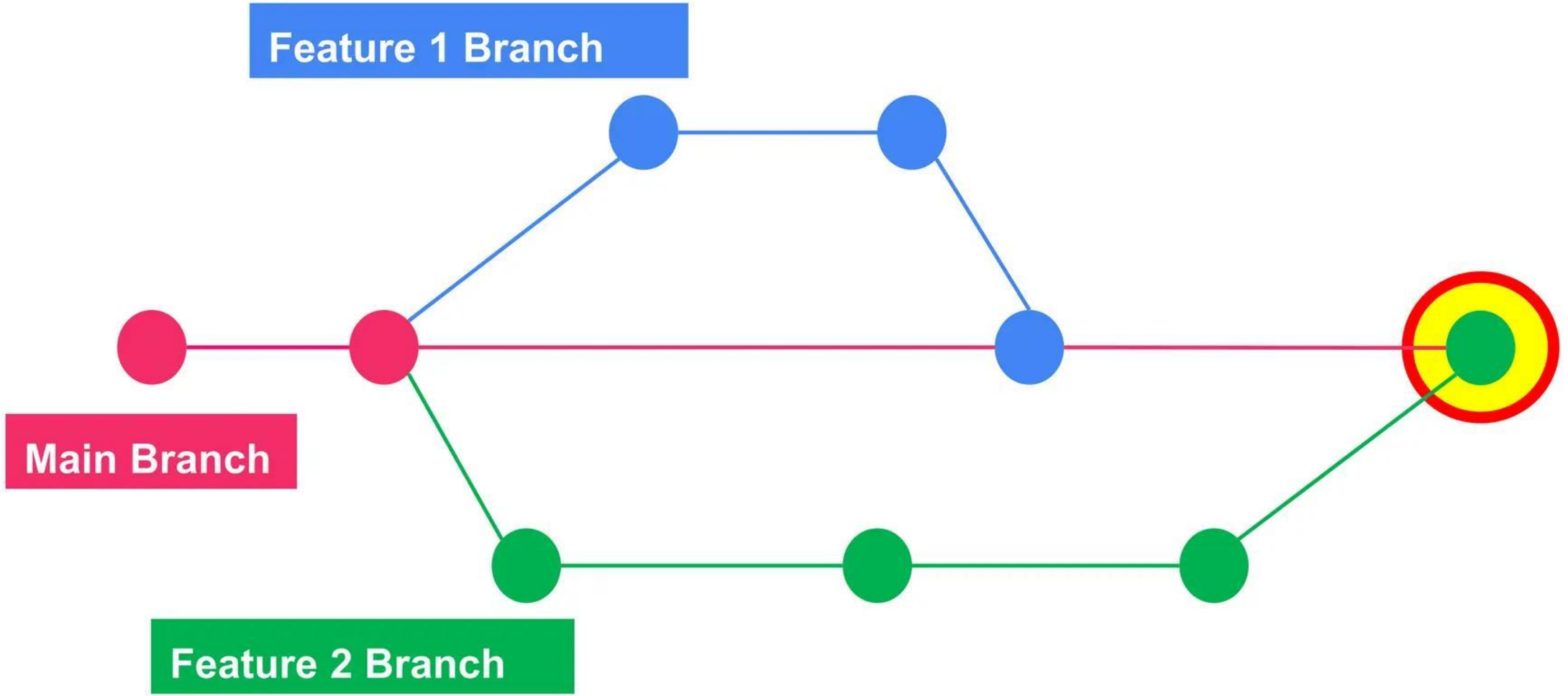
# Cherry-Picking a Hotfix to a Release Branch



# Rebase Conflict



# Merge Conflict



# Common Reasons/Situations for Merge Conflict

**Occurs when there are conflicting changes**

**Line Change Merge Conflict**

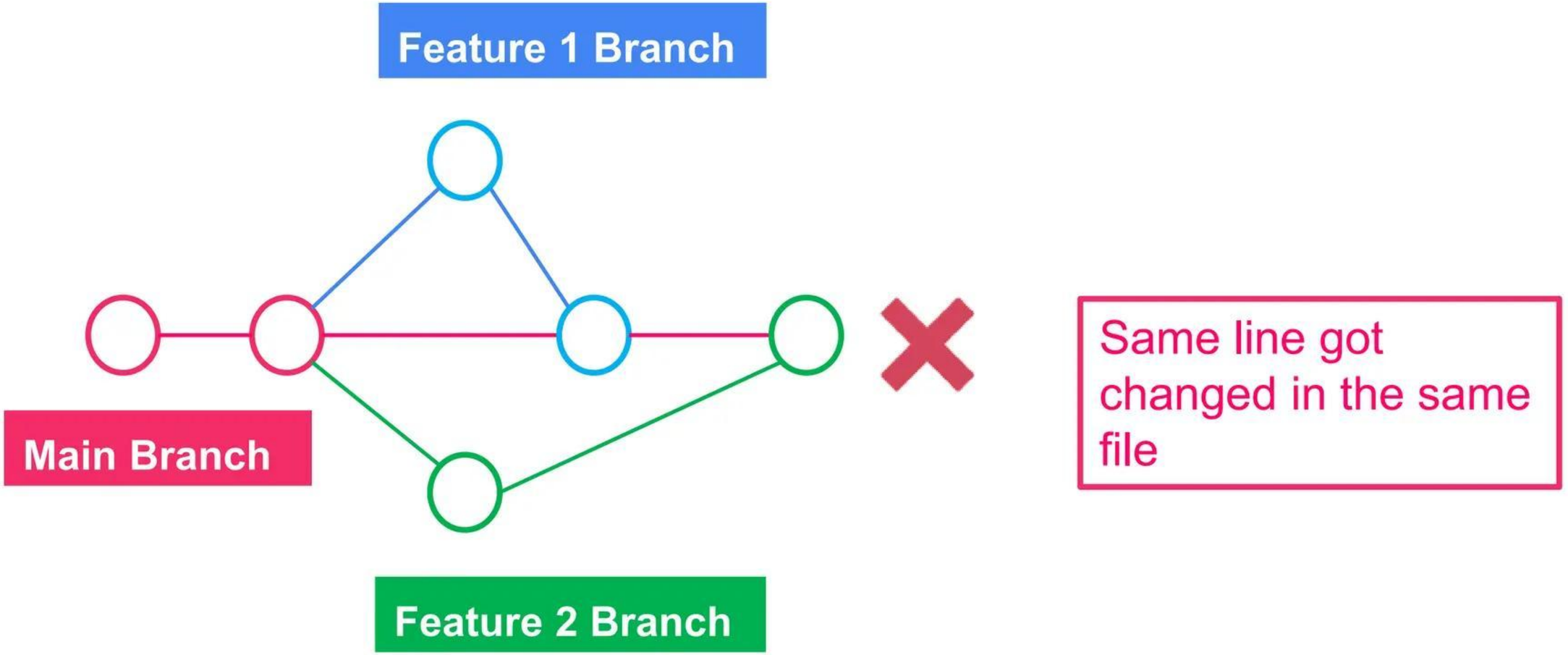
- When more than one developer changes the same line in a file and tries to merge the change to the same branch

**File Merge Conflict**

- When one developer deletes the content in a file but another developer edits the same content and tries to merge the change to the same branch

- When one developer deletes a file but another developer edits the same content and tries to merge the change to the same branch

# Line Change Merge Conflict



# Line Change Merge Conflict

**Main Branch**

calculator.py

```
1    def add(x, y):
2        """Adds two numbers"""
3        return x + y
4
```

**Feature 2 Branch**

calculator.py

```
1    def add(x, y):
2        """Adds two numbers. Added logging."""
3        print(f"Adding {x} and {y}")
4        return x + y
```

```
<<<<<<< HEAD
    """Adds two numbers"""
    return x + y
==========
    """Adds two numbers. Added logging."""
    print(f"Adding {x} and {y}")
    return x + y
>>>>>>> Feature 2
```

```
    """Adds two numbers. Added logging."""
    print(f"Adding {x} and {y}")
    return x + y
```



This is the changed code in Main branch



Separator for other file content



Changed code in the Feature 2 branch

Resolve the conflict and commit

# Merge Conflict Resolution

**Line Change Merge Conflict**

- Git resolves most differences by itself

**Conflict**

- Git needs you to decide how to merge/combine

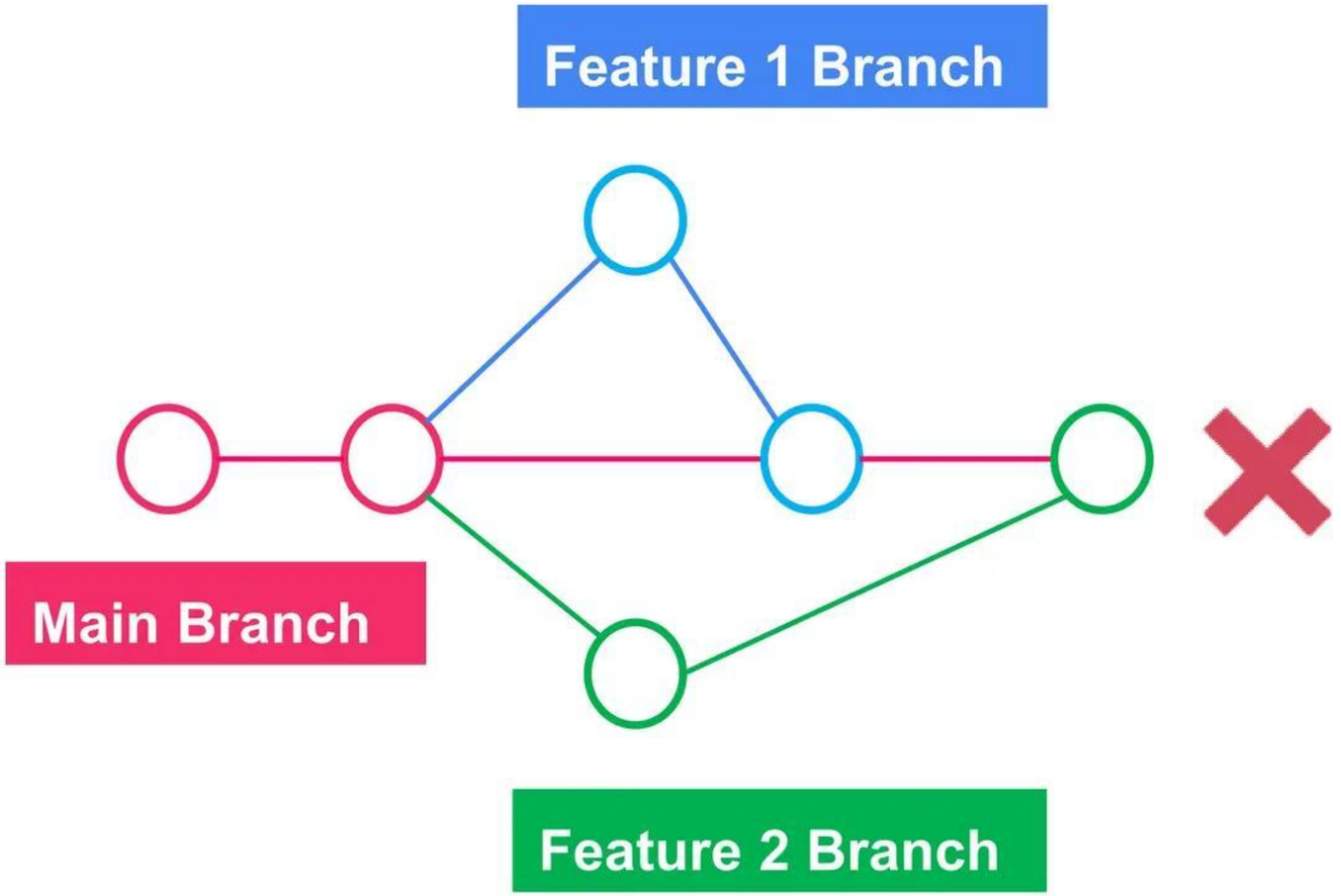
**File markers indicate conflicting changes**

- Review the **<<<<<** and **>>>>>** markers

**Edit the file to combine the content**

**Finally perform commit (merge commit)**

# File Merge Conflict




Add file with same name but different content

# Types of File Merge Conflict

## Add/add

File with same name added in both branches but the content is different

## Rename/rename

A file is renamed under different names in two branches

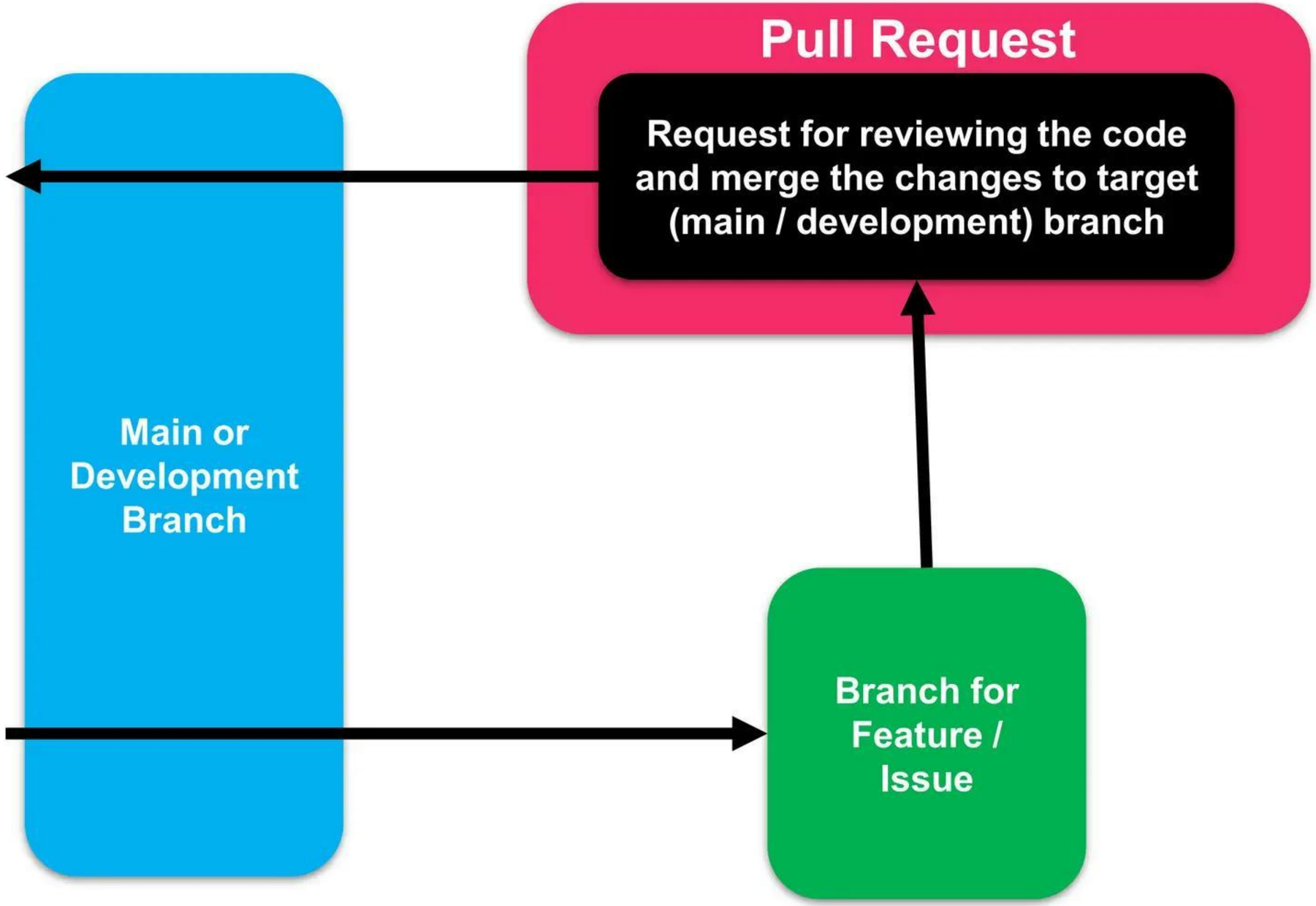
## Rename/delete

A file is renamed in one branch and deleted in another

## Modify/delete

A file is modified in one branch but same file is deleted in another branch

# What is a Pull Request



# Why is it called Pull Request ?

# Pull Request is a process

## Create

**Raise a Pull Request**

## Review

**Reviewer performs the review and provides feedback**

## Approve

**If everything looks fine, code is merged into the target branch**

# Rebase Scenarios

## Clean up of history

Cleaning up your local history before sharing a branch

## Pull without merge

Pull changes from other branch to your branch

# Merge Conflict

So what is a merge conflict and how do they happen?

It updates the base of the current branch (typically feature) by replaying its commits on top of the latest commits from the  other branch (typically main or master)

# Rebase

Rebase is used to integrate changes from one branch onto another branch

It updates the base of the current branch (typically feature) by replaying its commits on top of the latest commits from the  other branch (typically main or master)

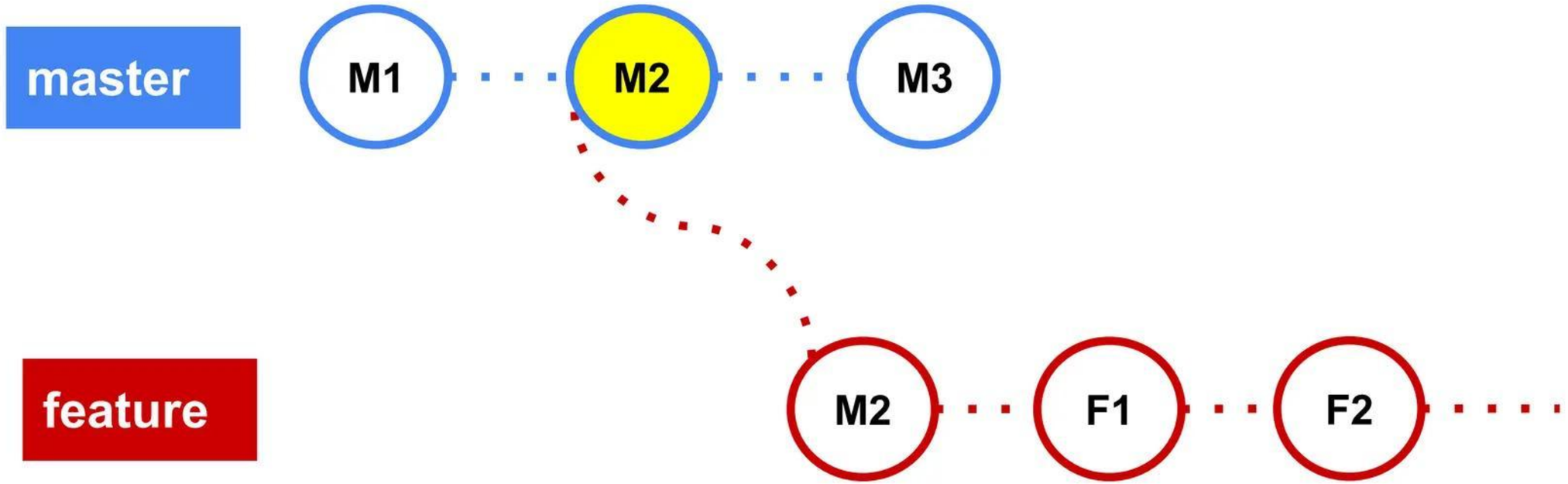
# Use Rebase carefully !!

**Never use Rebase on a branch which is shared with other developers**

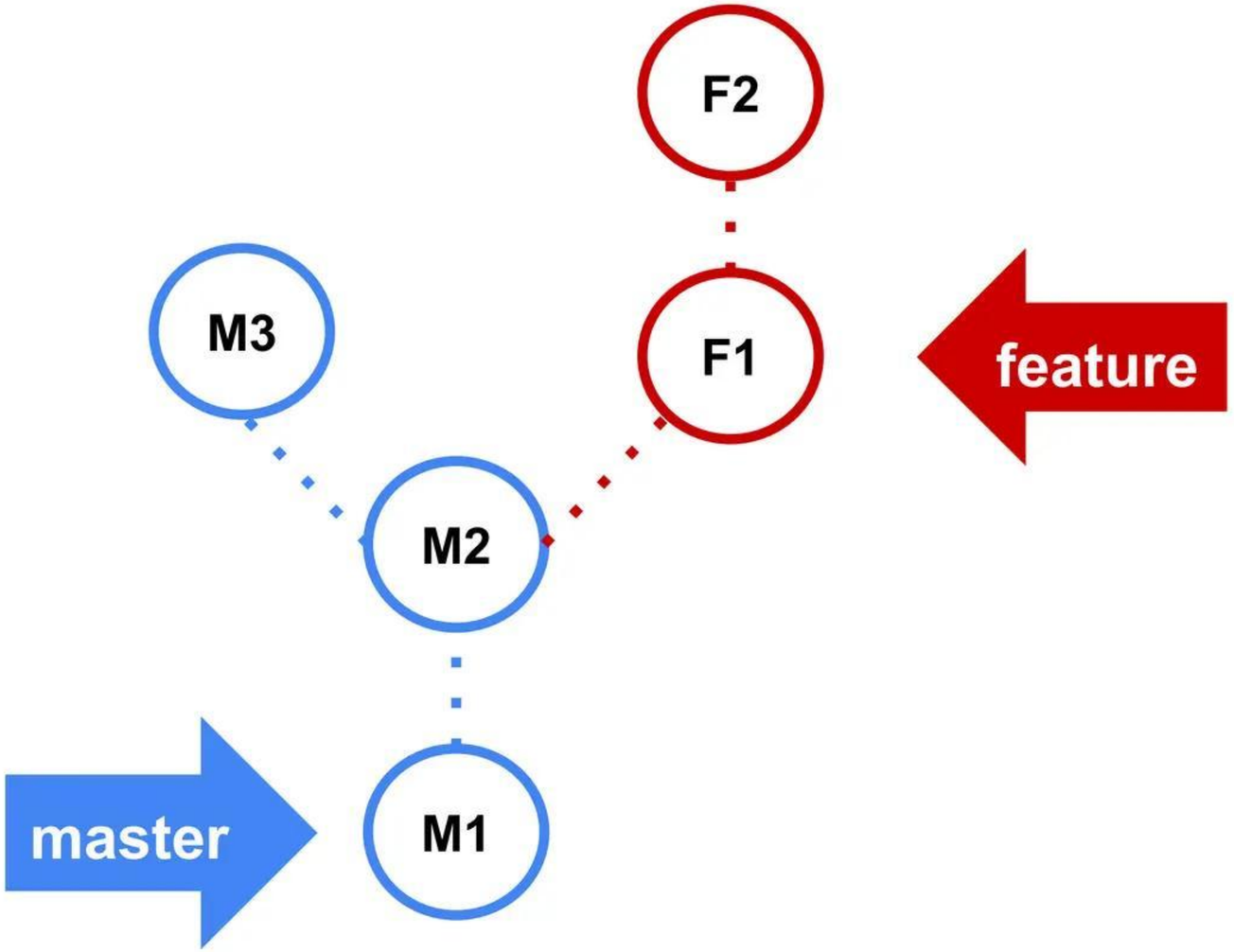
**As Rebase alters the history of the branch ,in case other developers are working ,it would be causing merge conflicts and inconsistency in the codebase.**

**Rebasing and then force-pushing (git push --force) can overwrite others' changes if they have pushed commits to the same branch.**

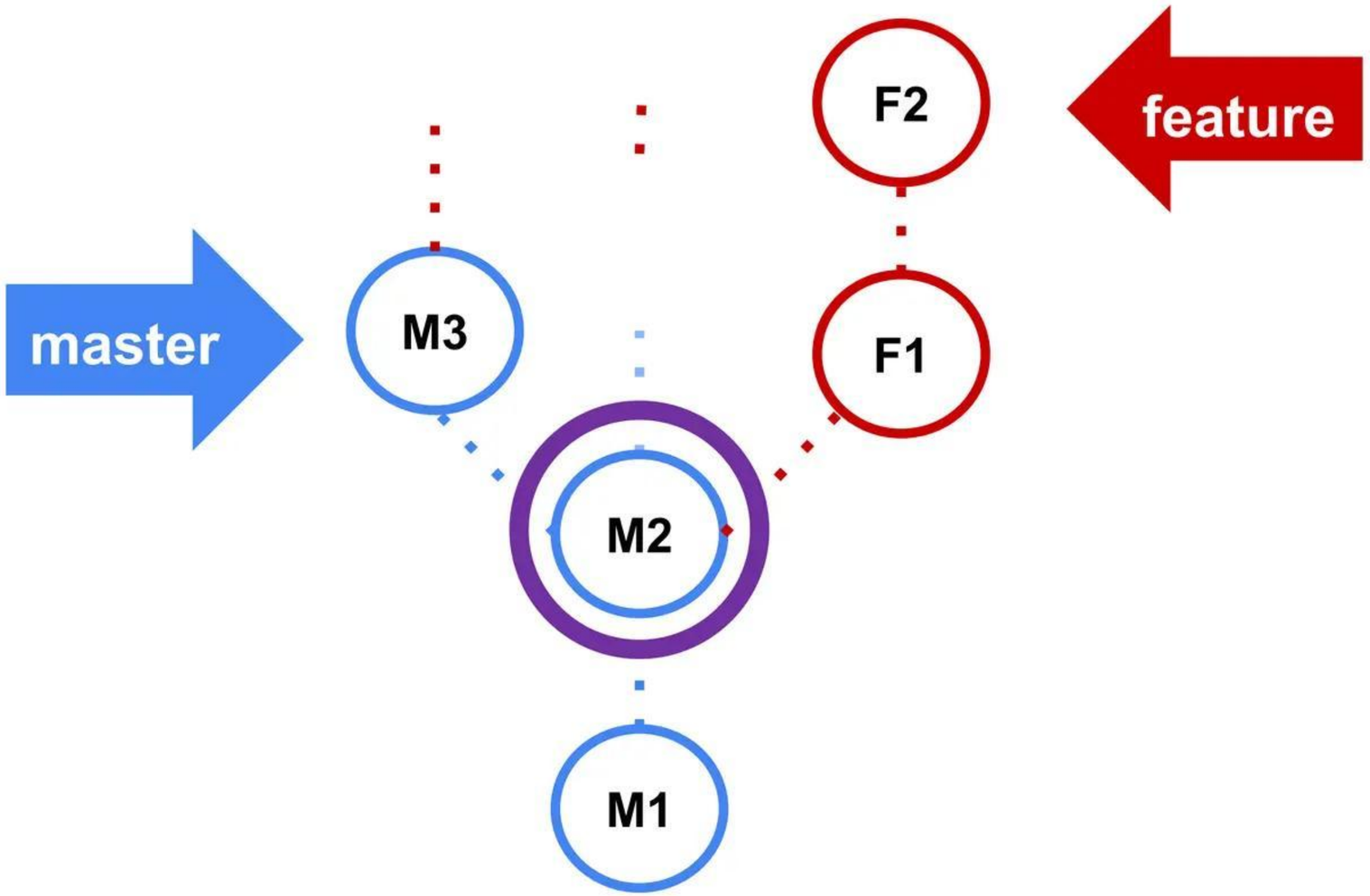
# Rebase



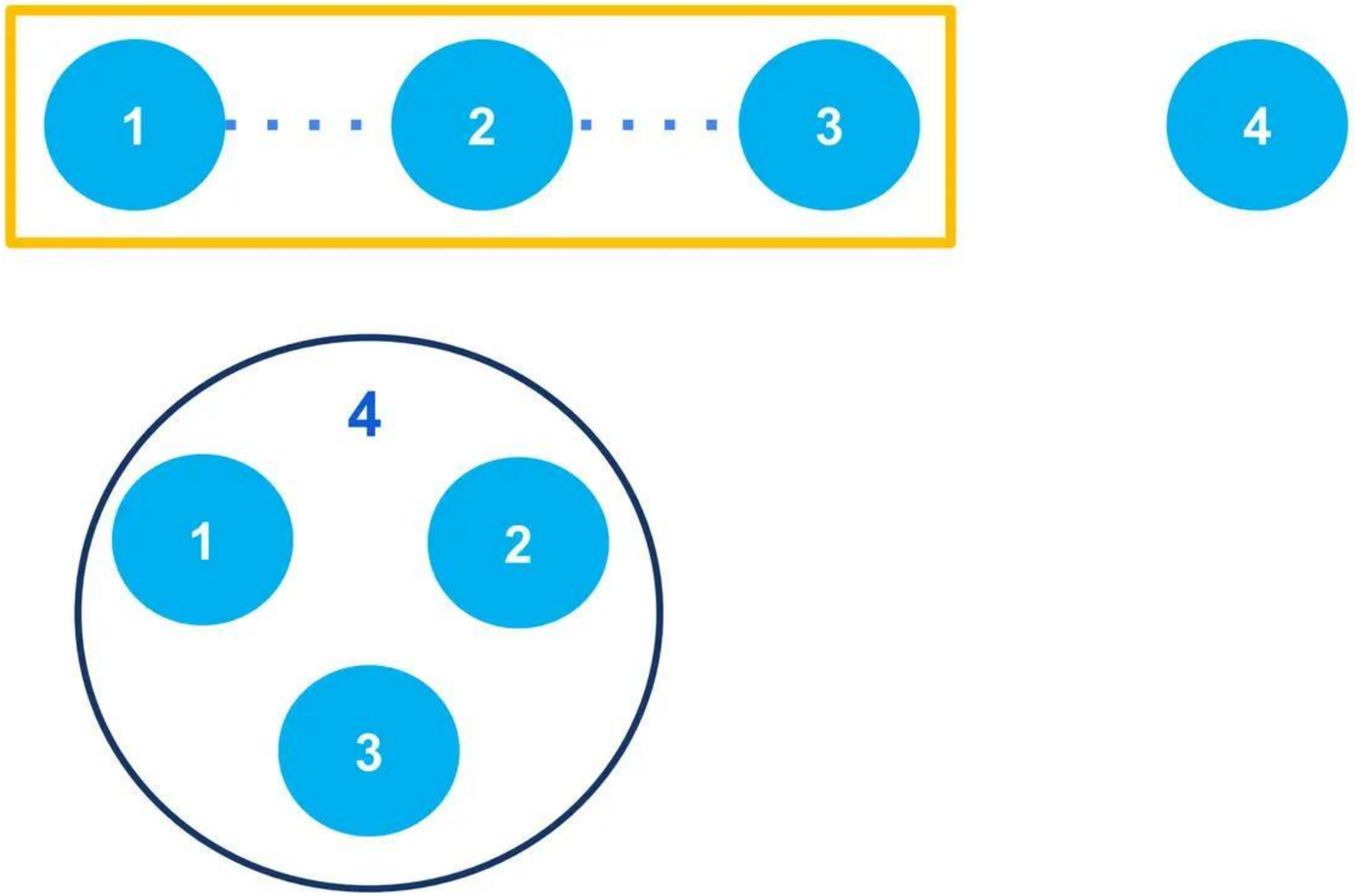
# Rebase



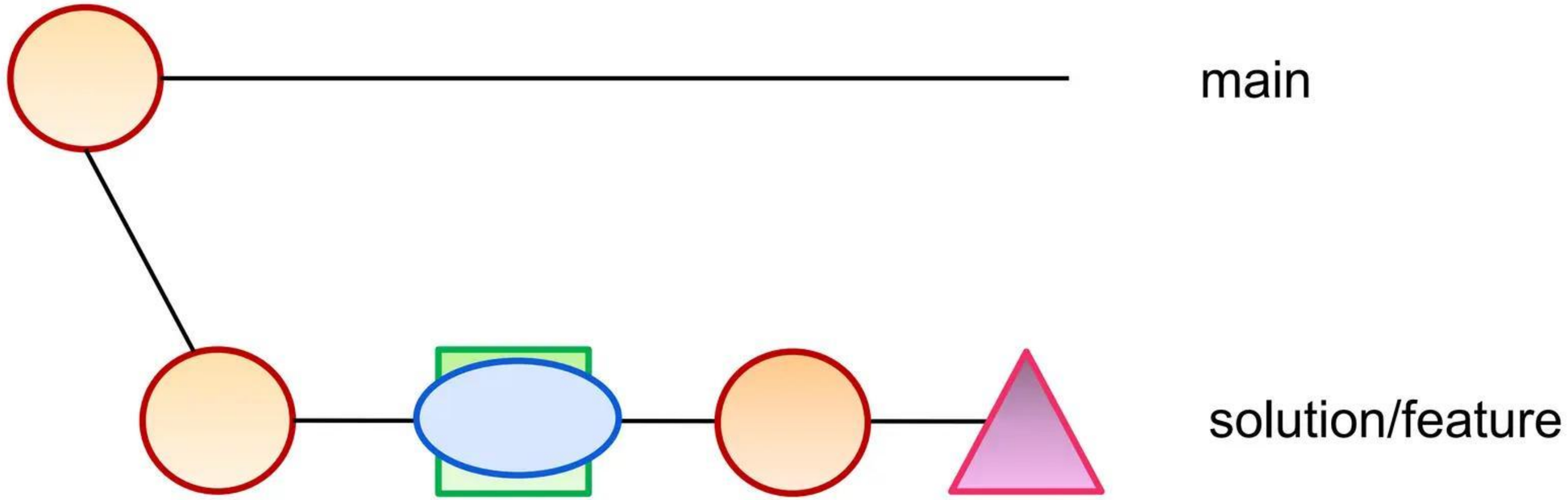
# Rebase



# Squash Commits



# Squash Commits



# Merge

**Combine changes from one branch to another**

- Takes commits from **source** branch
- Integrates into the **target** branch

**Combine history of one branch into another**

**How features and bug fixes are added**

# Supported Merge Strategies

Fast forward Merge

3-Way Merge
or
Standard Merge
or
True Merge

# Fast forward Merge

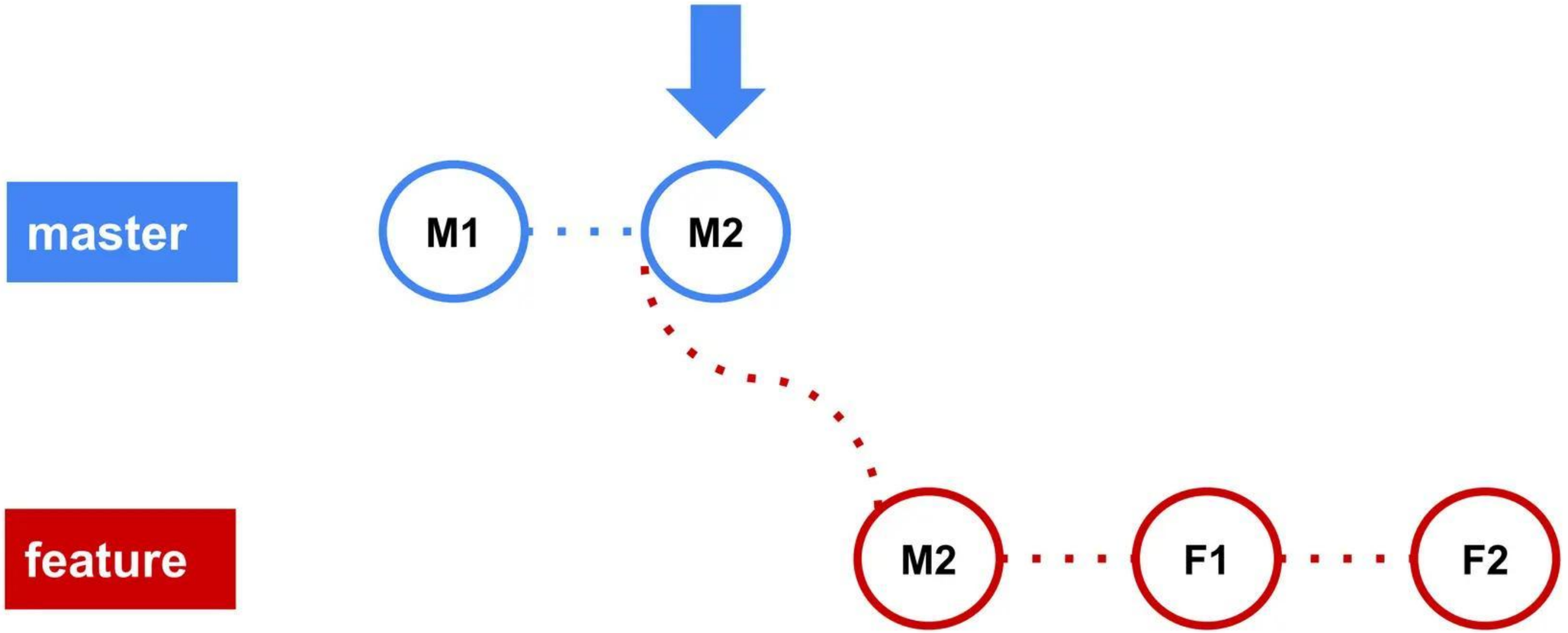
**Special type of merge**

- Branch being merged has no additional commits since the branching point

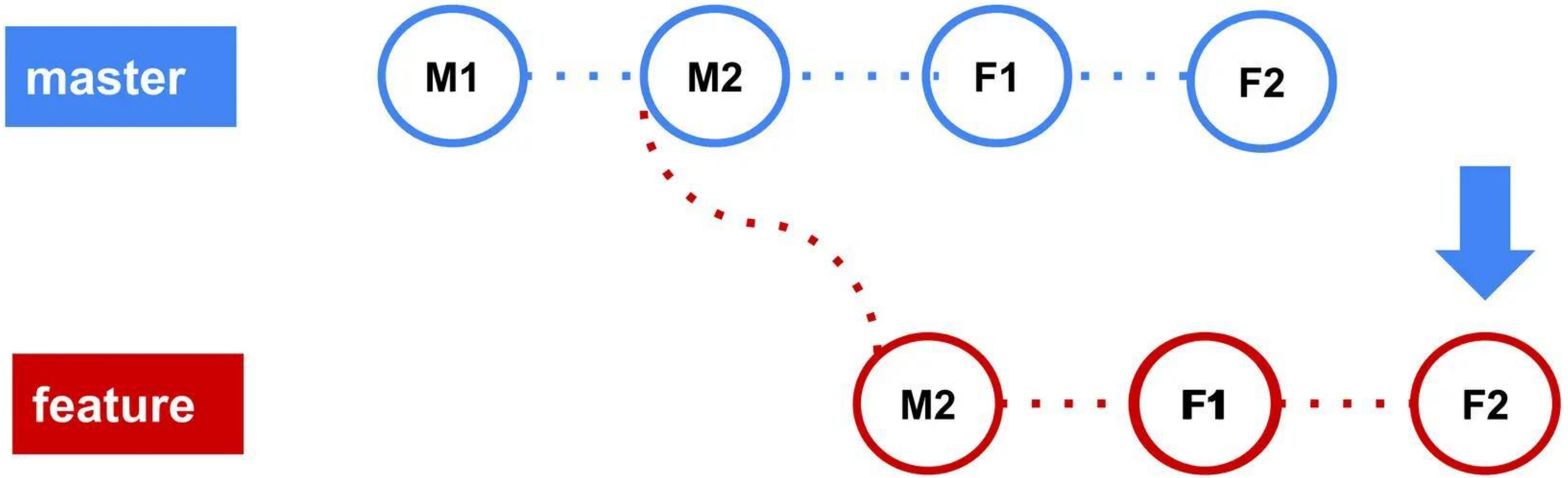
**The head/branch pointer of the target branch is moved to the latest commit of the source branch**

**Causes the target branch to “fast forward”**

# Fast forward Merge



# Fast forward Merge



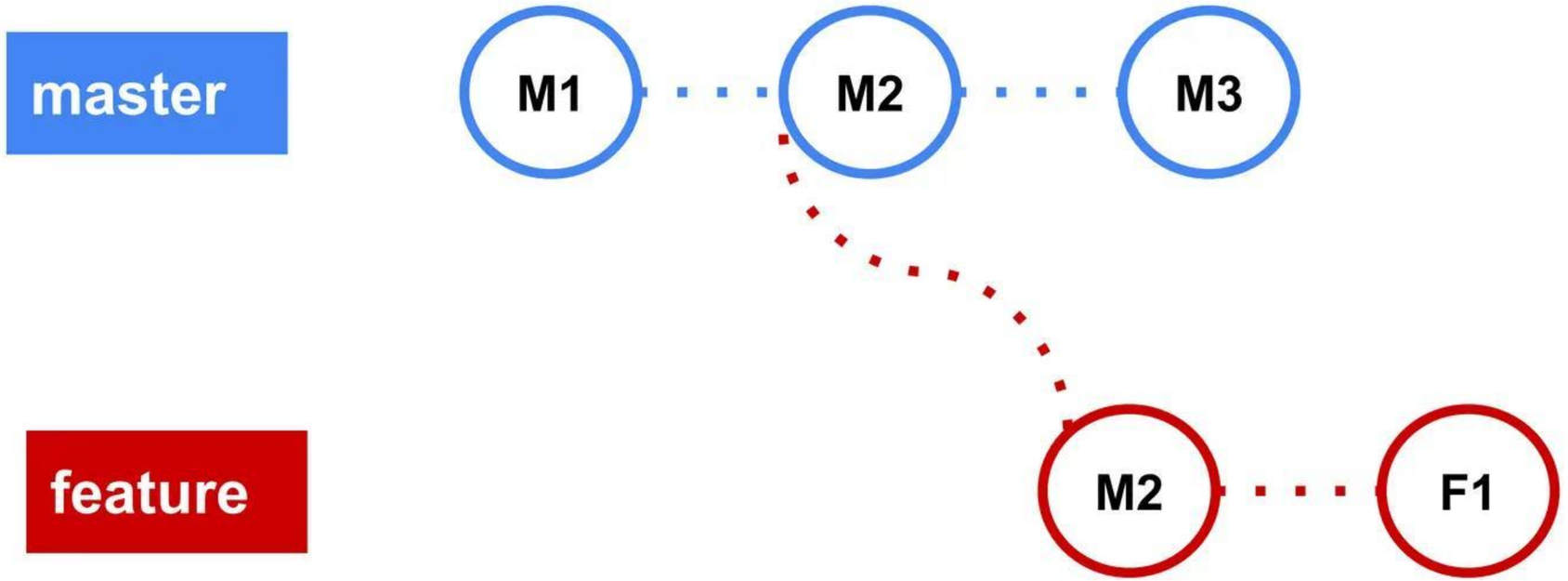
# 3-Way Merge

Both of the branches have **diverged**. Meaning both having additional commits since the branching point.

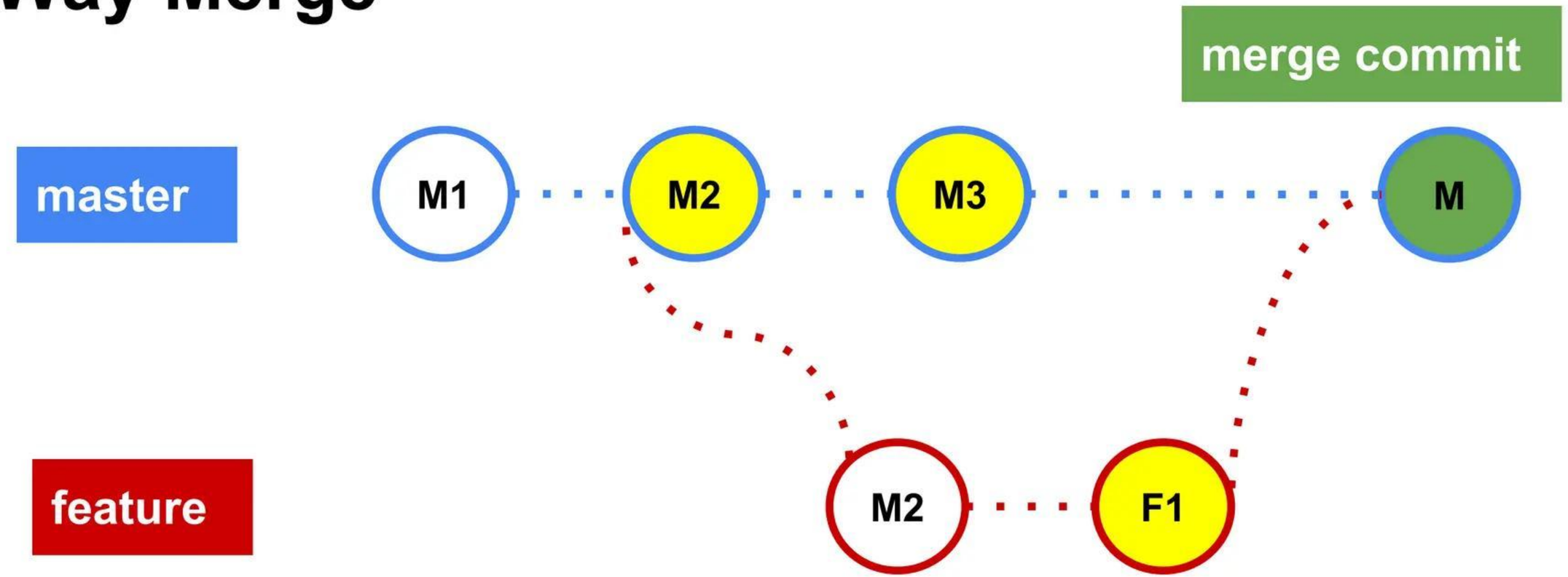
3-Way Merge use a dedicated commit (i.e. **merge commit** ) to tie together the two commit histories.

Git uses **three commits** to generate the merge commit: the **two branch tips** and **their common ancestor**.

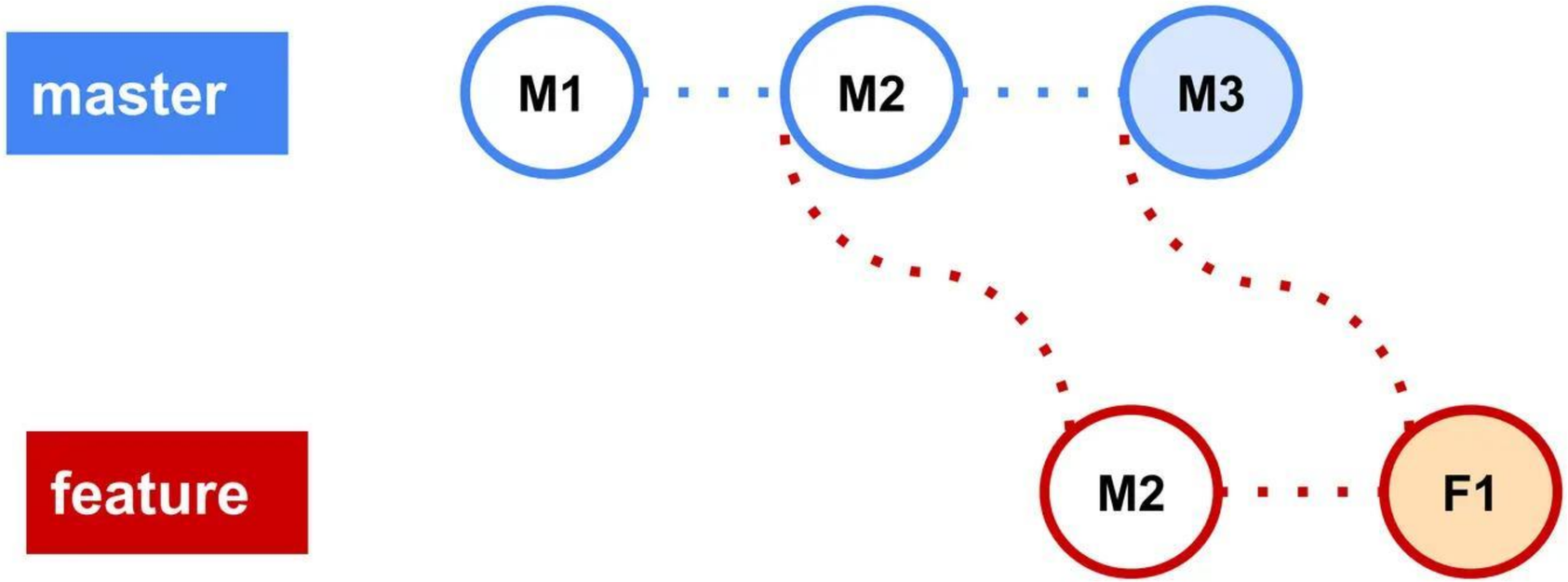
# 3-Way Merge



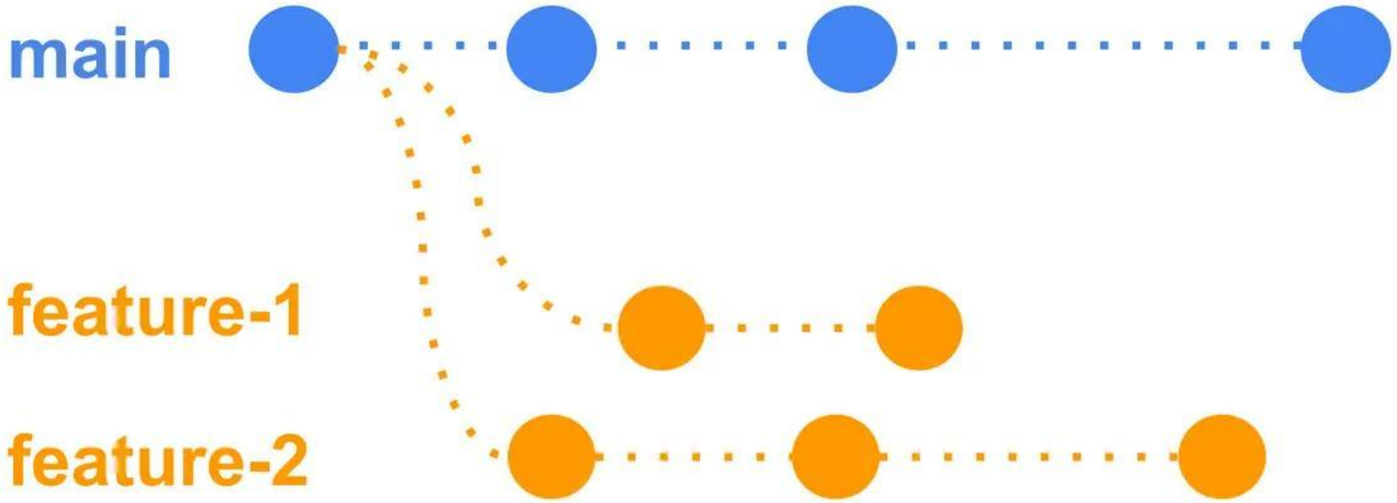
# 3-Way Merge



# Rebase



# Branch




**Independent development line of work**

**Default branch is main or master**

**Each branch has its own commit history**

# Branch

**Branches isolate your development work from other branches in the repository**

- Allows working on different features or bugs without affecting the main codebase

**Developers can create new branches**

- Part of their daily workflow

**Changes made on one branch**

- Do not affect other branches
- Until they are **merged**

# What We Would Do with Branches ?

- ❑ Create a new branch
- ❑ Create a branch from a previous commit
- ❑ Choose a branch and make few changes
- ❑ Compare branches
- ❑ Publish a branch to remote
- ❑ Rename a branch
- ❑ Delete a branch

# Type of Branches

- ❑ Local Branch
- ❑ Remote (tracking) Branch

# Local Branches

- A local branch exists only on your local machine.

- When you create a new branch in GitHub Desktop, you're creating a local branch. This branch is a copy of your code that you can work on independently.

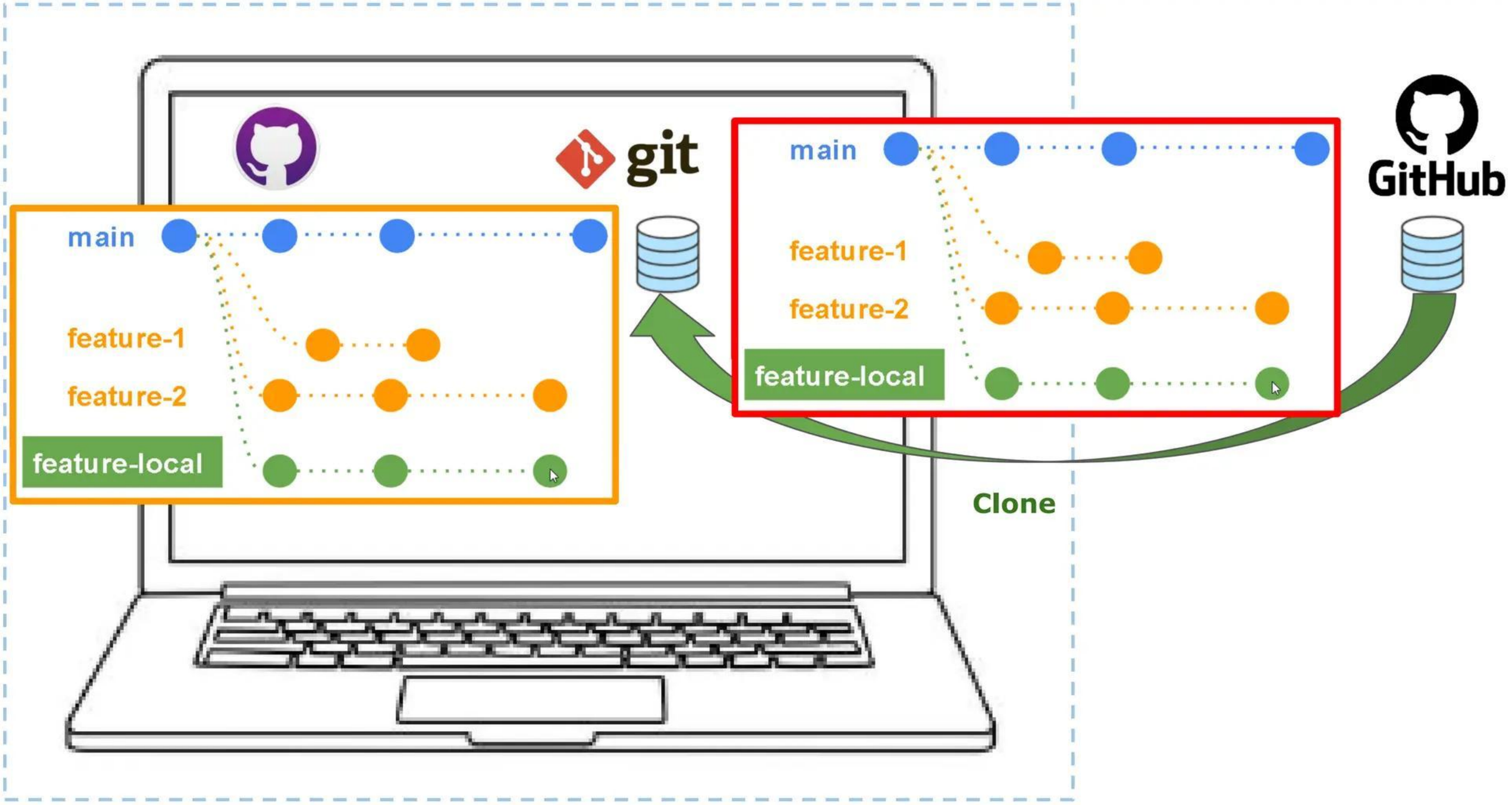
- Any changes you make to local branch are saved **locally**, meaning they won't be visible to others until you push the branch to a remote repository.

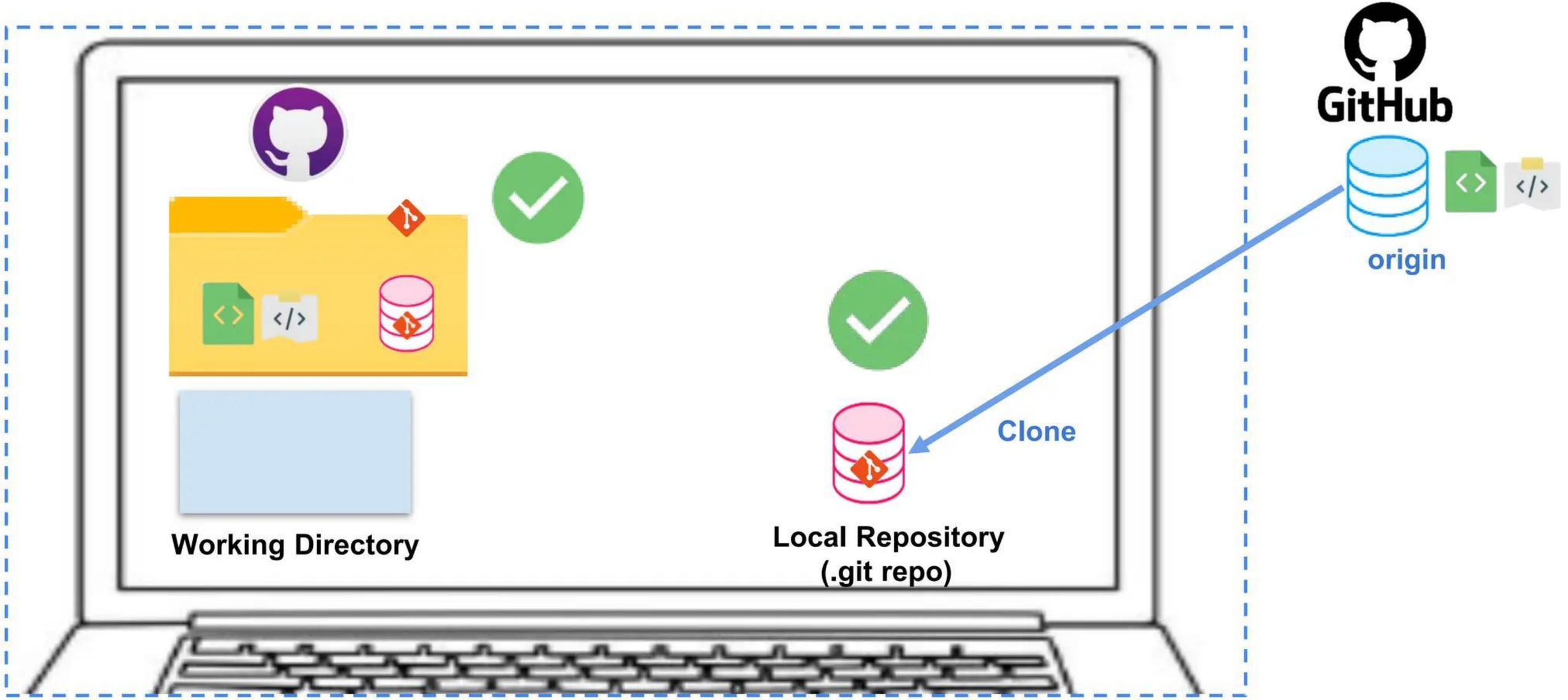
# Remote (tracking) Branches

- Remote tracking branches are basically local branches that have a special connection to a branch on a remote repository ,such as GitHub.

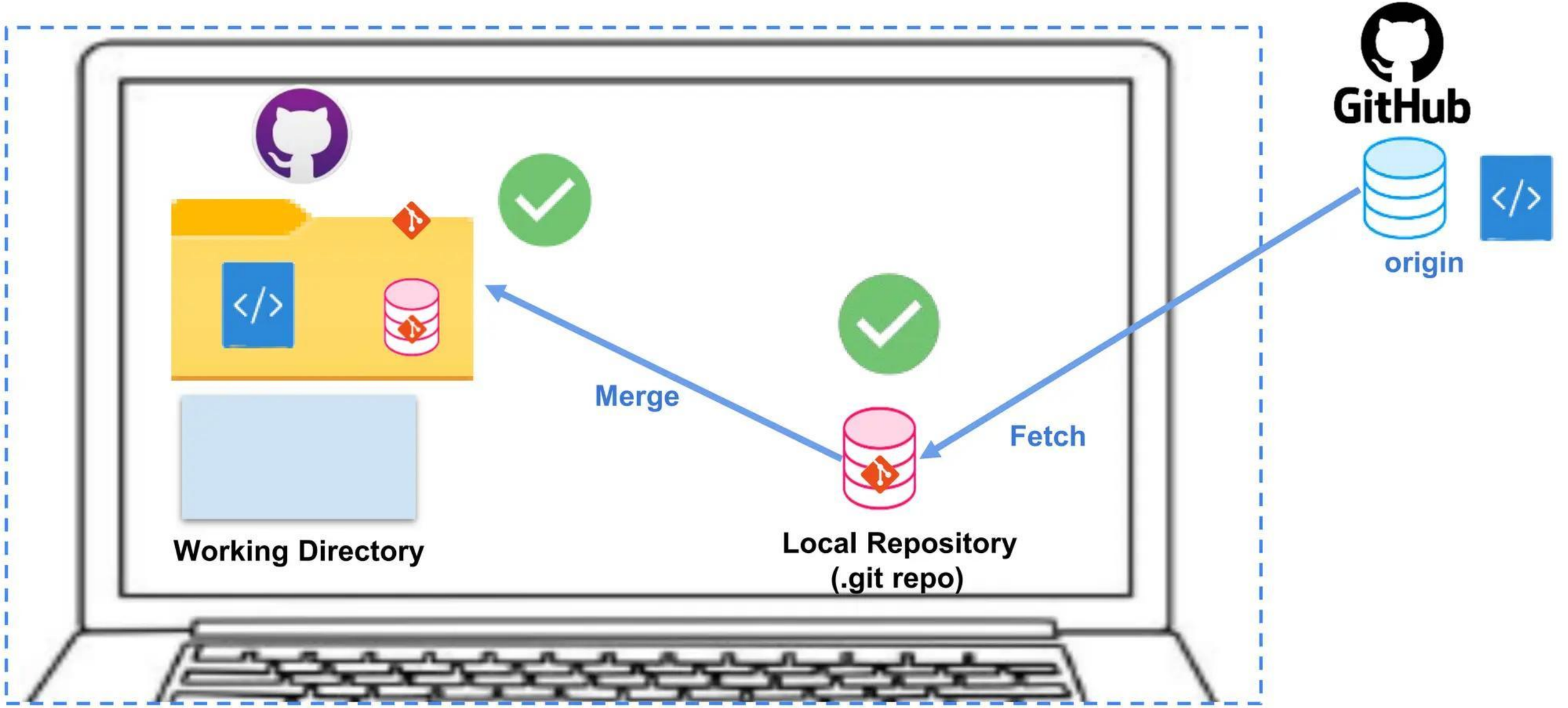
- When you clone a repository, GitHub Desktop automatically creates remote tracking branches that mirror the branches on the remote.

- Remote tracking branches are used for collaboration and staying synchronized


git
main
feature-1
feature-2
feature-local
main
feature-1
feature-2
feature-local
GitHub
Clone


GitHub
origin
Clone
Working Directory
Local Repository
(.git repo)

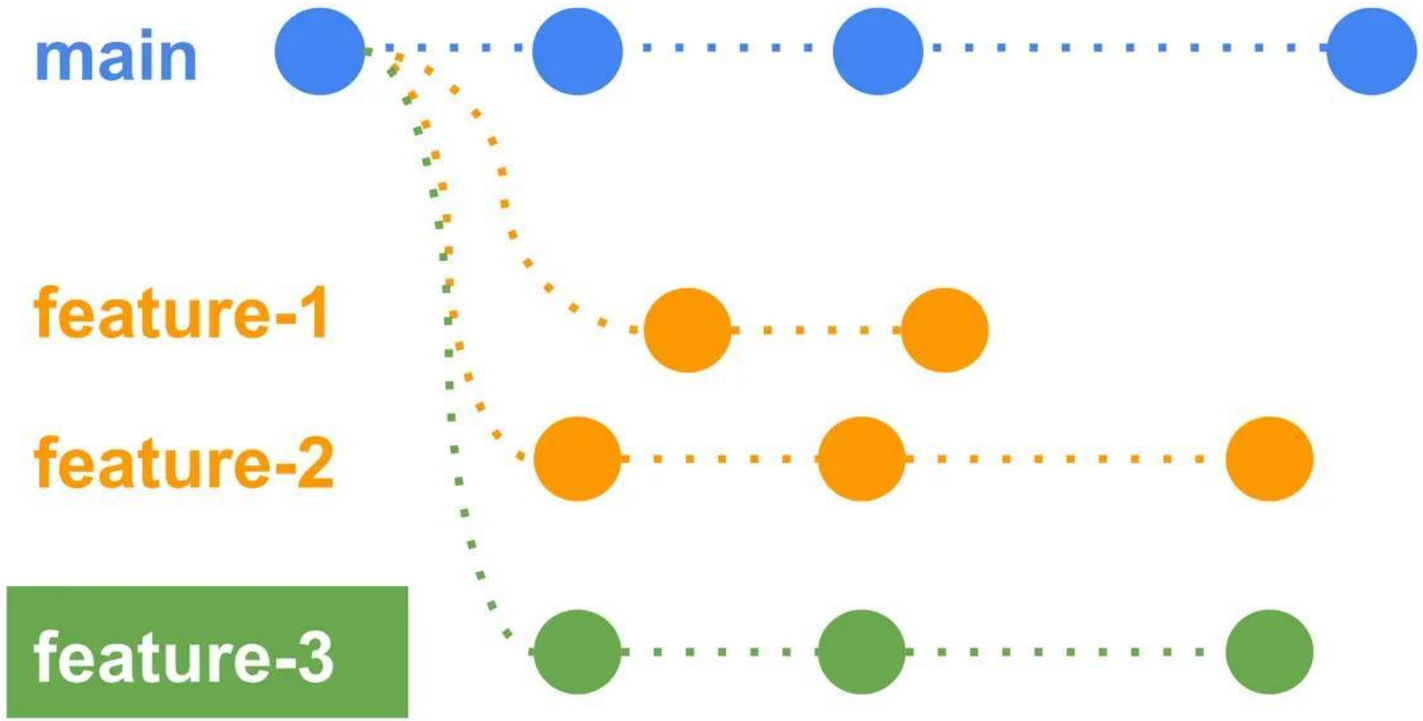
# Branch and Merge


GitHub
origin
Merge
Fetch
Working Directory
Local Repository
(.git repo)

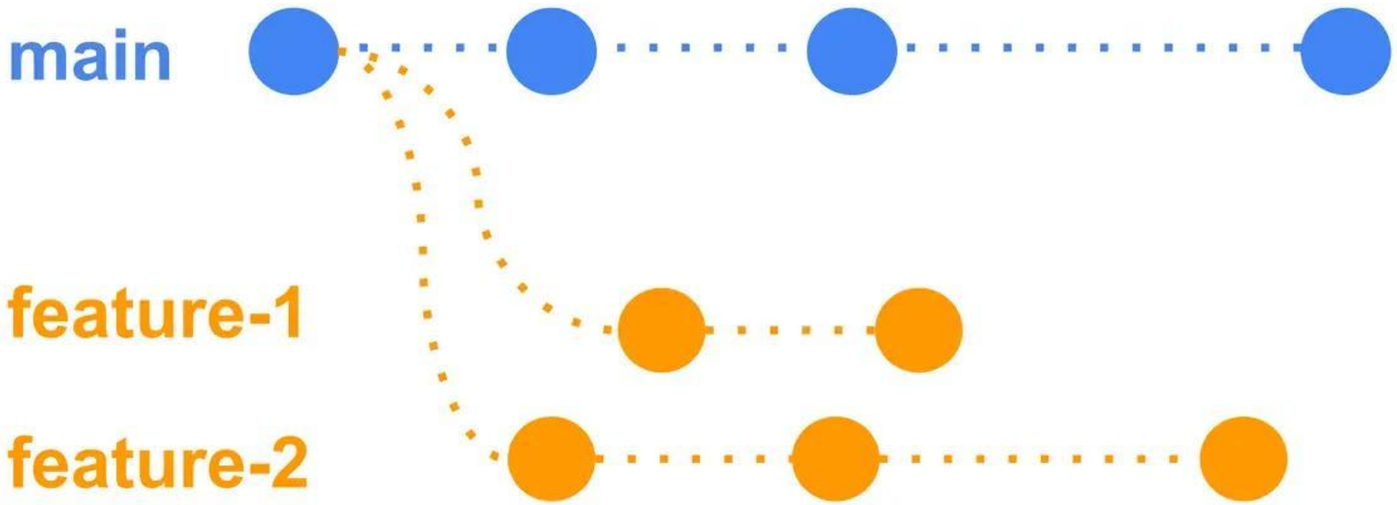
# What Can We Do with Branches ?

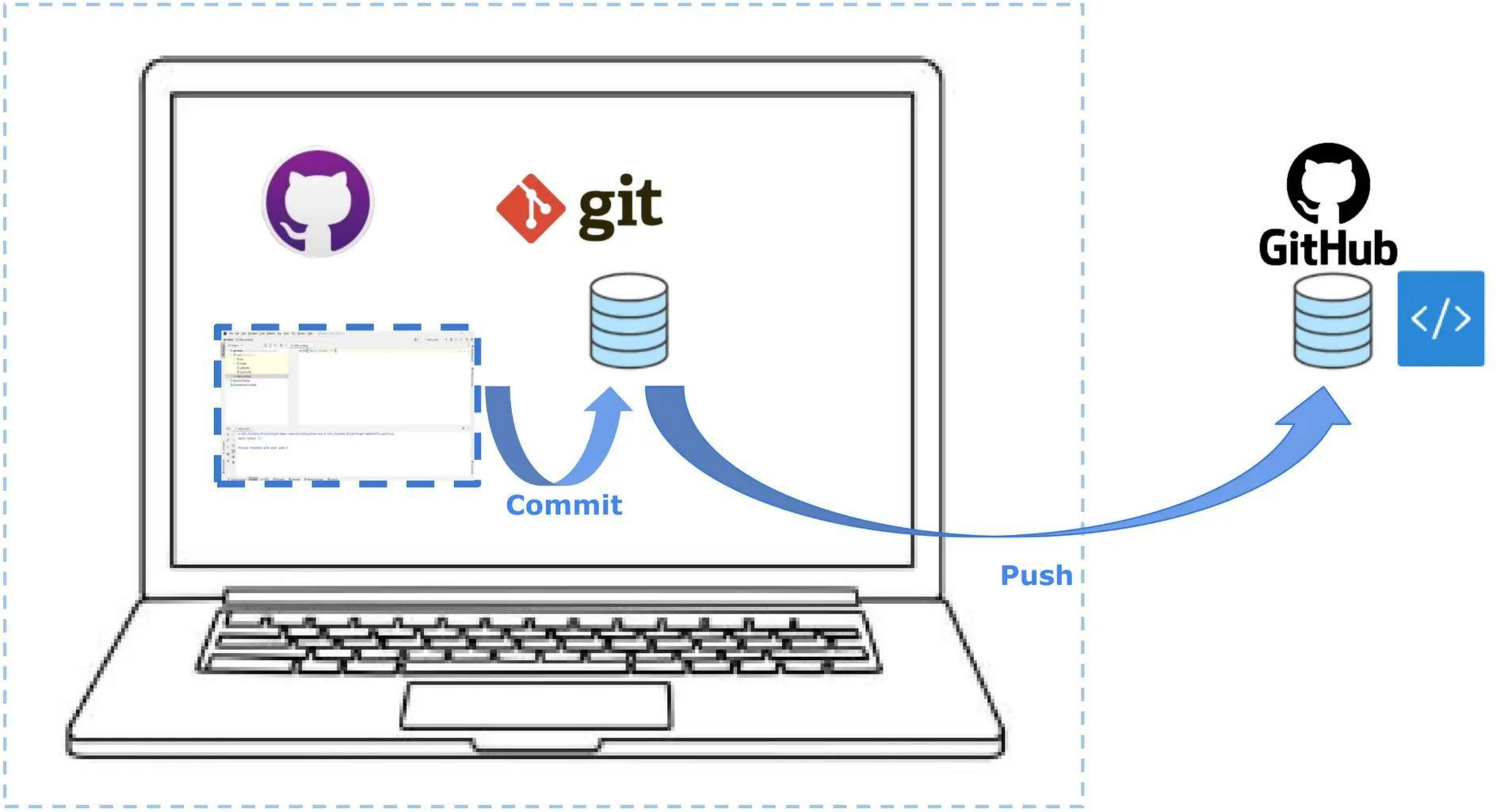
- ❑ Publish branches
- ❑ Commit into branches
- ❑ Rename or delete branches
- ❑ Compare branches
- ❑ Merge branches
- ❑ Rebase branches

# Branch

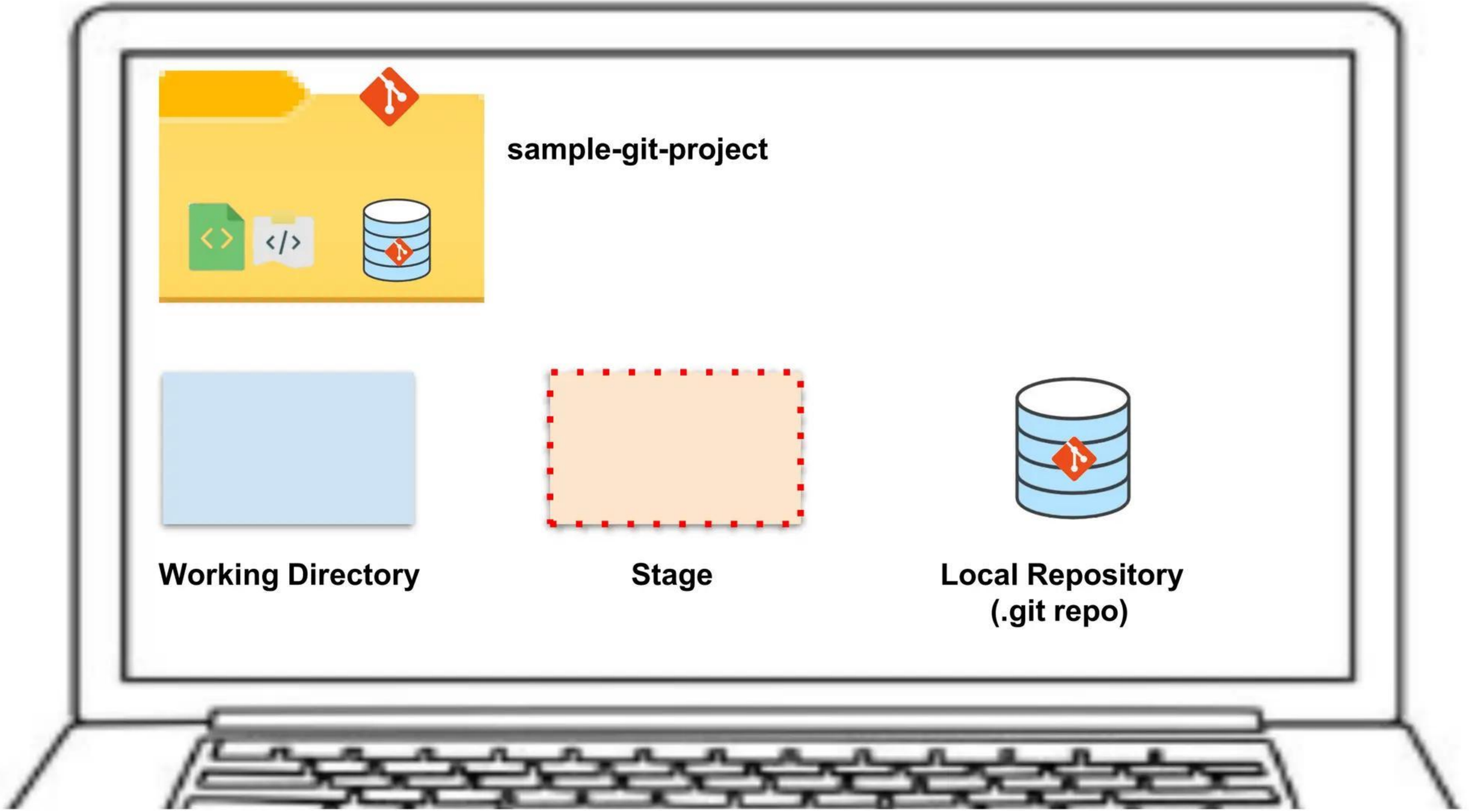


# Branch




git
GitHub
Commit
Push

- Switch to Repo
- Initial Commit bottom left


sample-git-project
Working Directory
Stage
Local Repository
(.git repo)

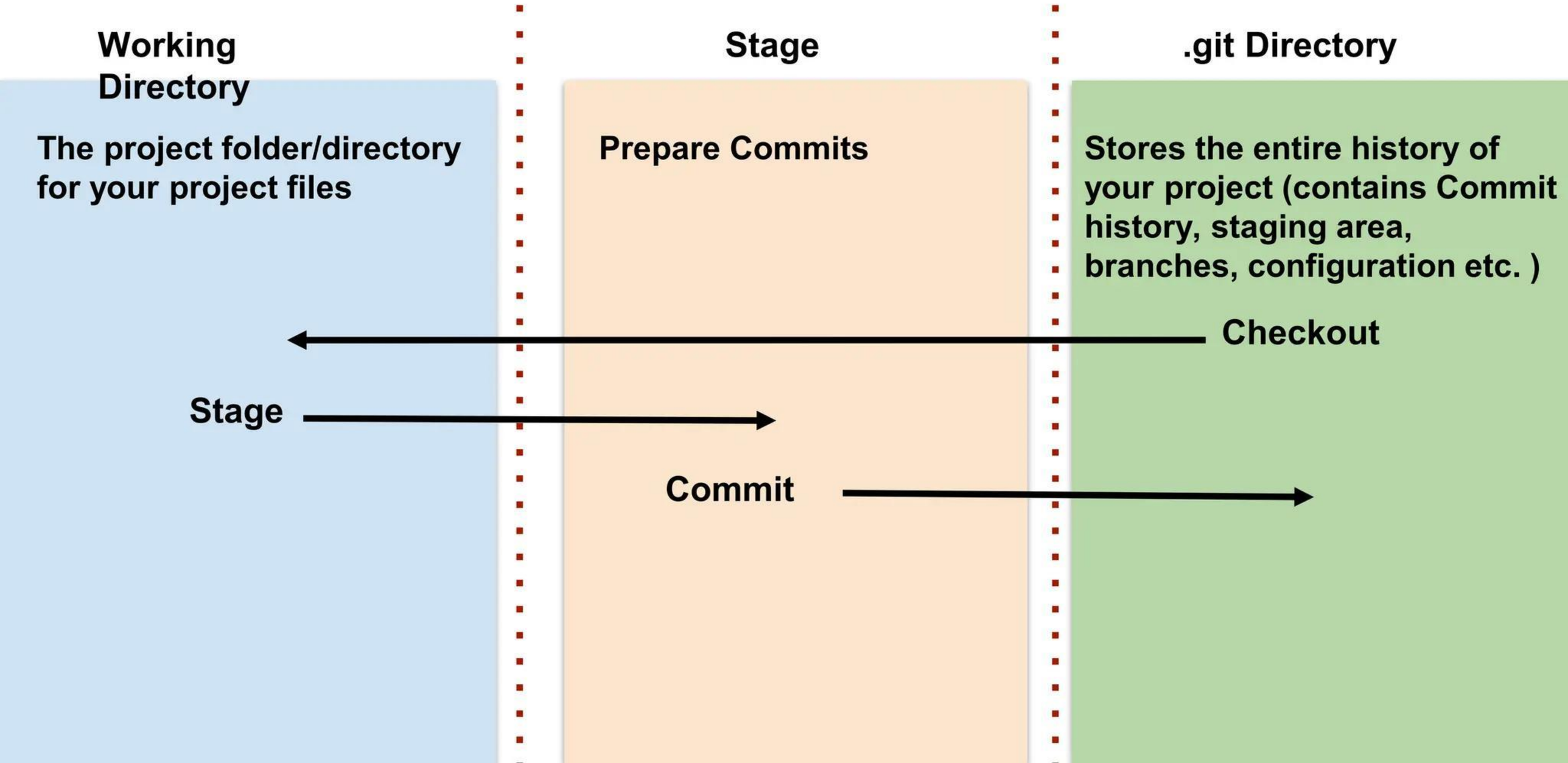
# Commit

Snapshot of a repository at one point in time

Each commit contains

- Snapshot of files
- Commit message
- Author and timestamp
- Unique Identifier

# The Three Sections of a Git Project



# The Three States of Files

# Modified or Untracked

Modified
- Changed since last commit

Untracked
- Not yet committed

Git detects changes by comparing current state of the file with version in repo

# Staged

Process of marking files to be included in the Commit

These files are added to Index

- Also known as Staging Area

An entire file can be staged

- Or only a specific portion

# Committed

Represents files that have been added permanently to the repository

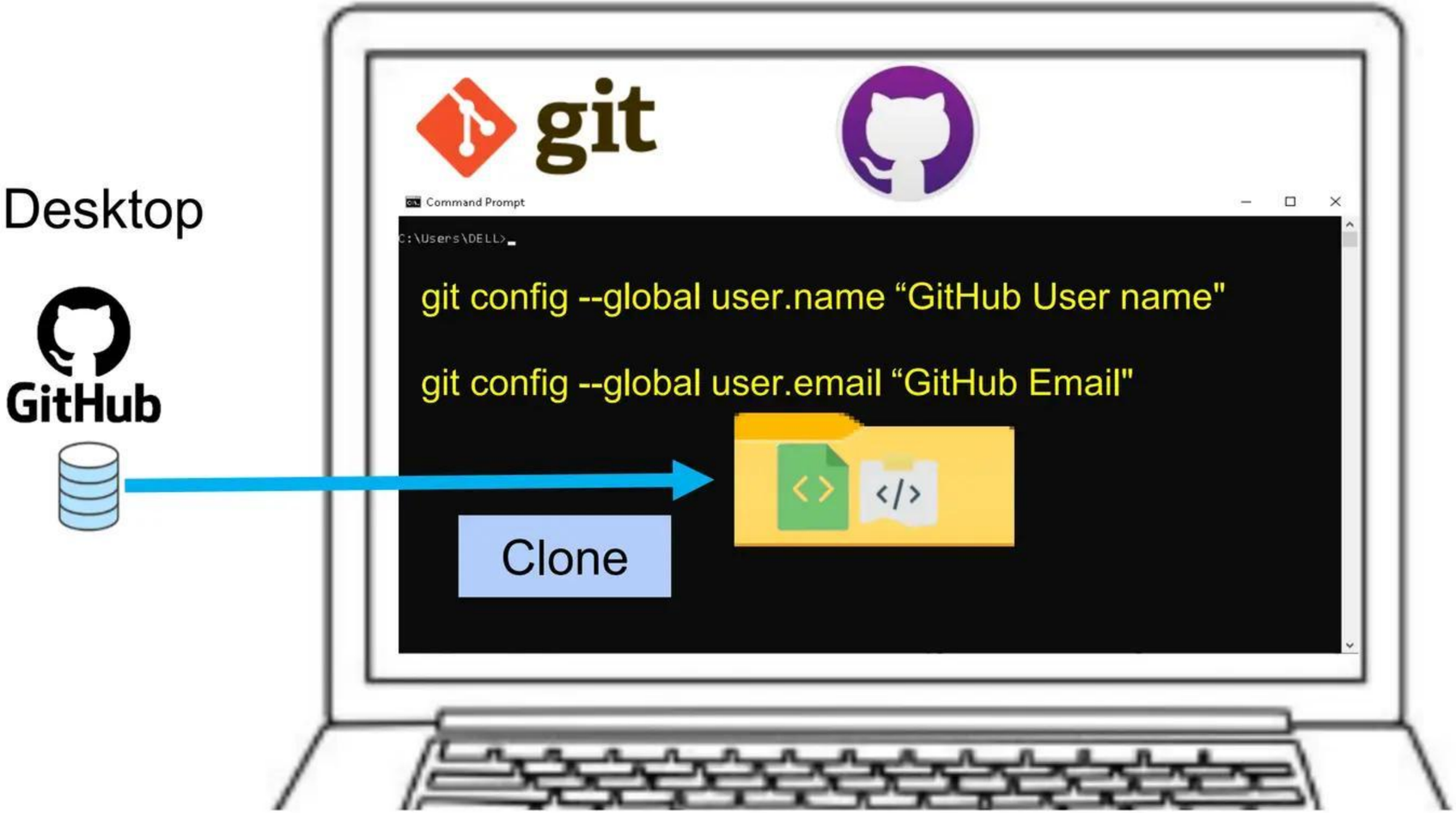
- Commit becomes part of repo’s History

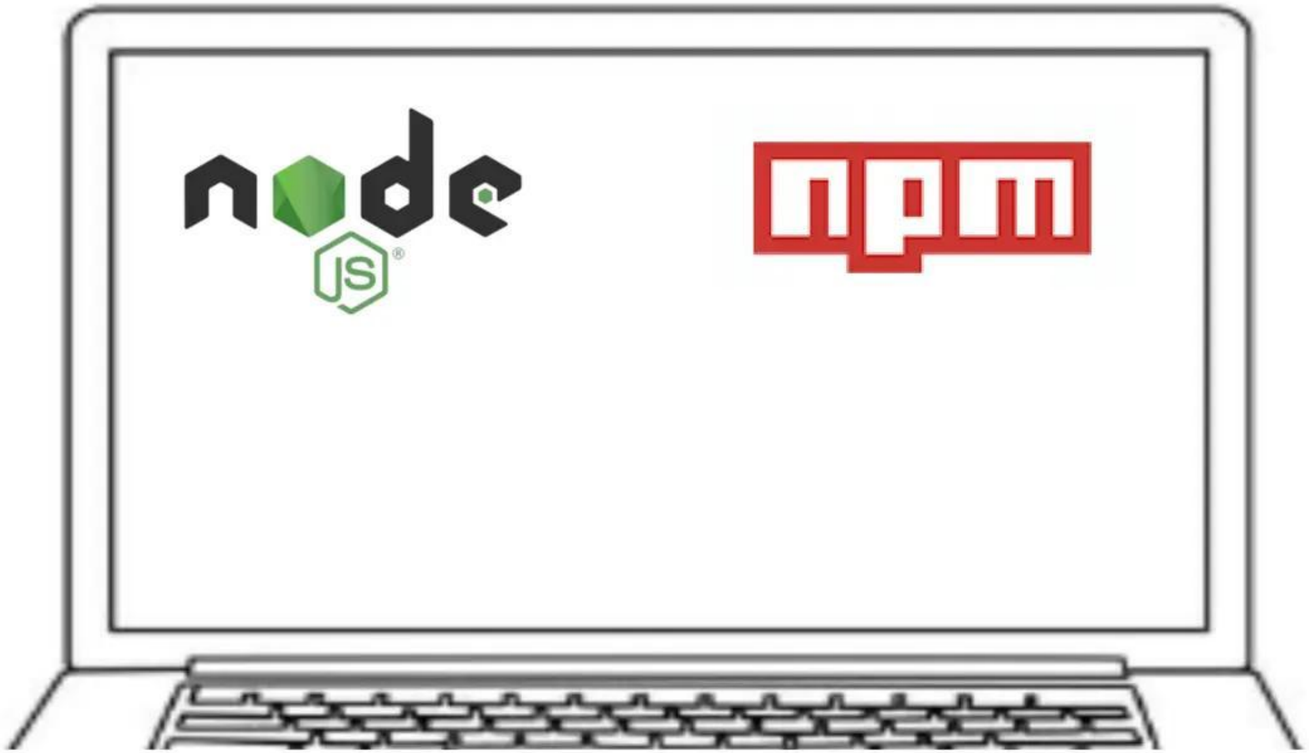
# Setting up GitHub Desktop

Install Git

Configure Git for GitHub Desktop

Install GitHub Desktop






**Node.js** is a open source and cross-platform runtime environment for JavaScript . It allows JavaScript to be run outside of a browser, enabling server-side scripting.

**npm** (Node Package Manager) is bundled with Node.js and is the package manager for Node.js packages

Install Node.js

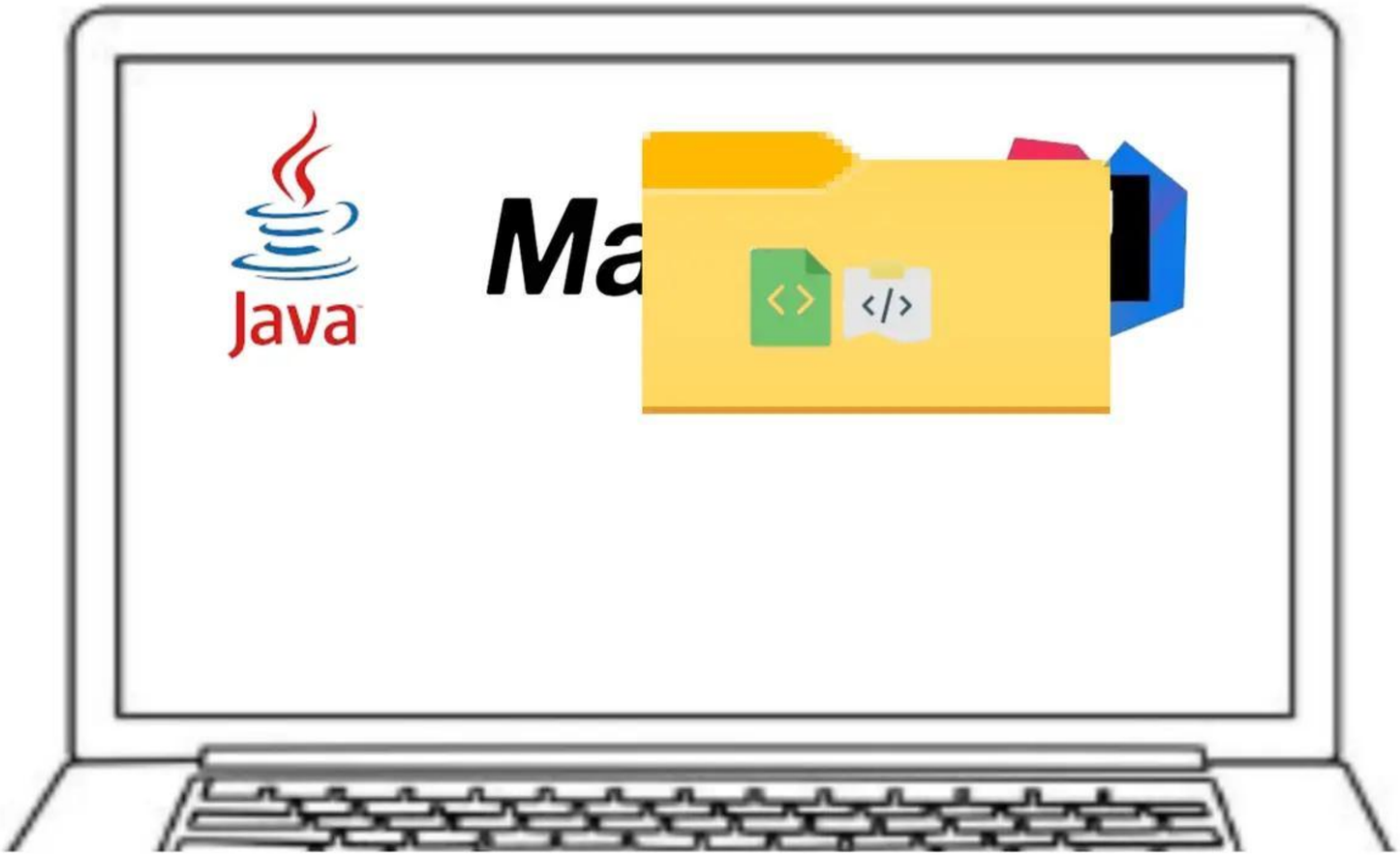
Install VS Code

Install Java 8

Install Maven

Install Intellij IDEA CE

Create and Run Sample Maven based project in Intellij

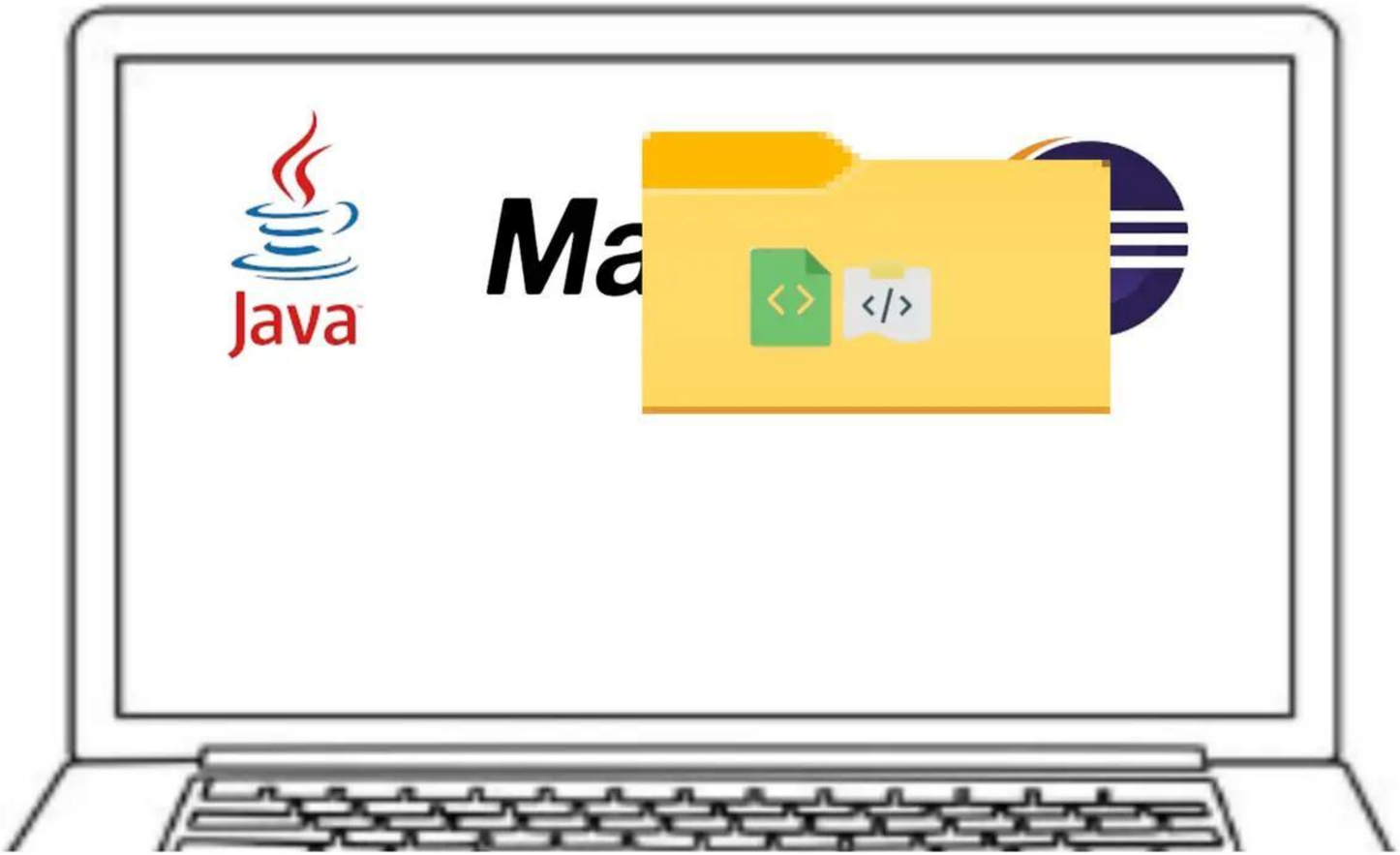


Install Java 8

Install Maven

Install Eclipse IDE

Create and Run Sample Maven based project in Eclipse



**Node.js** is a open source and cross-platform runtime environment for JavaScript . It allows JavaScript to be run outside of a browser, enabling server-side scripting.

**npm** (Node Package Manager) is bundled with Node.js and is the package manager for Node.js packages

- Used to install ,manage and share JavaScript packages
- Handles dependency management

Why **"npx create-react-app <app_name> "** command looks for the location **"C:\Users\<username>\AppData\Roaming\npm "** ?

**create-react-app** is an npm package

The location "C:\Users\<username>\AppData\Roaming\npm" is the directory for **globally installed npm packages**

If create-react-app is already globally installed, npx will use the global installation

If create-react-app is not globally installed, npx will temporarily install and run it

Install Node.js

Install VS Code

Create a React App

Architect

GitHub

main

development

feature/logging

feature/validation

Developer1

Developer2

Architect

main

development

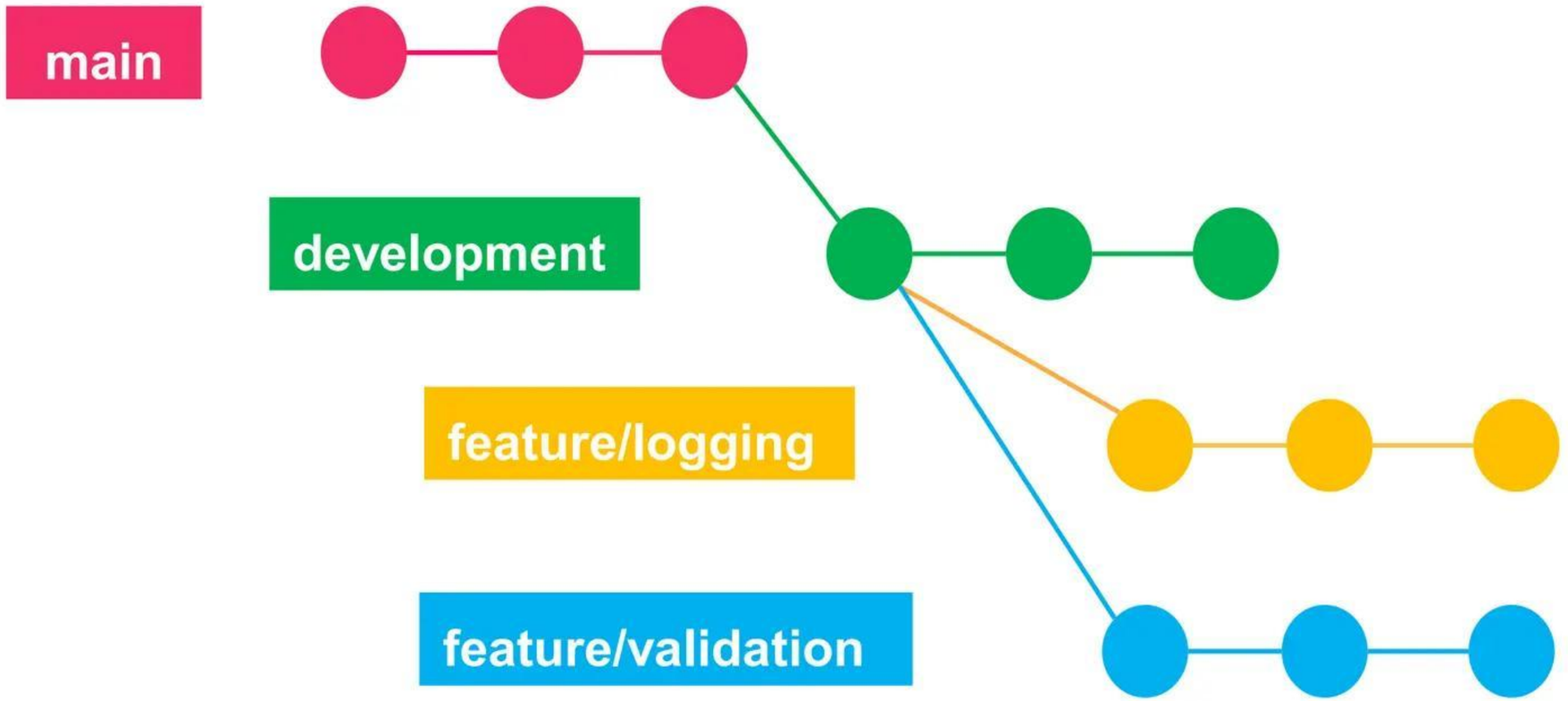
feature/logging

feature/validation

GitHub

Developer1

Developer2


main
development
feature/logging
feature/validation

# Stash

If you have saved changes that you are not ready to commit yet, you can **stash** the changes for later.

After stashing changes on a branch, you can safely change branches or make other changes to your current branch

# Stash

When you stash changes

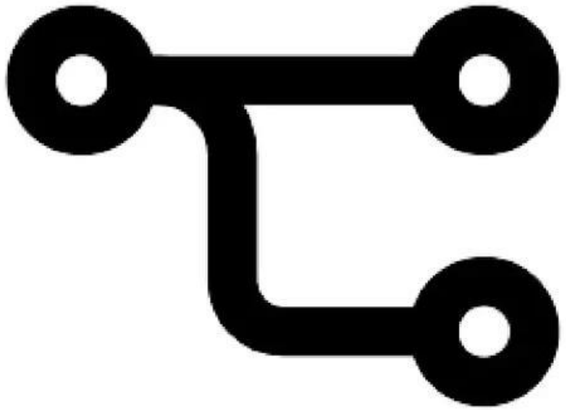
- The changes are temporarily removed from the files

- You can choose to **restore** or **discard** the changes later.

Stashes are **local operation** , they are not pushed

You can only stash one per branch at a time with GitHub Desktop

# Fork



Fork is used to create a copy of a repository in GitHub

Does not impact the original repository which is commonly referred as **upstream repo**

Used for contributing code to a repository where you are not an authorized contributor

Commonly used in open source projects

Changes can be merged back into original repository using pull requests

# Clone vs Fork

Both are used to create a copy of an existing repository

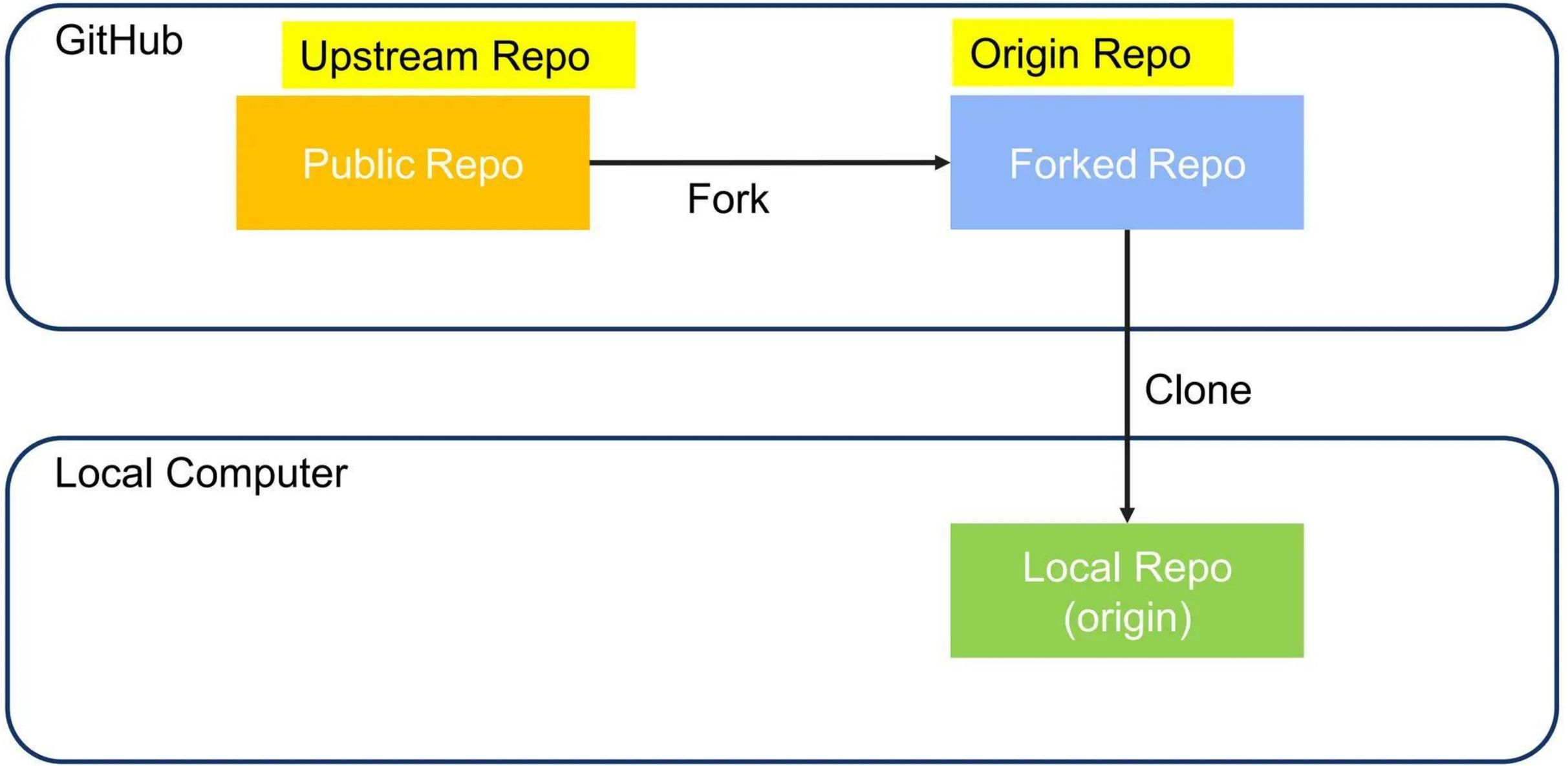
Use **Clone** to make a local copy of an existing repository

Use **Fork** to make a copy of the repository on GitHub

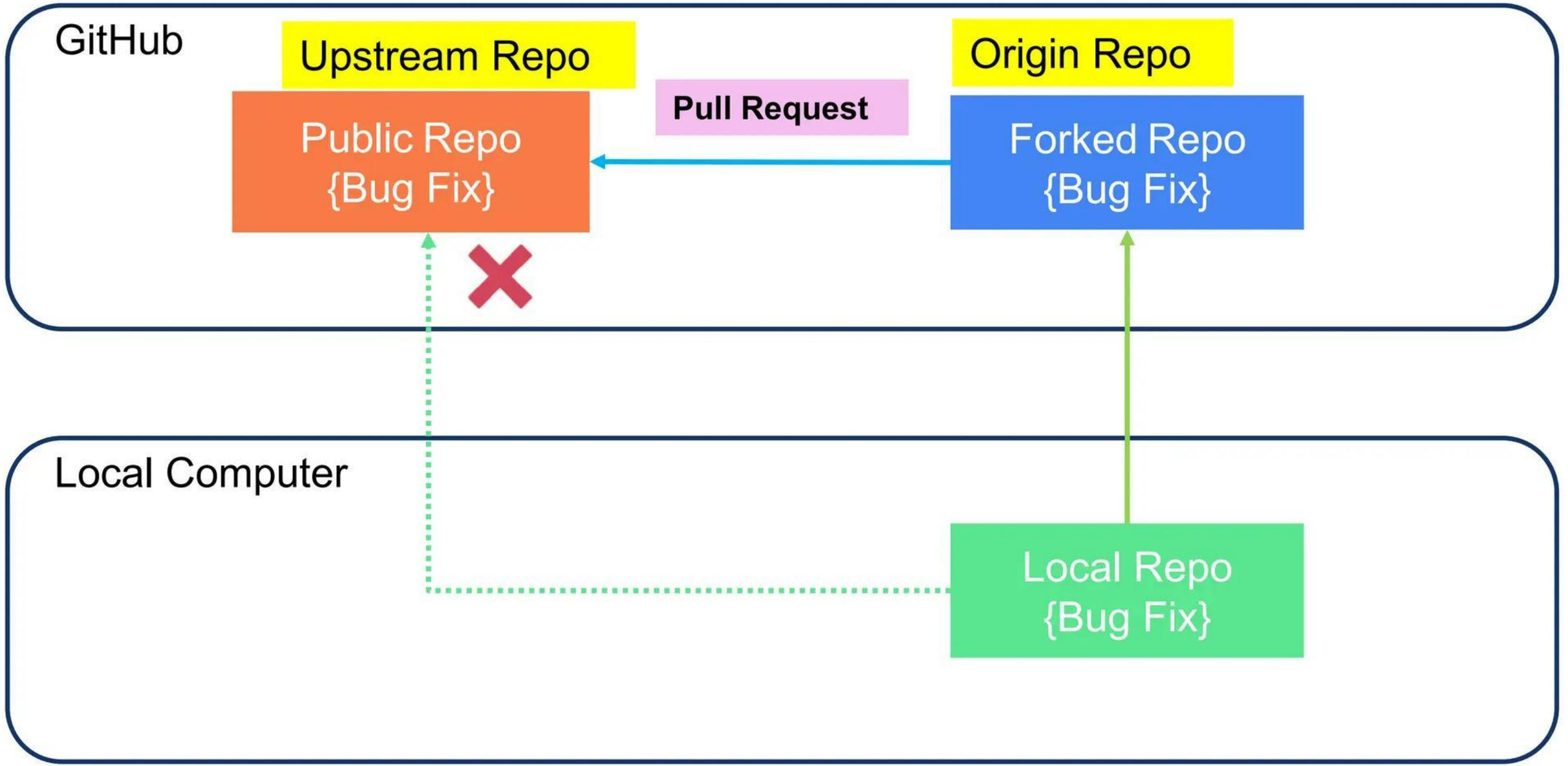
On GitHub, once you create a Fork of a repository ,you have your own **public copy** of the repository in GitHub.

Post Fork creation , to work with the code, **you need a local copy so you have to make Clone of the Fork**. Once you clone, you have a local repo on your computer which is configured with a remote repo named Origin on GitHub.

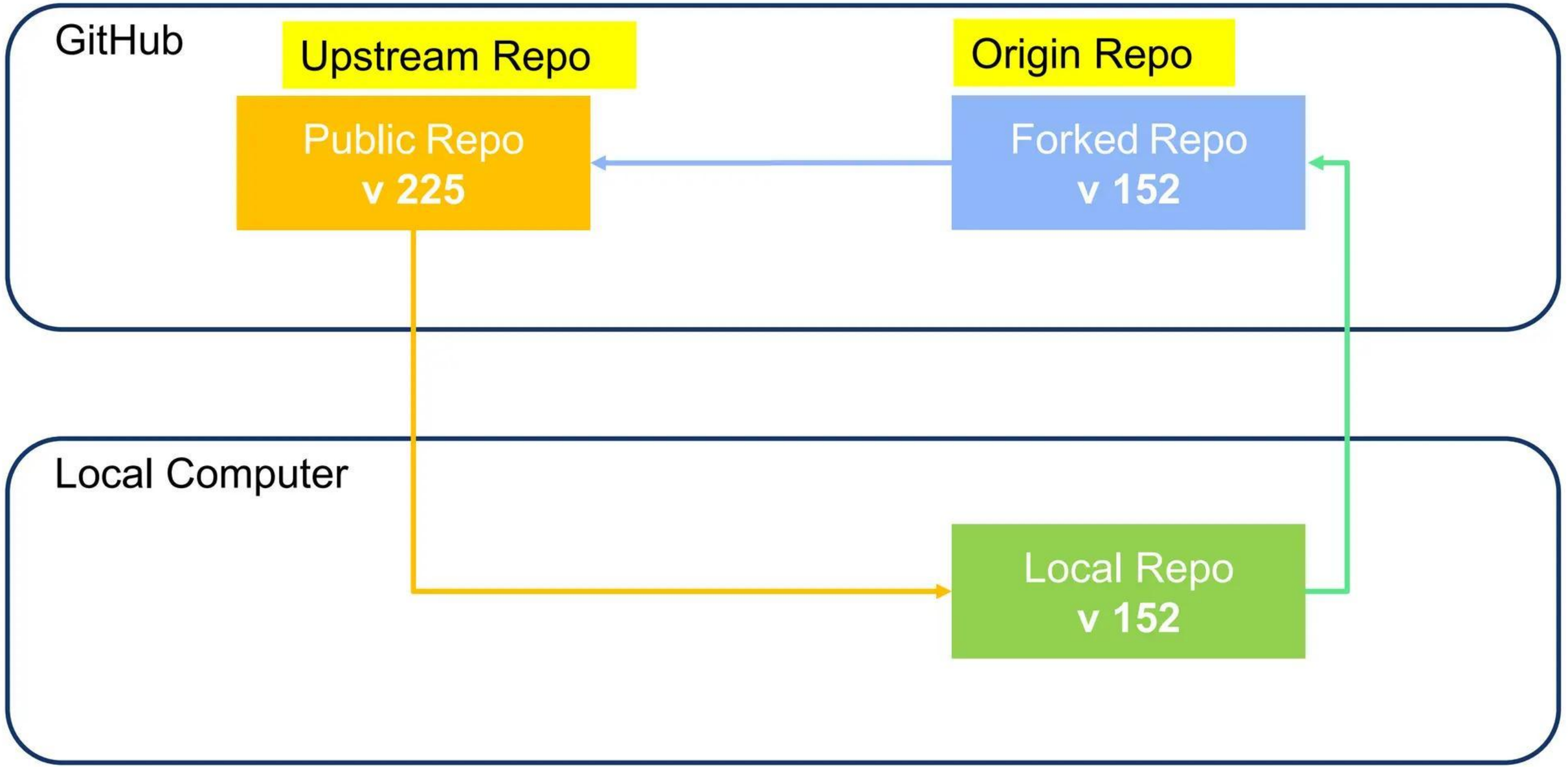
# Fork Overview

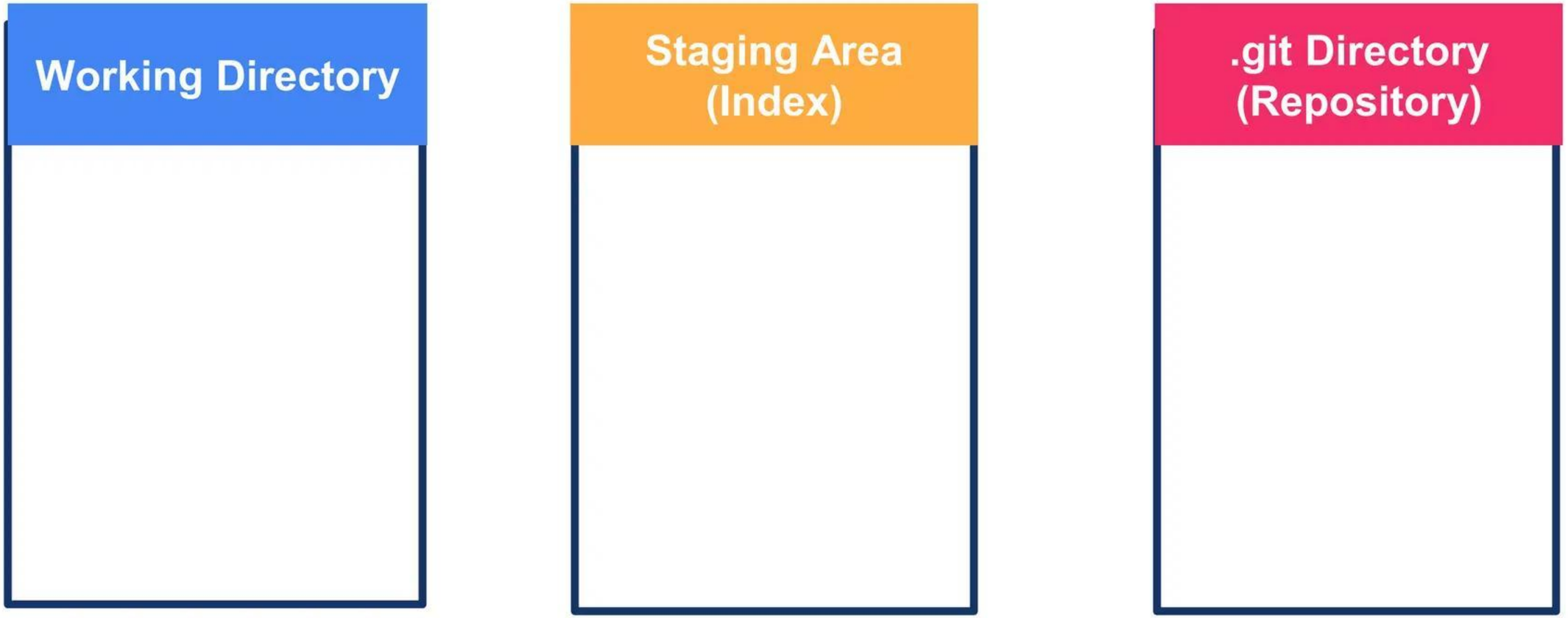


# Fork Overview



# Syncing with parent repository




Working Directory
Staging Area
(Index)
.git Directory
(Repository)

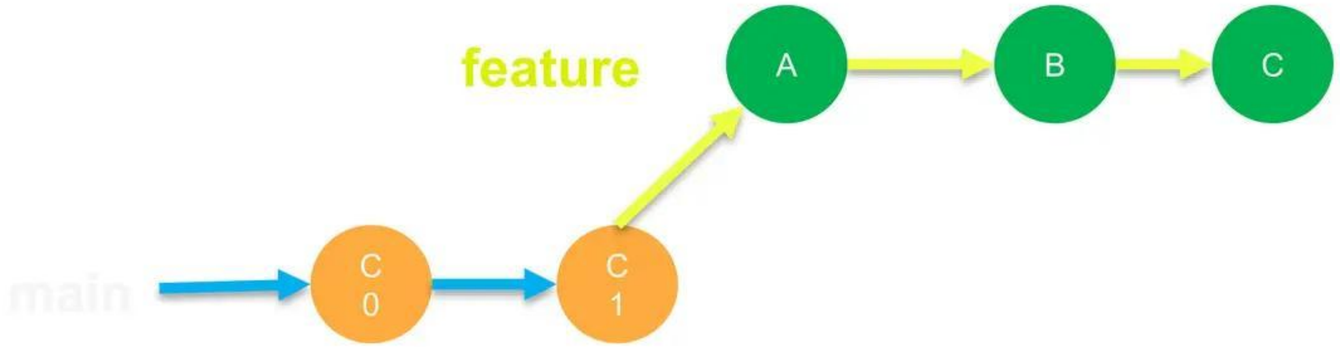
# Merge vs Rebase

Key differences between Merge and Rebase

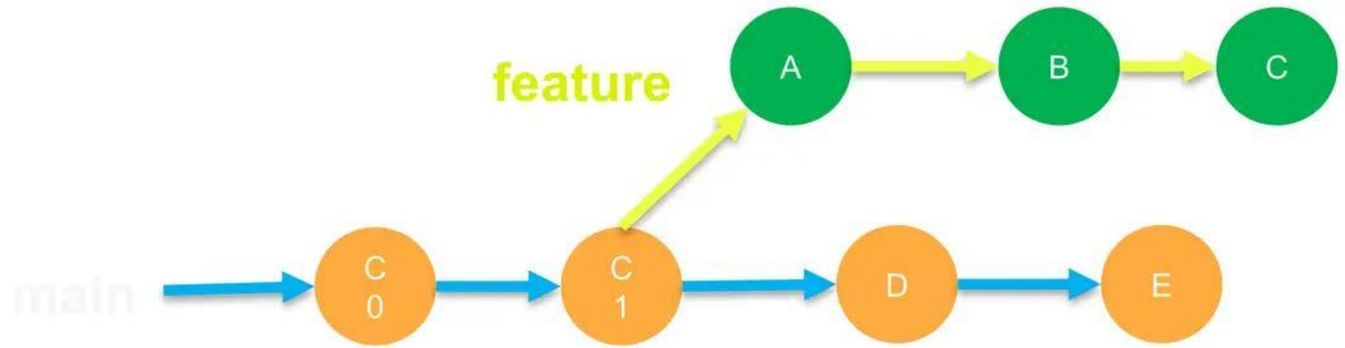
When to use which one for our workflow

- Get changes from a feature branch back into the main branch
- Update  feature branches with changes from main branch

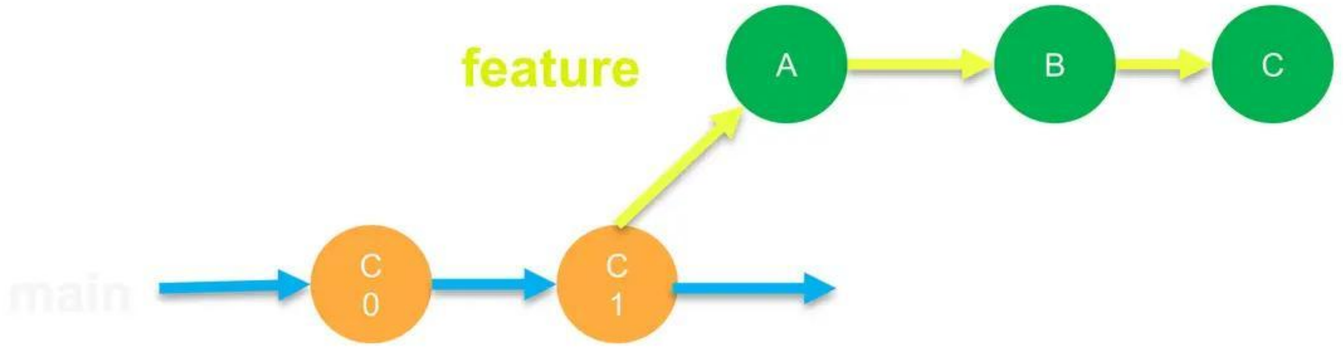
# Scenario 1 : Incorporating changes from Feature to Main



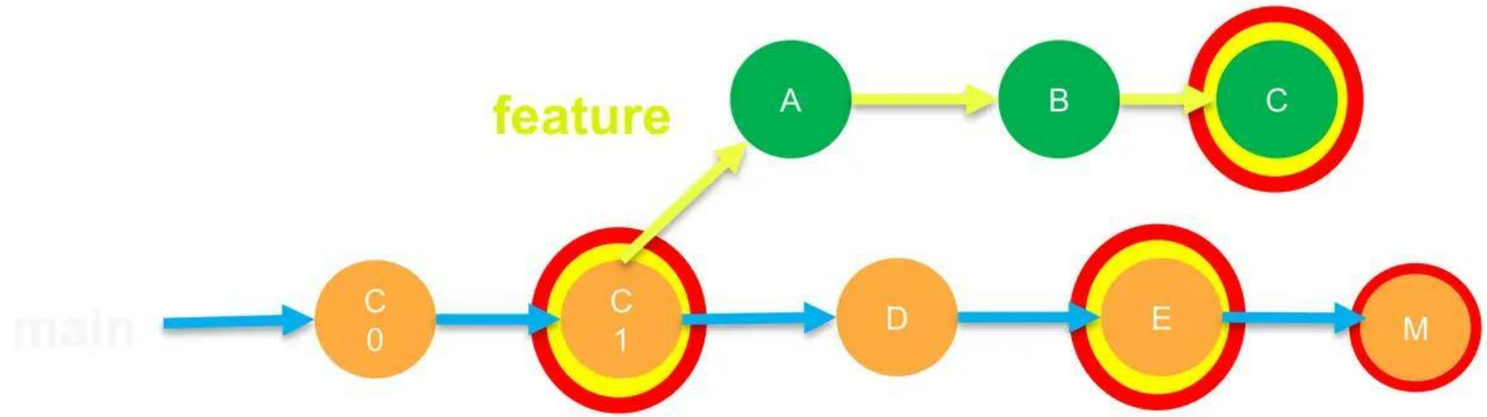

Fast Forward Merge




3-way Merge

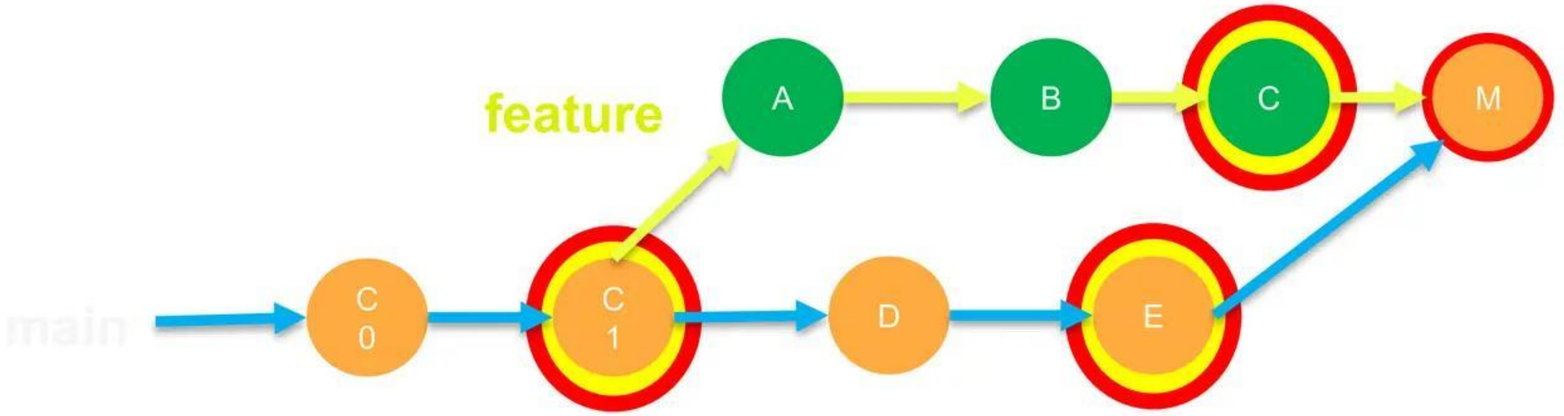



Fast Forward Merge



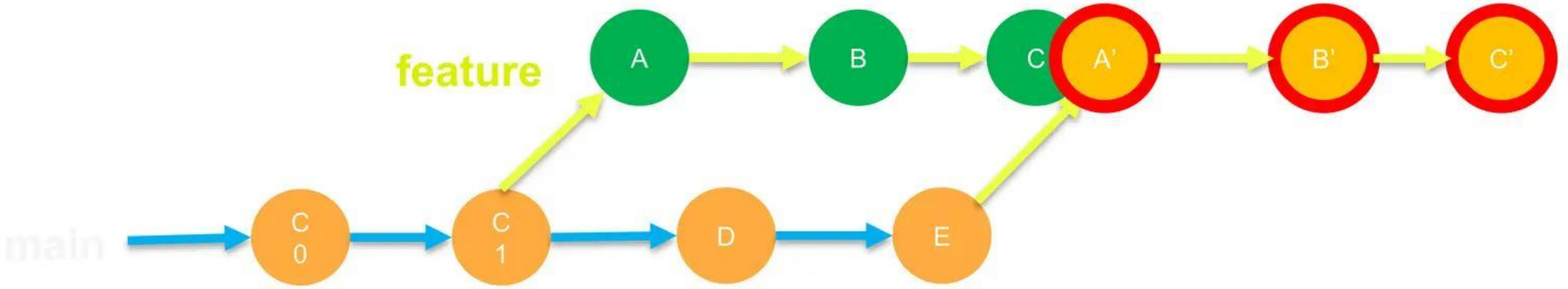

3-way Merge

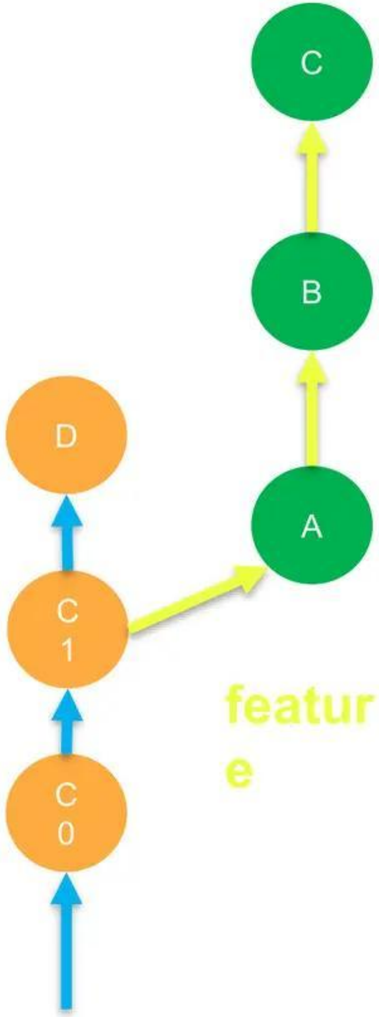
# Scenario 2 : Incorporating changes from Main to Feature



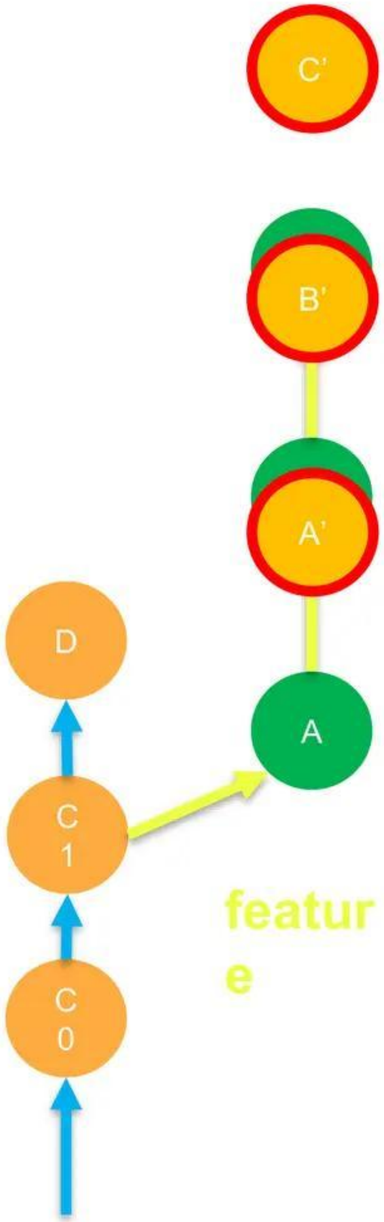

Merg
e

Rebase



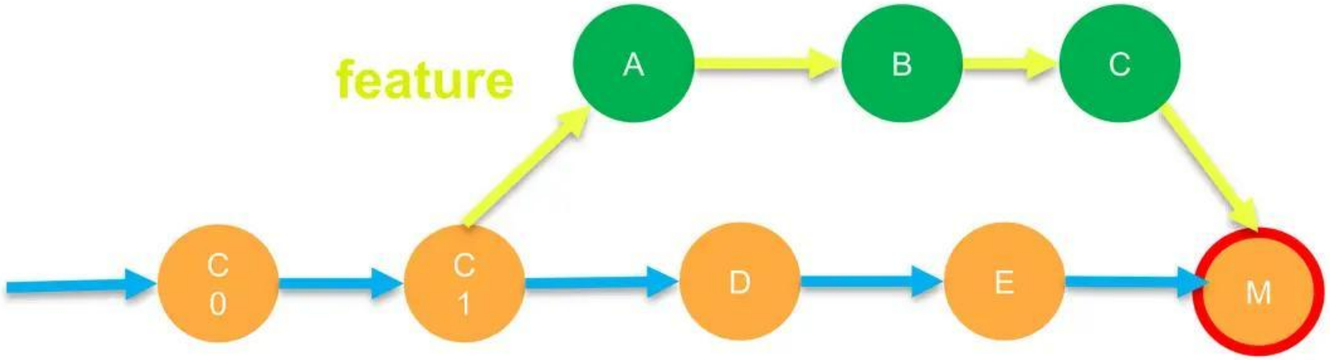



Remot
e

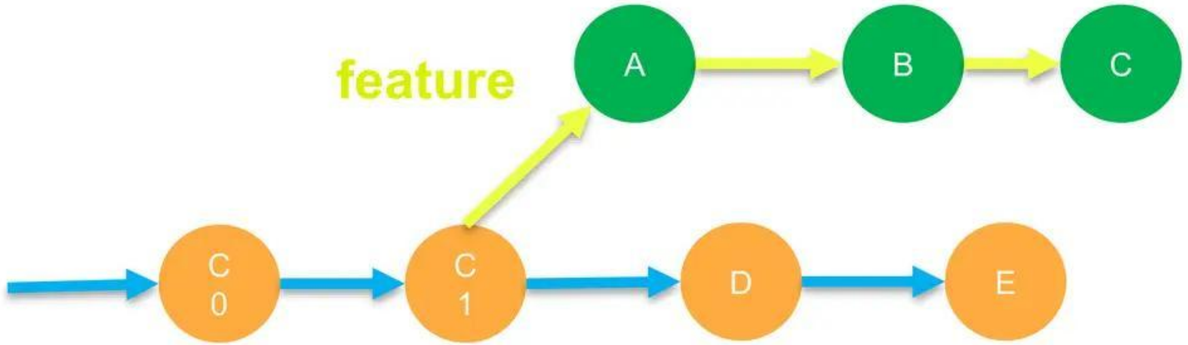



Local

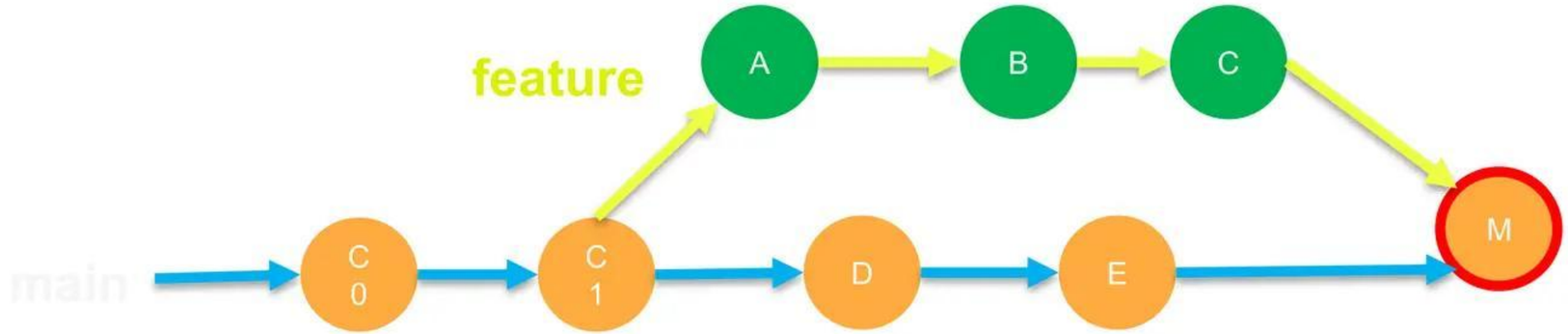
# Scenario 1 : Incorporating changes from Feature to Main



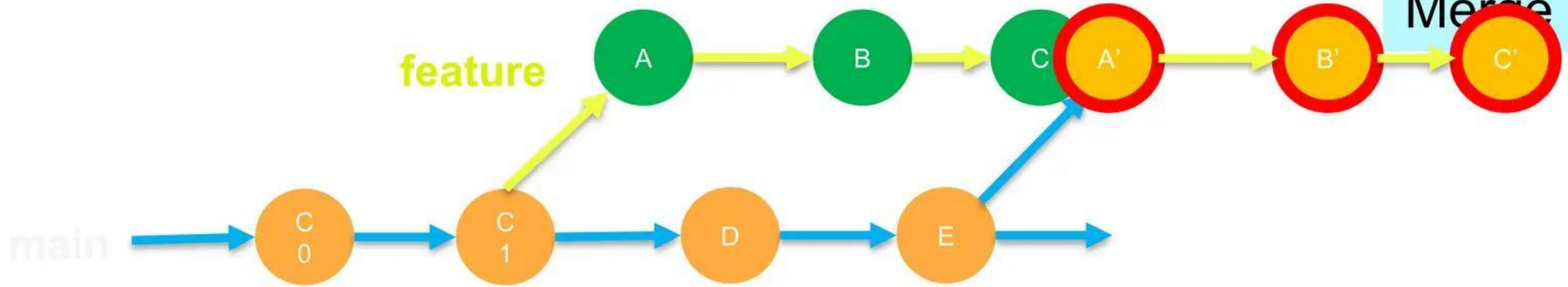

Merg
e




Rebase
??




Merg
e




Rebase and FF
Merge