

# Goals of a version control system

Version control records changes to a set of files over time.

This enables you to:

- Keep a history of your work
  - Summary commit title
  - See which lines were co-changed
- Checkpoint specific versions (known good state)
  - Recover specific state
- Binary search over revisions
  - Find the one that introduced a defect
- Undo arbitrary changes
  - Without affecting prior or subsequent changes
- Maintain multiple releases of your product

# Who uses version control?

## **Everyone should use version control**

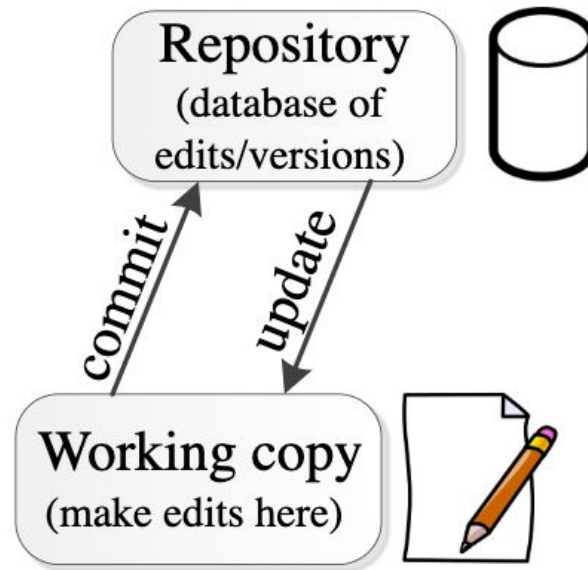
- Large teams (100+ developers)
- Small teams (2-10+ developers, like this course!)
- Yourself (and your future self)
  - Multiple features or multiple computers

## **Example application domains**

- Software development
- Research (infrastructure and data)
- Documents (See: “Version History” in Google Docs)

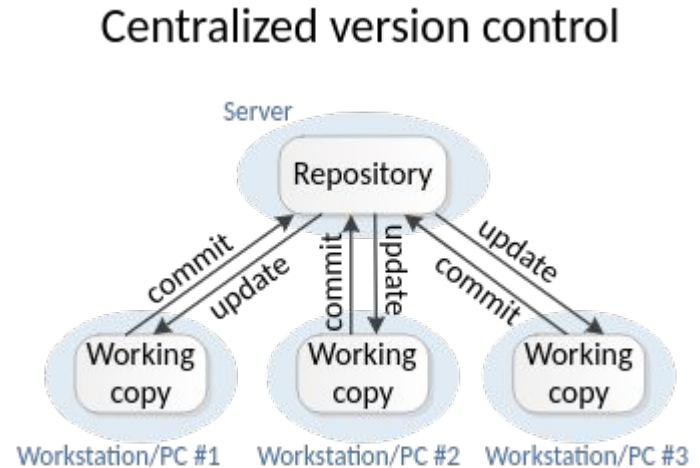
# Version Control

Working by yourself



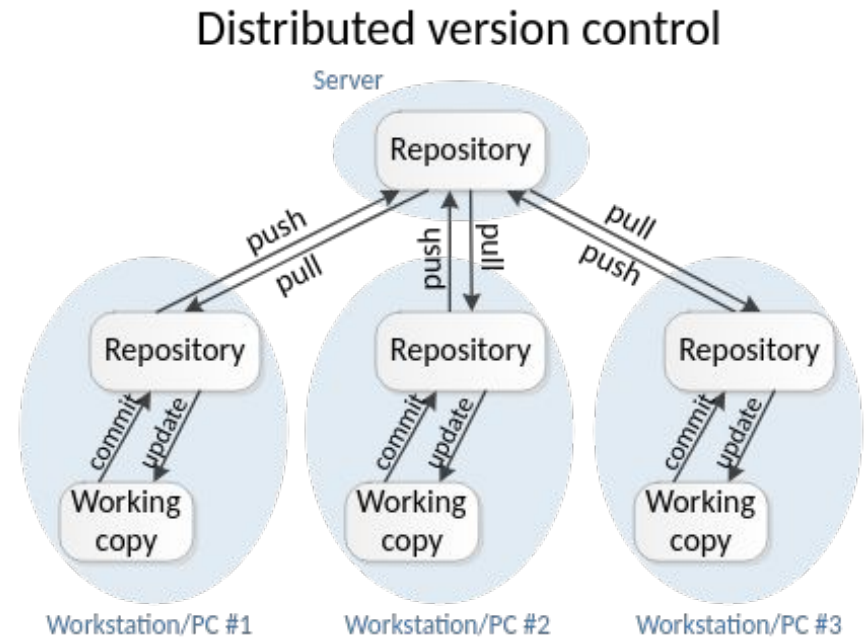
# Centralized version control (the old way)

- **One central repository.**
- All users **commit** their changes to a **central repository**.
- Each user has a working copy. As soon as they commit, the repository gets updated.
- Examples: SVN (Subversion), CVS.



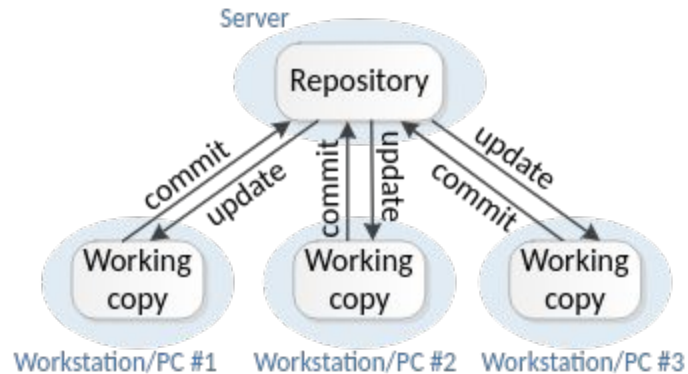
# Distributed version control (the new way)

- **Multiple copies of a repository.**
- Each user **commits** to a **local** (private) repository.
- All committed changes remain local unless **pushed** to another repository.
- No external changes are visible unless **pulled** from another repository.
- Examples: Git, Hg (Mercurial).

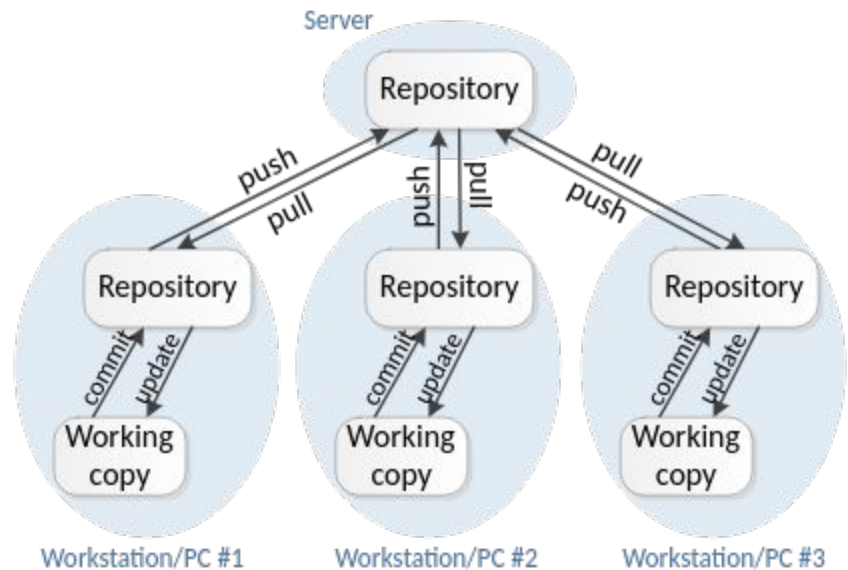


# 2 different version control modes

## Centralized version control



## Distributed version control



**Branch**

**vs**

**Clone**

**Vs**

**Fork**



# Multiple versions of your program

---

What if you have to support:

- Version 1.0.4 vs version 2.0.0
- Windows vs macOS
- Adding a feature
- Fixing a bug

Git handles these!

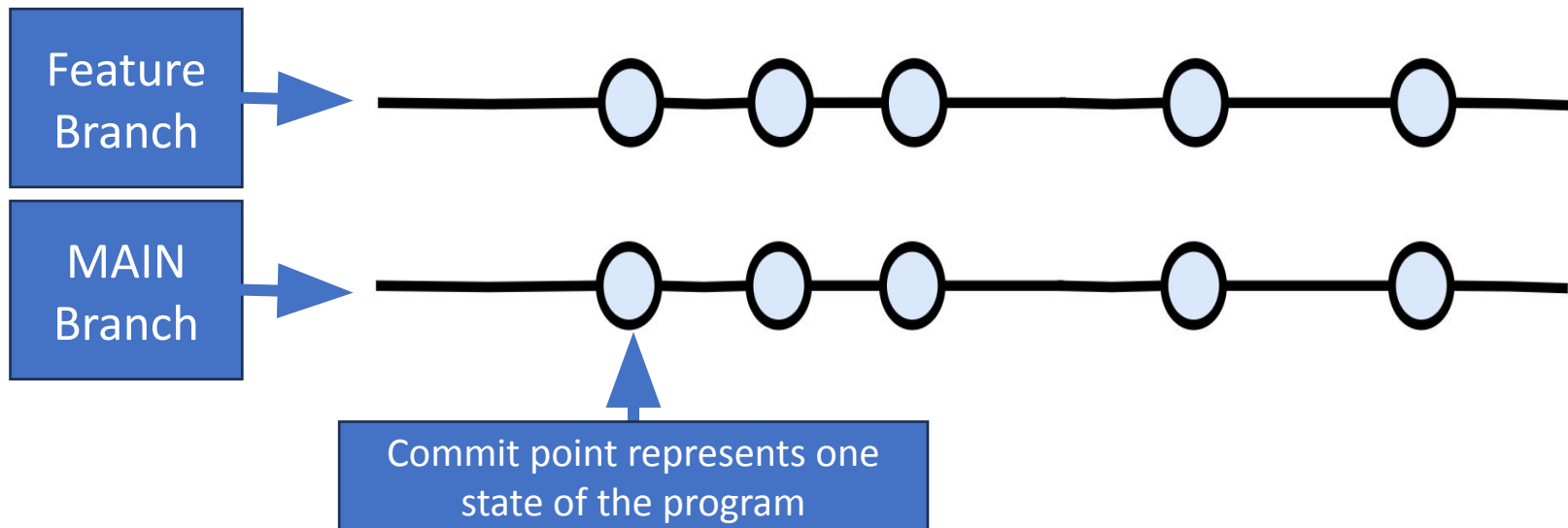
- Branch: Start a parallel history of changes to the code in the repository
- Clone: Make a copy of the repository to work on code changes
- Fork: Make a copy the repository that will not necessarily be merged back with original (but can be through a pull request)



# Branches

---

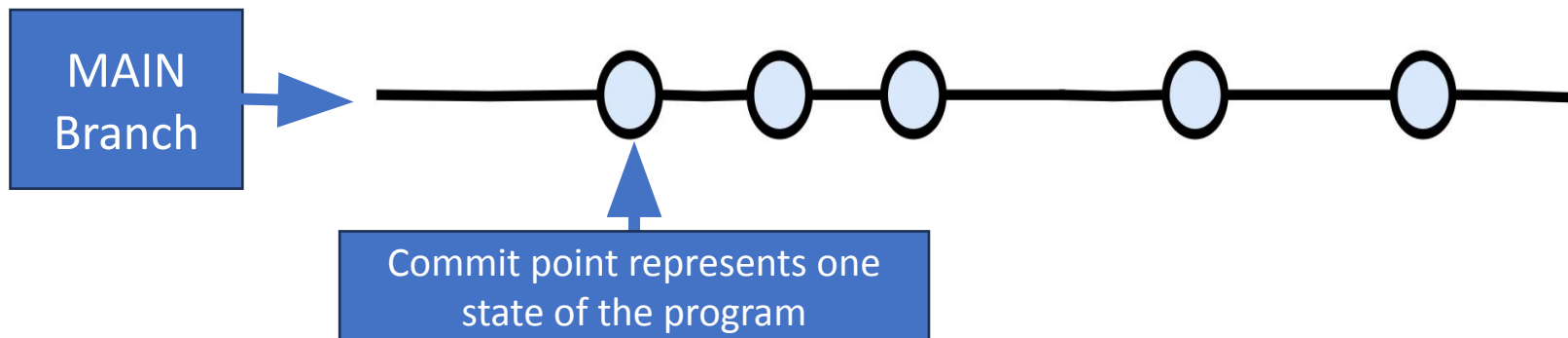
- Git has a basic concept of a branch
  - Branch: one history of changes to the code
- You can have many branches
  - Lightweight - every work item (feature, bug) has its own branch



# Branches

---

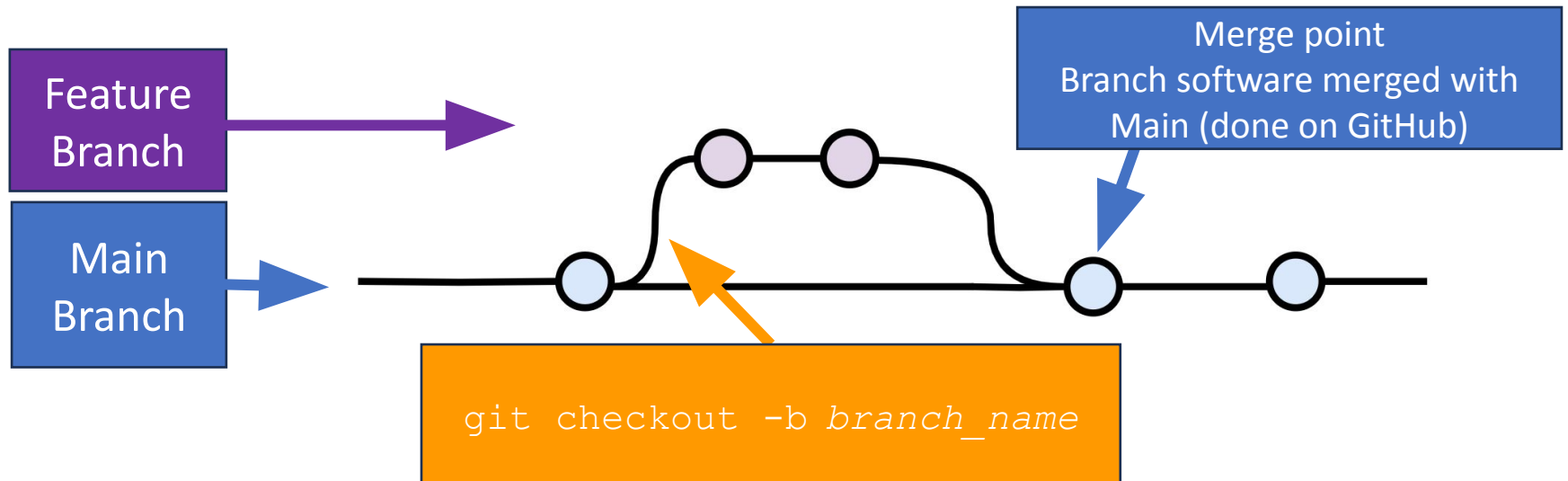
- There is one **main** development **branch** (called “main”)
- This is considered your latest working version of the code
  - You should always be able to ship “**working software**” from main



# Branches

---

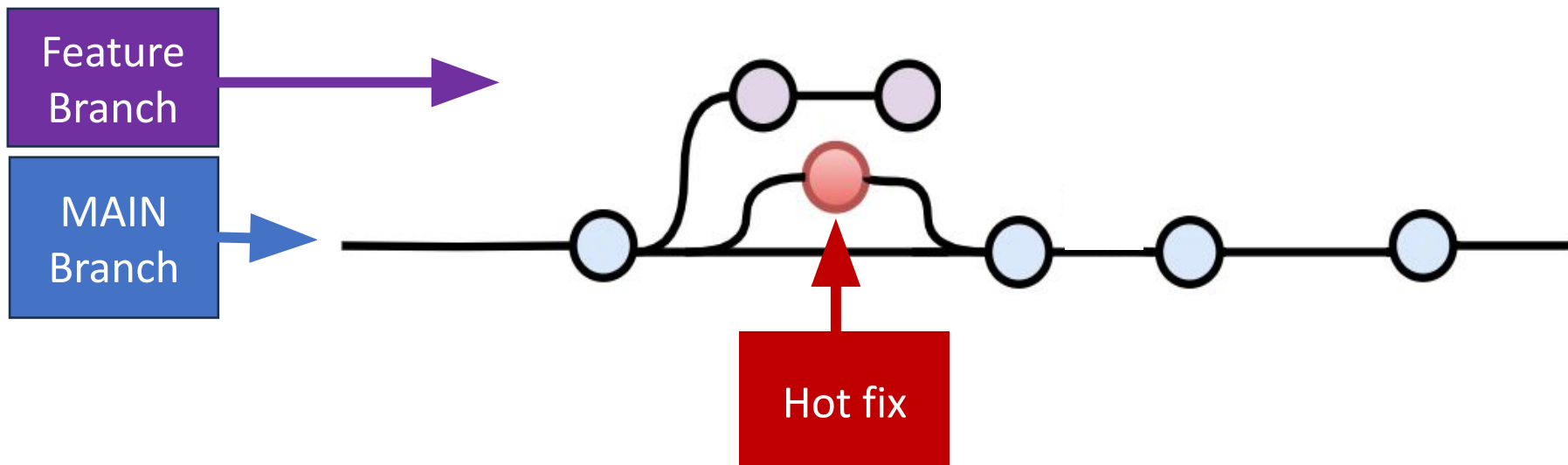
- You can have many branches
  - Life goal of a branch is to be merged into main and deleted as quickly as possible
- To develop a feature or bug fix, create a new branch
  - And then later merge it with Main
  - **Why is this a good practice?**



# Branches

---

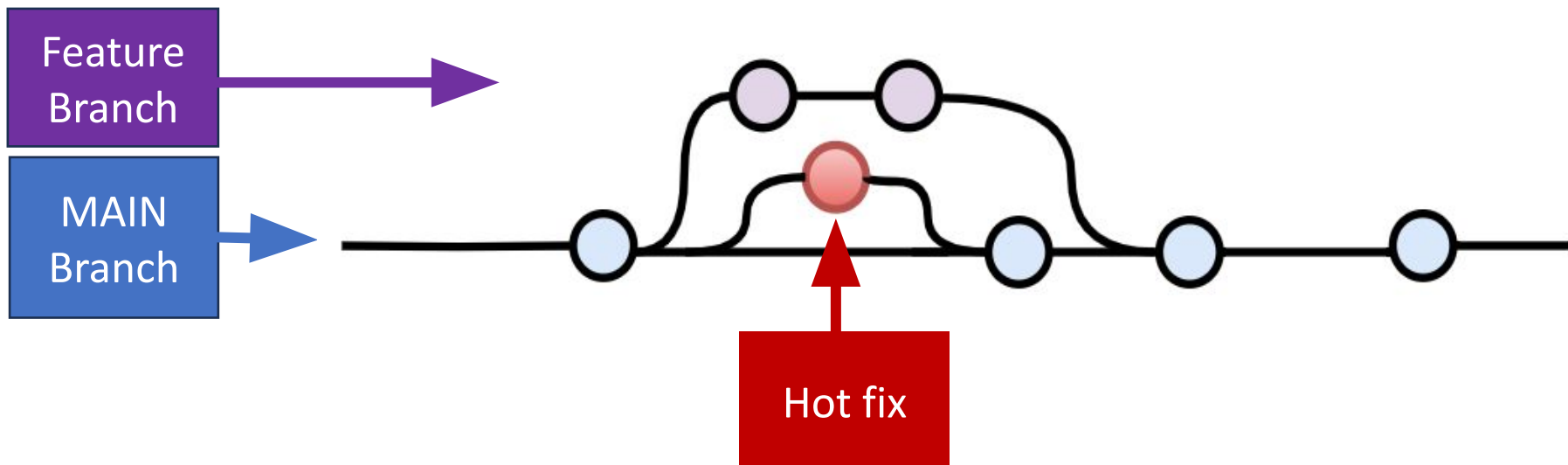
- To develop a feature or bug fix, add a new branch
  - Why? Keeps Main **always working** and allows for **parallel development**
    - It's ok to have many branches



# Branches

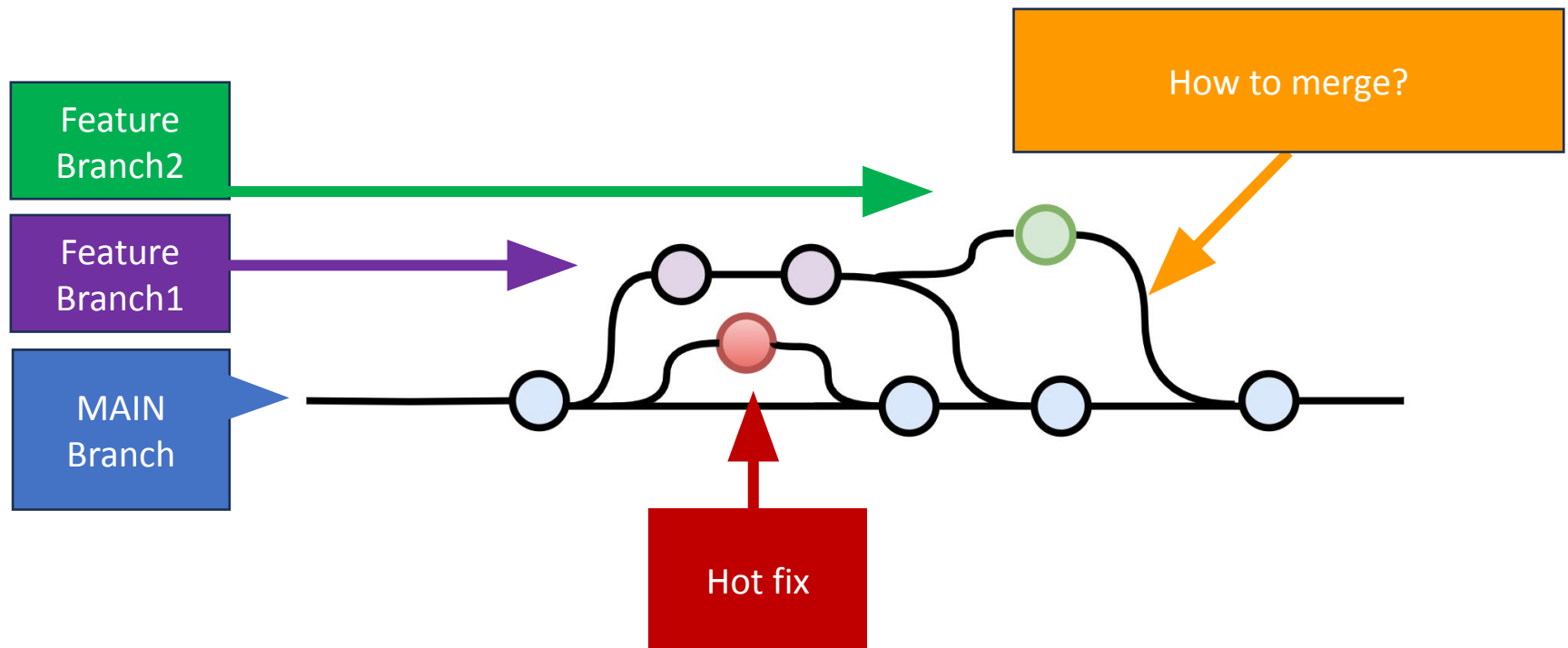
---

- To develop a feature or bug fix, add a new branch
  - Why? Keeps Main **always working** and allows for **parallel development**



# Branches

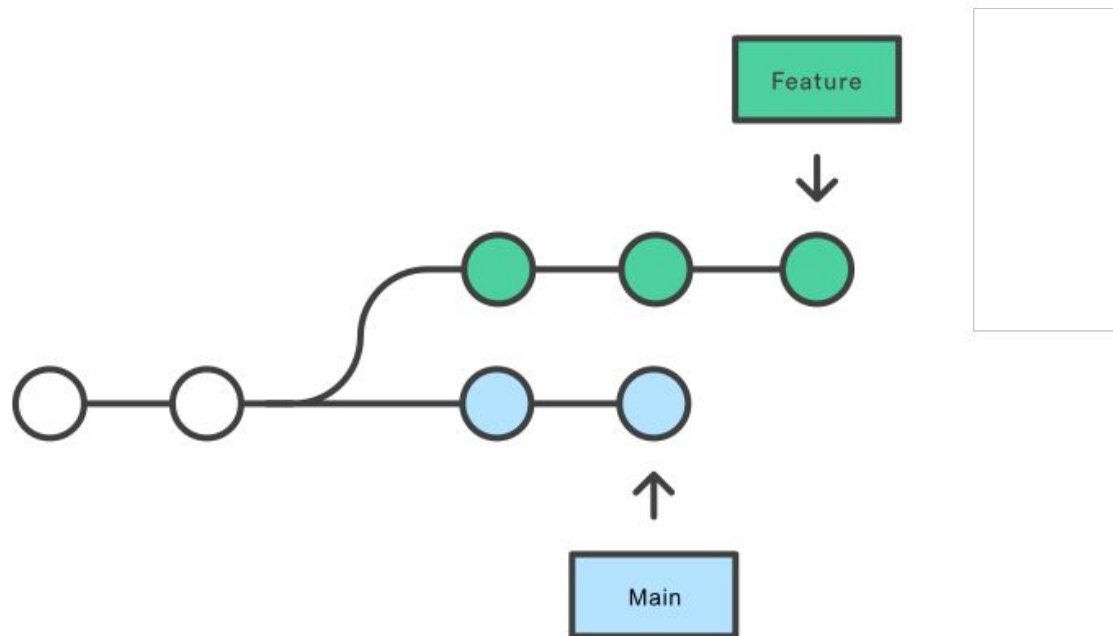
- To develop a feature or bug fix, add a new branch
  - Why? Keeps Main **always working** and allows for **parallel^2 development**



# Merging

# Scenario where merging is possible

Developing a feature in a dedicated branch

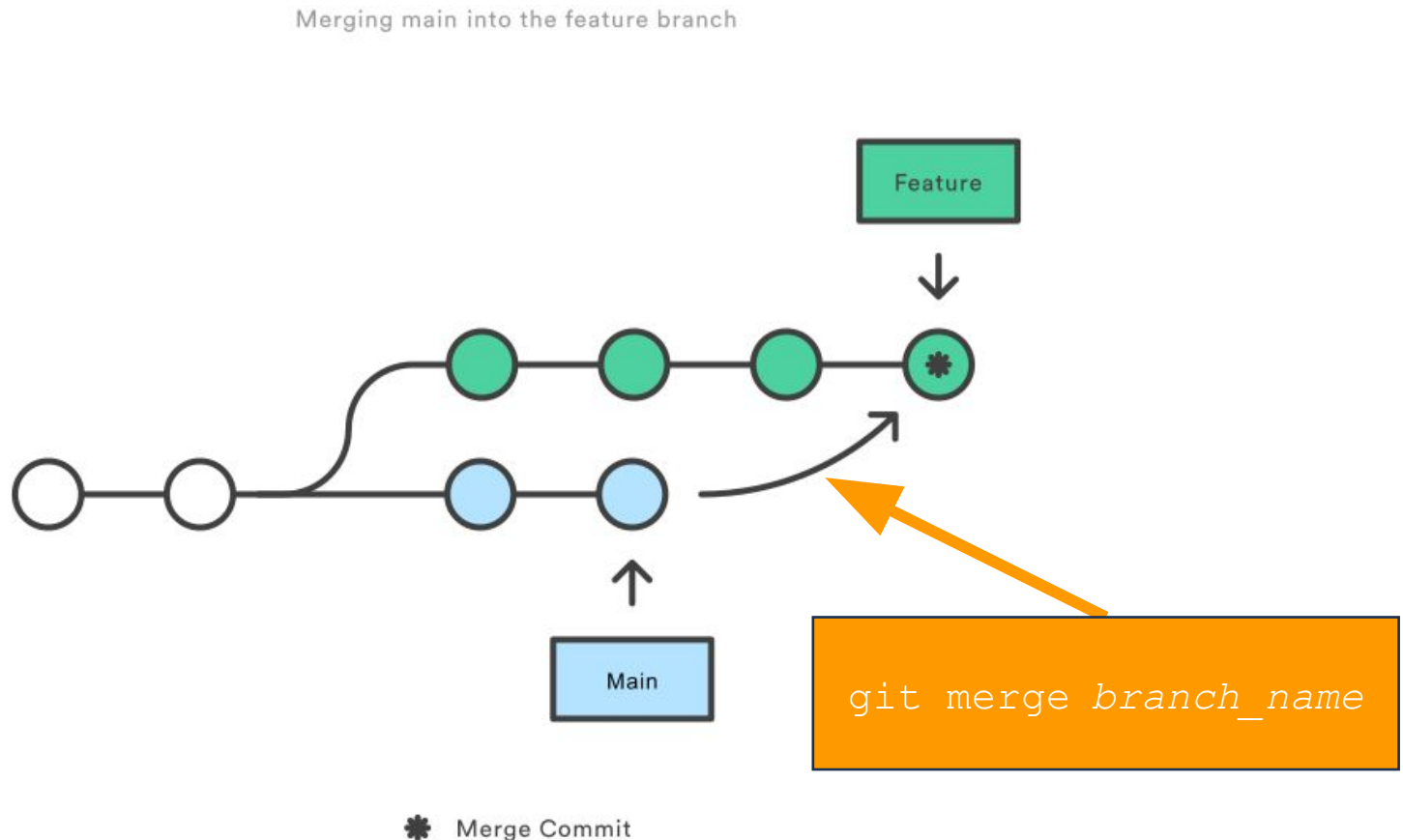


(2 types: **from** main and **into** main)

<https://www.atlassian.com/git/tutorials/merging-vs-rebasing>



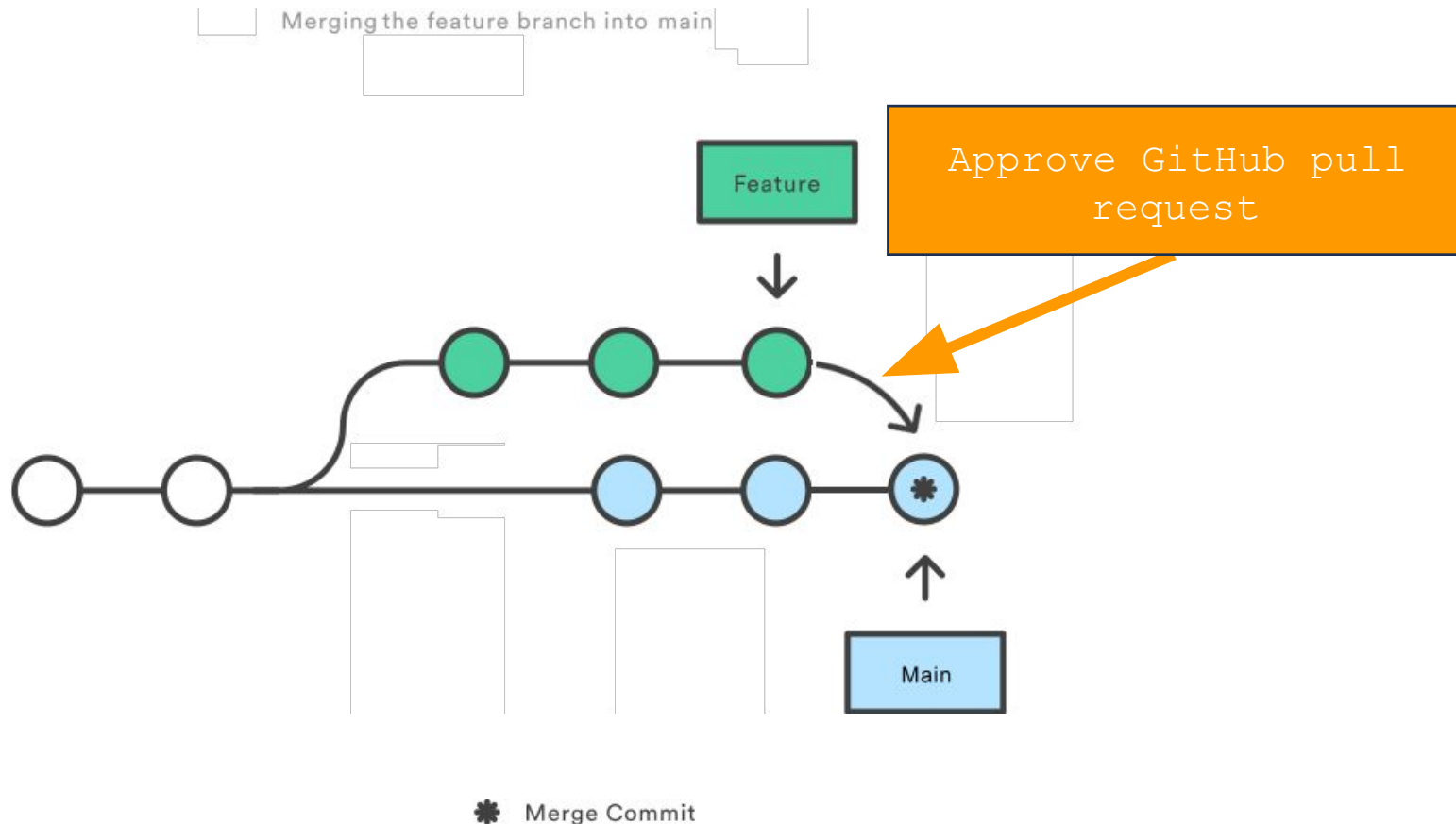
# Merge (integrating changes from main)



First merge: resolve conflicts before pushing to main

<https://www.atlassian.com/git/tutorials/merging-vs-rebasing>

# Merge (integrating changes into main)

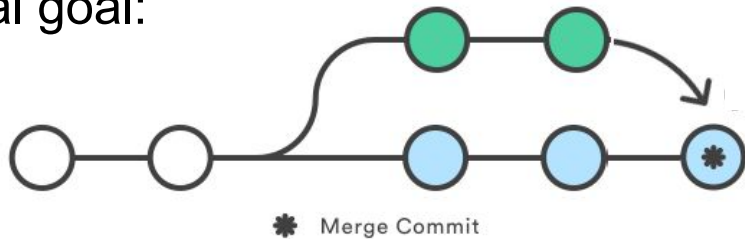


Second merge: pull request to get your changes into main

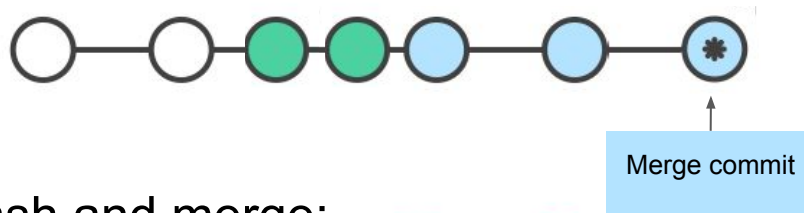
<https://www.atlassian.com/git/tutorials/merging-vs-rebasing>

# Merge: Squash & merge on GitHub

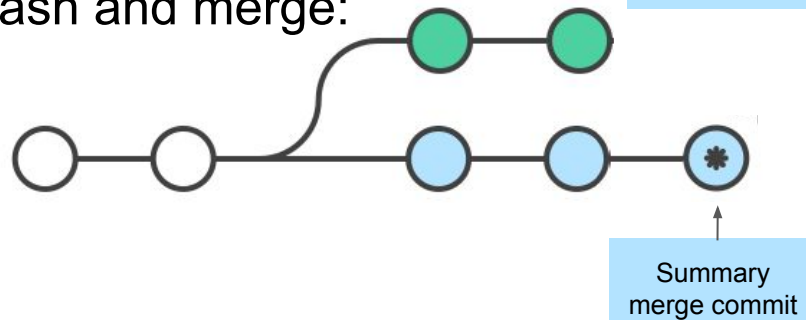
Initial goal:



Merge:



Squash and merge:



## Create a merge commit

All commits from this branch will be added to the base branch via a merge commit.

## ✓ Squash and merge

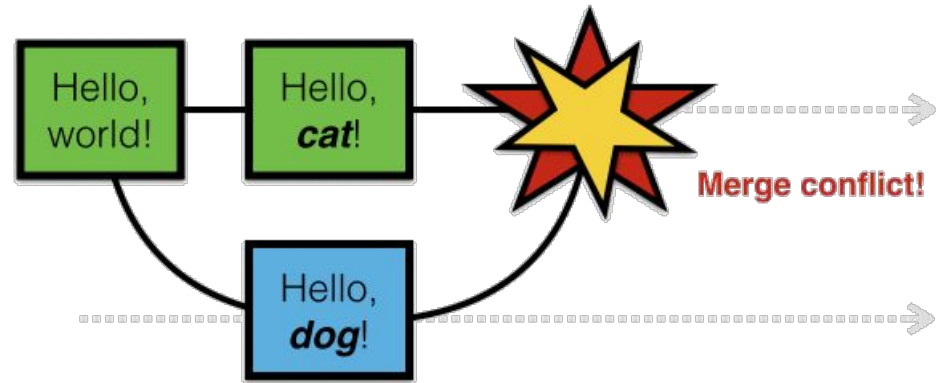
The 14 commits from this branch will be combined into one commit in the base branch.

## Rebase and merge

The 14 commits from this branch will be rebased and added to the base branch.

Merge conflicts

# Conflicts



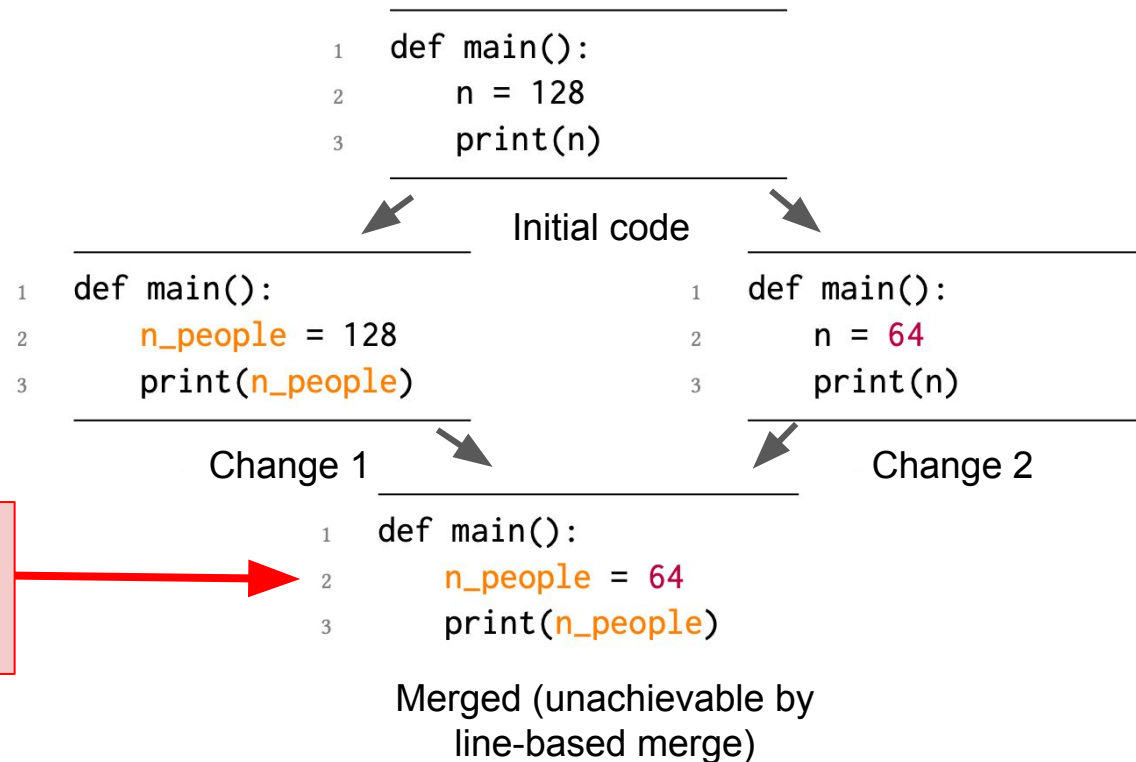
- **Conflicts** arise when two users **change the same line** of a file
- When a conflict arises, the person doing the merge needs to resolve it by manual inspection

How to avoid merge conflicts?

# Merge Algorithm: May Fail to Make a Merge

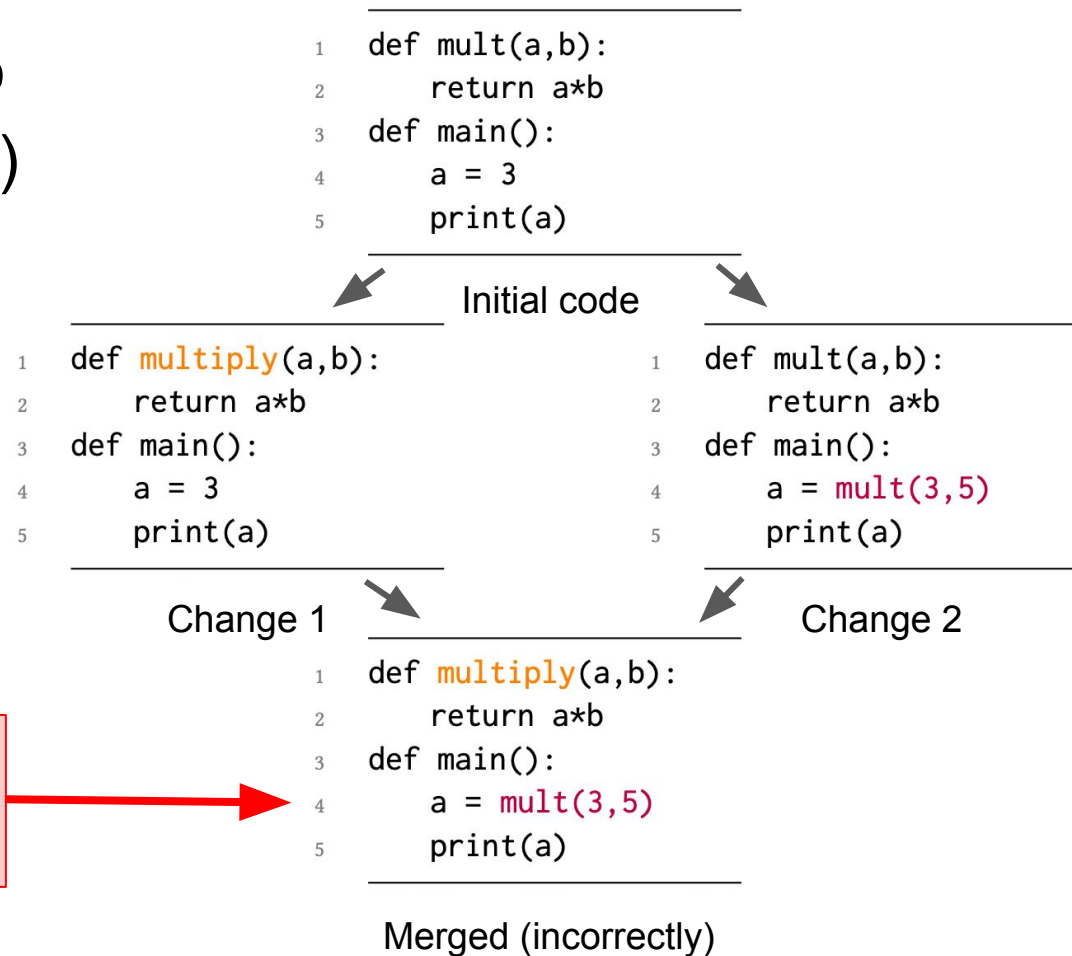
- Line-by-line merge reports a problem
- Inspection reveals they can be merged

Still works despite  
2 changes



# Merge Algorithm: Falsely Successful Merge

- Can merge cleanly (no textual merge conflicts)
- Resulting code is incorrect



How to avoid merge conflicts



# Synchronize with teammates often

- Pull often

- Avoid getting behind the main branch

- Push as often as practical

- Don't destabilize the main build
- Use continuous integration (automatic testing on each push, even for branches)
- Avoid long-lived branches

# Commit often

- On the main branch:
  - 1. Every commit should address one concept
  - 2. Every concept should be in one commit
- On feature branches:
  - 1. Make single-concern commits (see next slide)
  - 2. From branch back into main: squash and merge
- Easier to understand, review, merge, revert

# Make single-concern commits

- Do only one task at a time
  - Commit after each one
- Create a branch for each simultaneous task
  - Need to keep track of all your branches, merge
  - Easier to share work with teammates
- Do multiple tasks in one working copy with multiple branches
  - Commit only a subset of files (use Git's “staging area” with **git add**)

# Do not commit all files

Use a .gitignore file (templates on Github)

Don't commit:

- Binary files
- Log files
- Temporary files

# Plan ahead to avoid merge conflicts

- Modularize your work

- Divide work so that individuals or subteams “own” a module
- Other team members only need to understand its specification
- Requires good documentation and testing

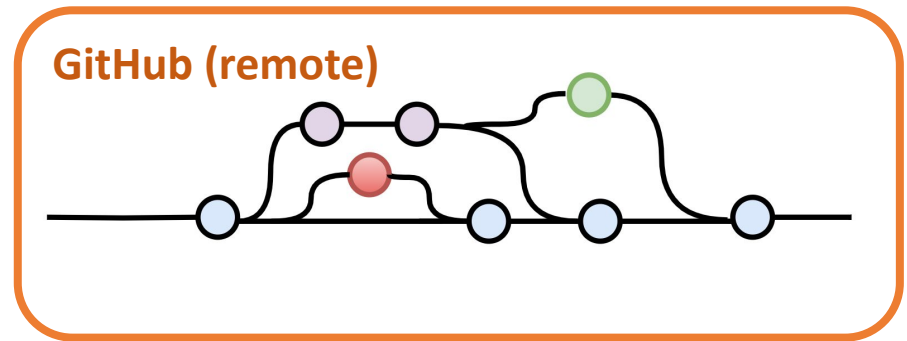
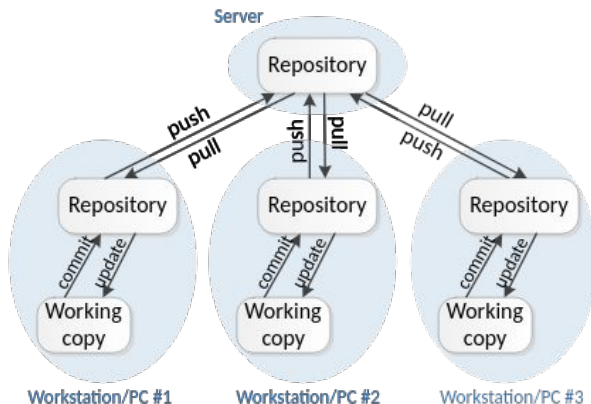
- Communicate about changes that may conflict

- Examples (rare!): reformat whole codebase, move directories, rename fundamental data structures

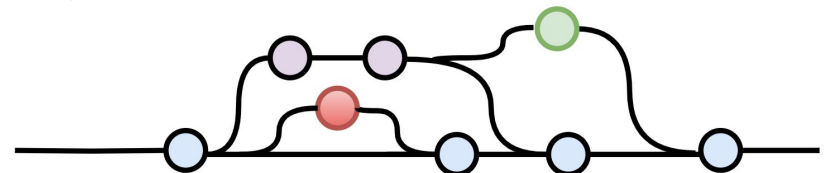
# **Git workflow and usage**

# Cloning

- When you **clone** a repo you are creating a **local copy** on your computer that you can sync with the remote
- Ideal for contributing directly to a repo alongside other developers
- After a clone, you can use **git push** to send local changes to remote repo



Clone  
(copy – often on remote (local) host)

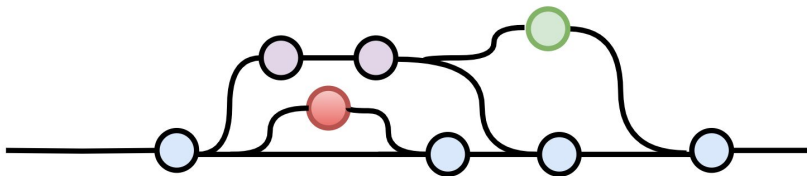


# Forking (GitHub concept, not a git concept)

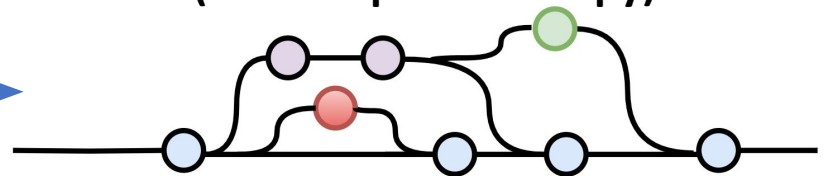
---

- Creates a **new, unrelated repository** (GitHub project) that is initially an exact copy
- Changes to either repository do not affect the other
- Allows you to evolve the repo without impacting the original
- If original repo is deleted, forked repo will still exist

GitHub



Fork  
(full independent copy)



- It's possible to update the original but only with **pull requests (original owner approves or not)**



# Git: workflows

<start day>

```
git pull
```

```
git checkout -b name
```

<work on a task>

```
git commit
```

```
git commit
```

<run tests>

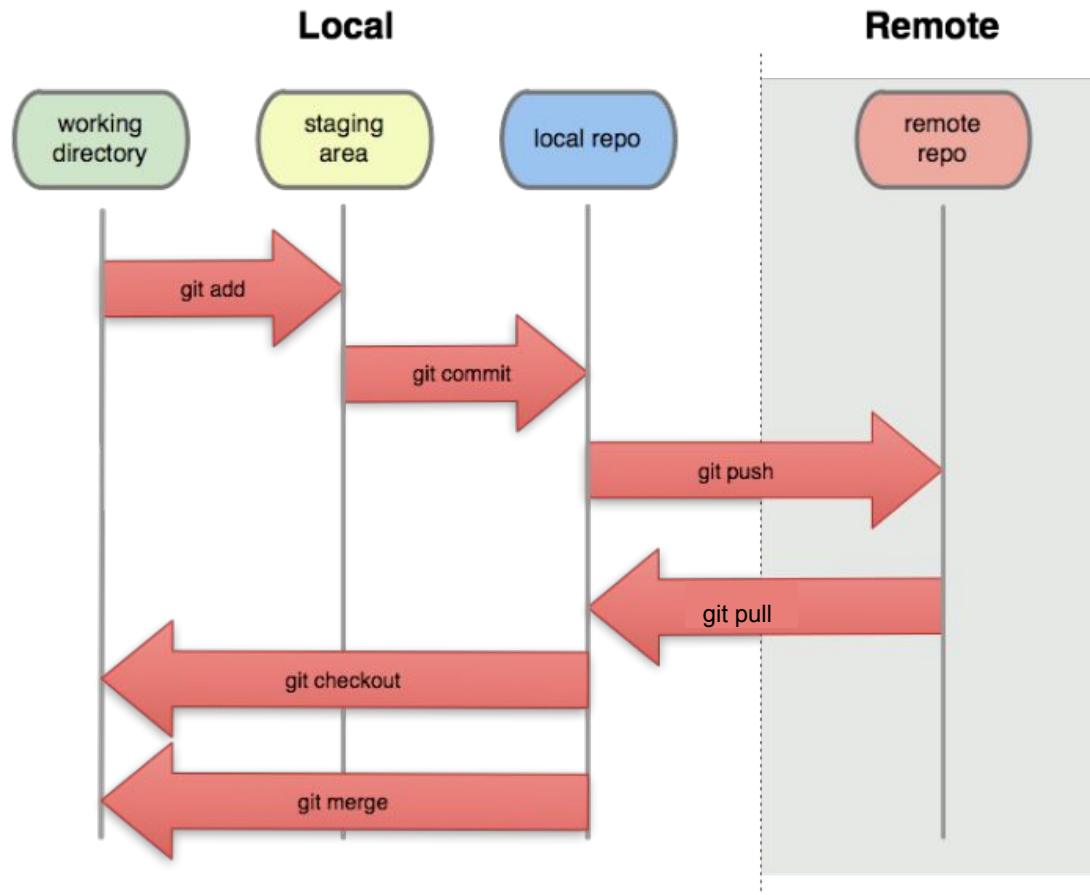
```
git merge
```

```
git push
```

<start another another task>

```
git checkout -b name2
```

<repeat>



# Learn more!

- Learn about git - many resources available for tips and practices
  - Michael Ernst: [VC Concepts](#) and [Pull Requests](#)
  - Atlassian [merge vs rebase](#)
  - Git [branching and merging](#)
  - Video tutorial “[Git, GitHub, & GitHub Desktop](#)”