Version control and Git

CSE P 504

Why use version control?

11:51pm

Why use version control?

11:51pm

11:57pm

Why use version control? – backup/restore

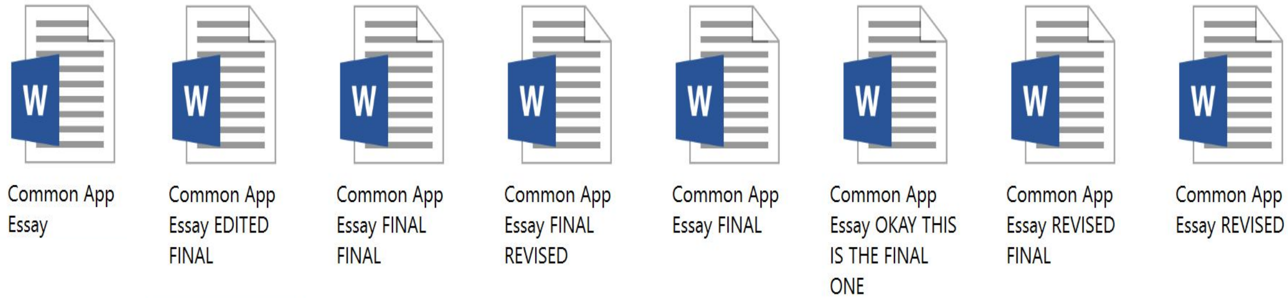
11:51pm

11:57pm

11:58pm

11:59pm

Why use version control? – teamwork



How are you going to make sense of this?

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)
- Yourself (and your future self)
  - Multiple features or multiple computers

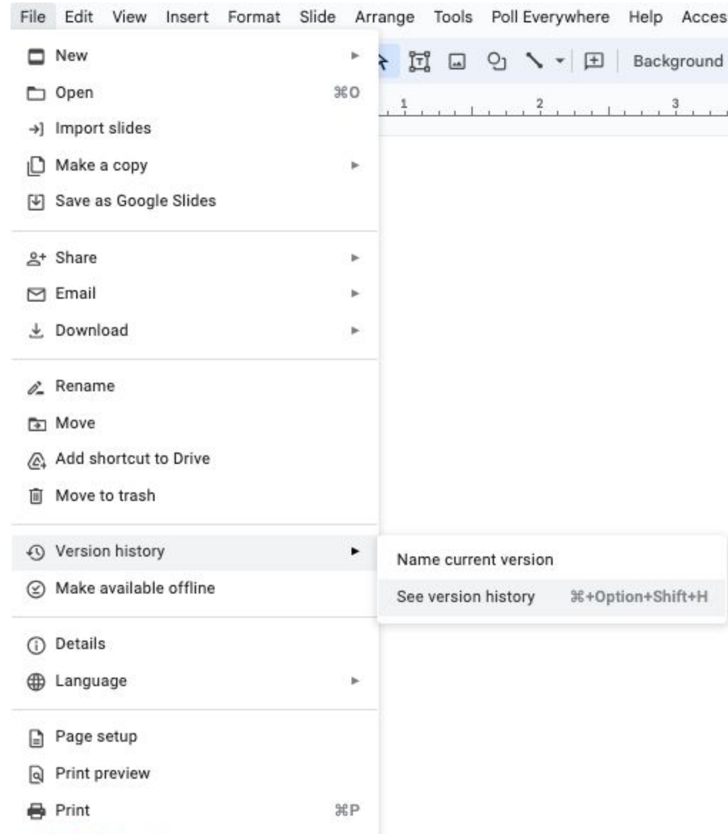
Example application domains

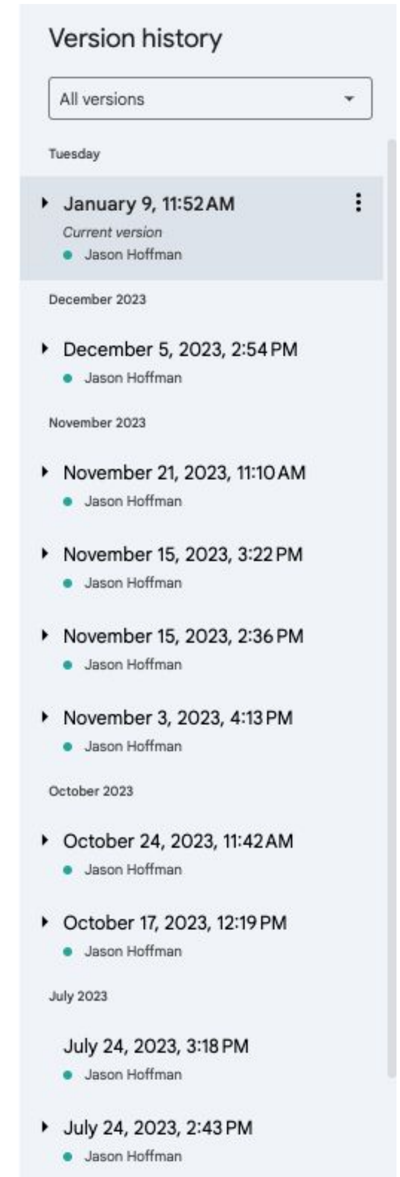
- Software development
- Experiments (infrastructure and data)
- Documents

Version control for documents

Common App Essay

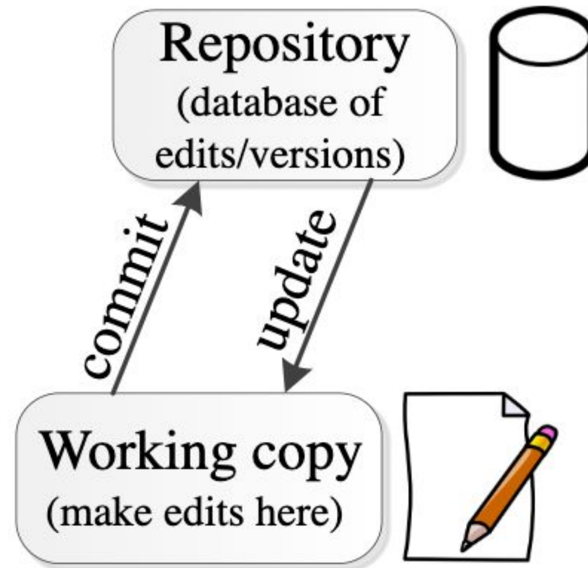
11:51pm





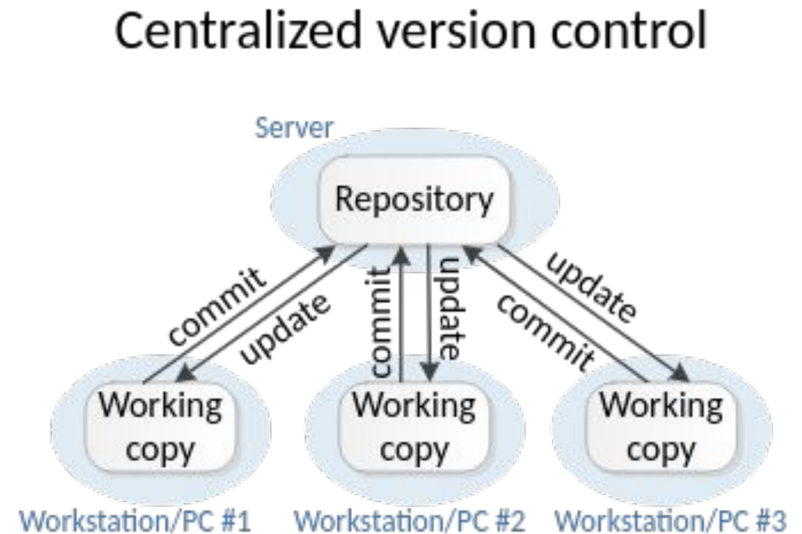
Version control

Working by yourself



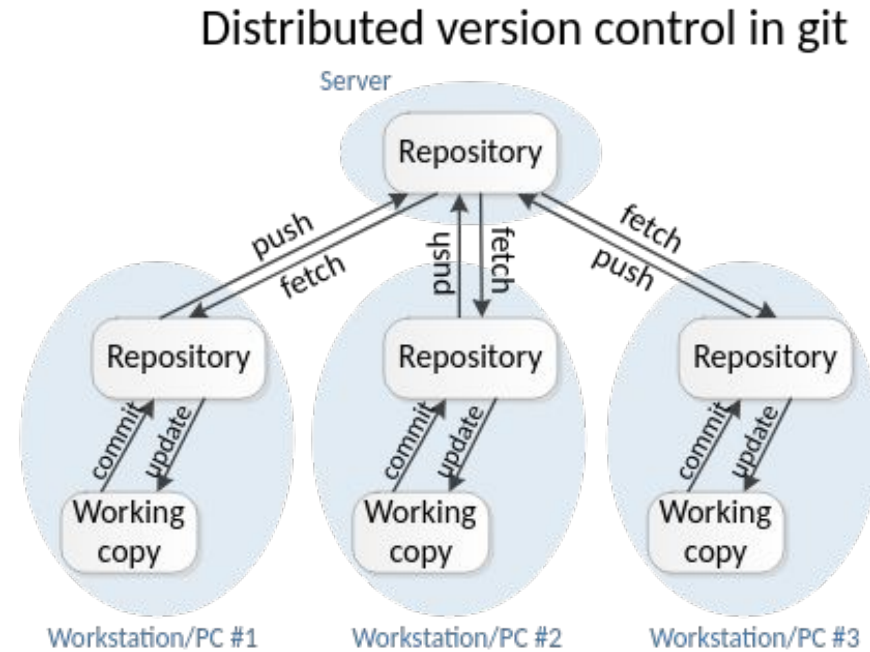
Centralized version control (the old way)

- **One central repository**.
  It stores a history of project versions.
- Each user has a **working copy**.
- A user **commits** file changes
  to the repository.
- Committed changes are immediately
  visible to teammates who **update**.
- Examples: SVN (Subversion), CVS.

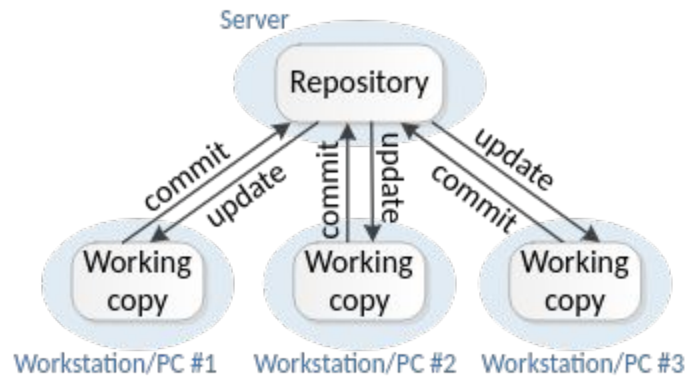


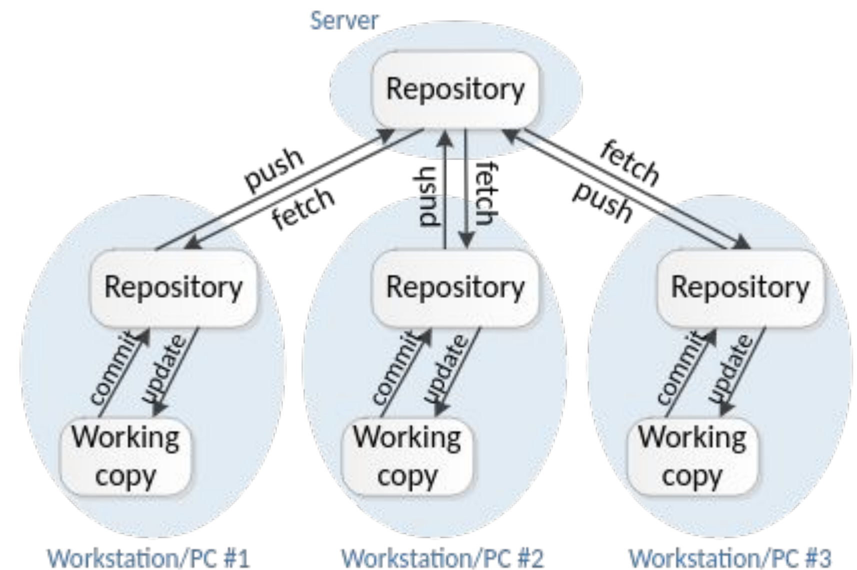
Distributed version control (the new way)

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



2 different version control modes





Branch
vs
Clone
Vs
Fork

Multiple versions of your program

What if you have to support:

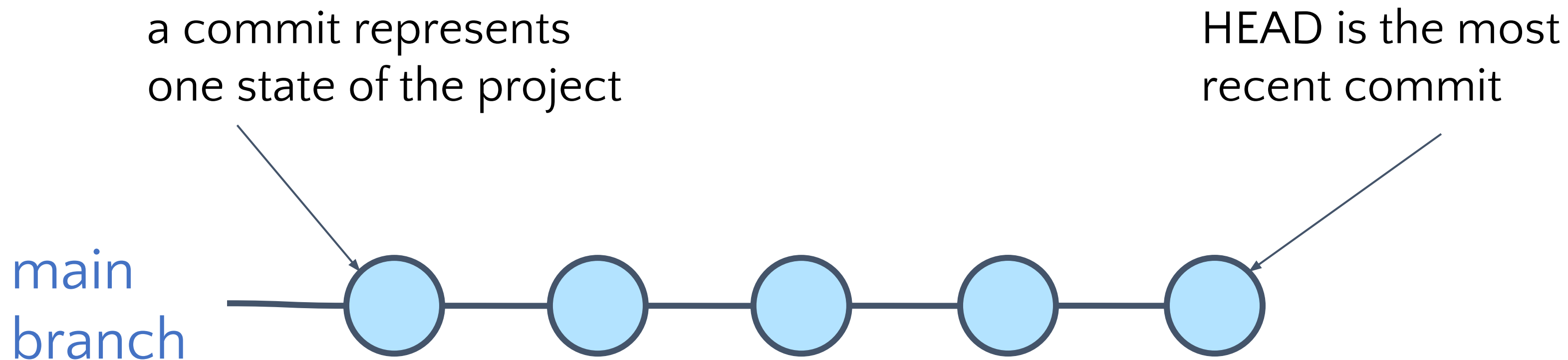
- Version 1.0.4 and version 2.0.0
- Windows and macOS
- Adding a feature
- Fixing a bug

Git has 3 ways to represent multiple histories:

- **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)

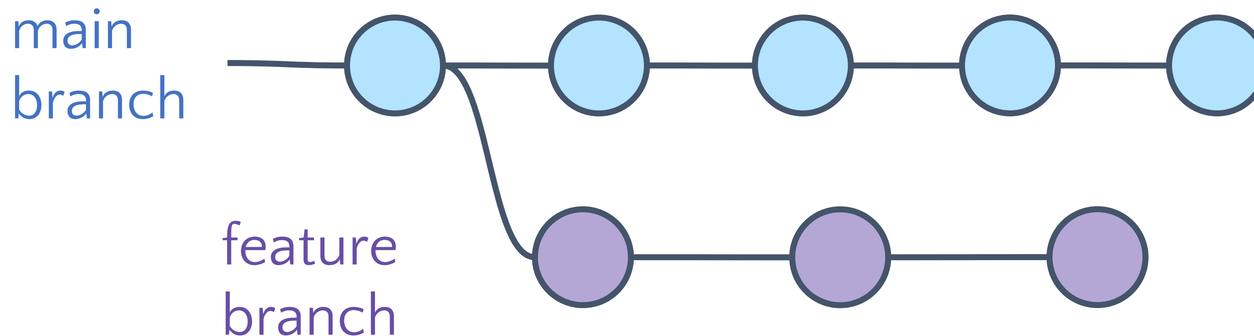
Branches

- A branch is a history of program versions
- There is one main development branch (main, master, trunk)
  - It should always pass tests and be ready to ship or deploy



Branches

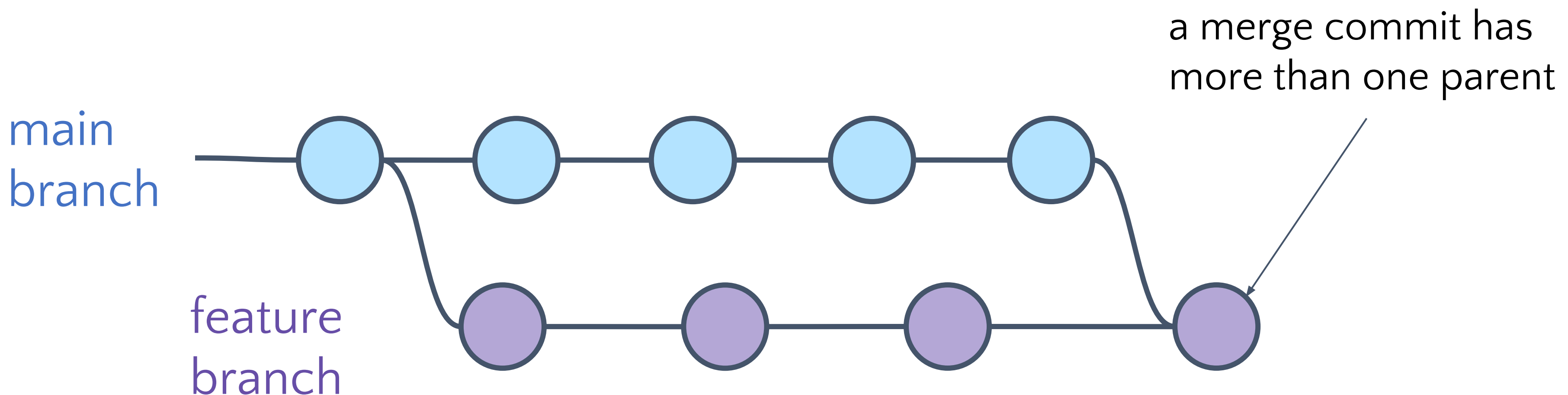
- Other branches are alternate histories
- You can have many branches
  - Lightweight - every work item (feature, bug) has its own branch
    - Why is this a good practice?
- Branches (histories) can get out of sync



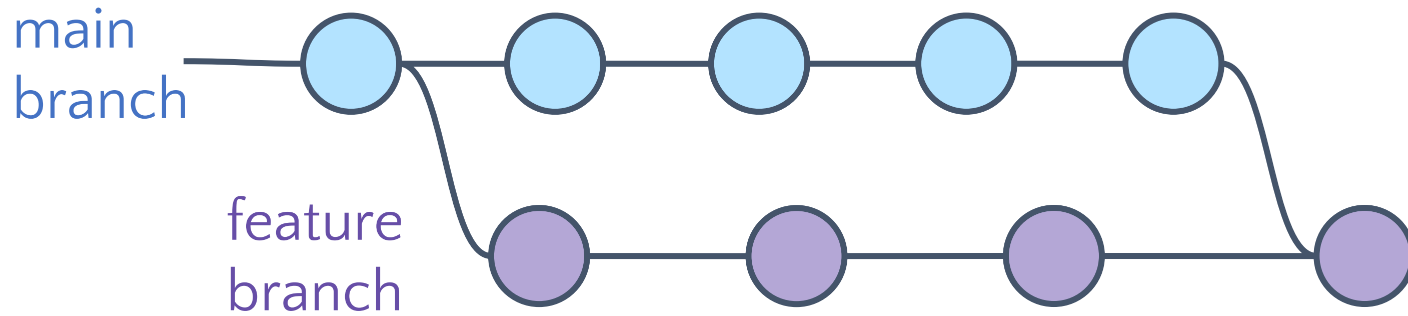
Merging branches

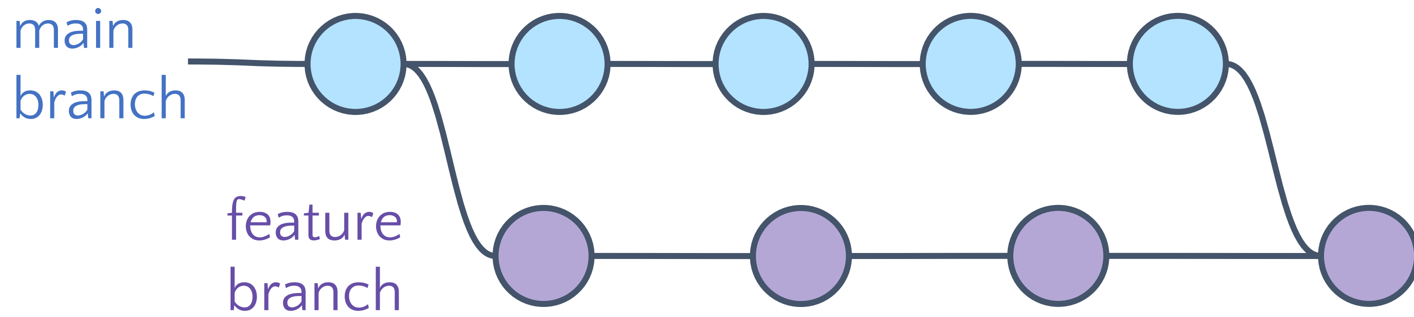
- Branches can get out of sync
- **Merge** incorporates changes from one branch into another
- From feature branch: `git merge main`

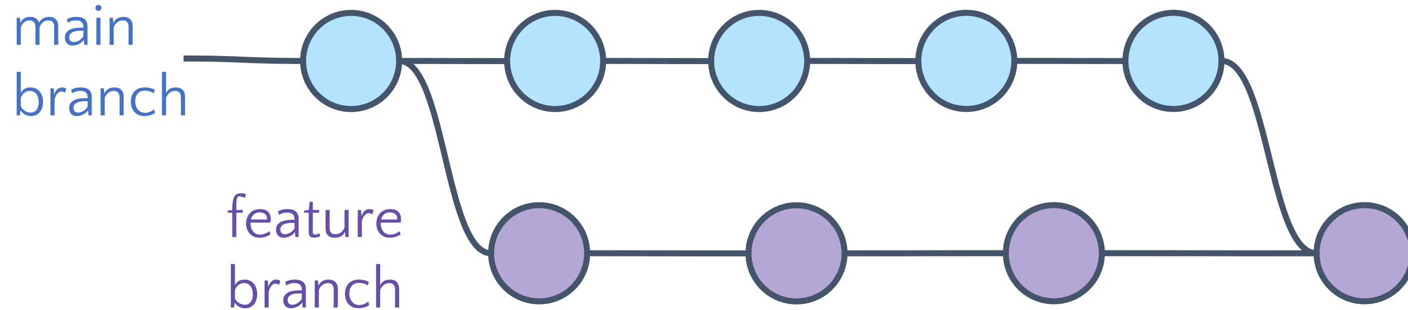
- Life goal of a branch is to be merged into main and deleted as quickly as possible
  - Done via a **pull request**, not via `git merge`



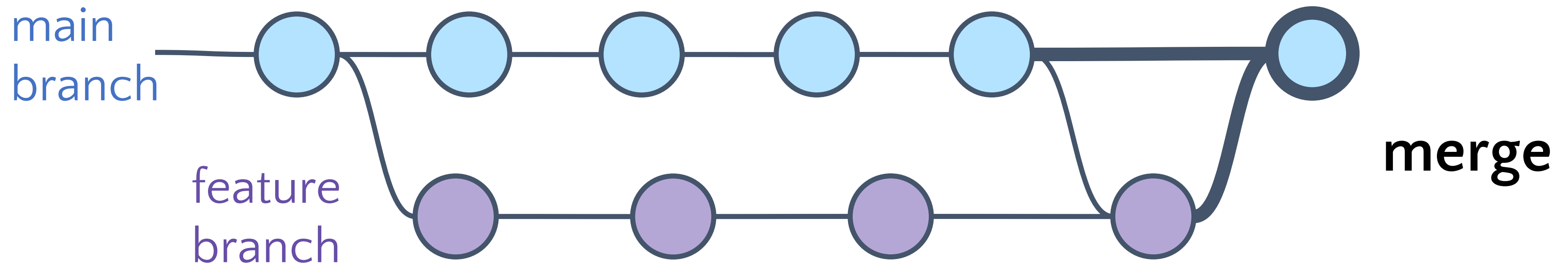
3 ways to resolve a pull request

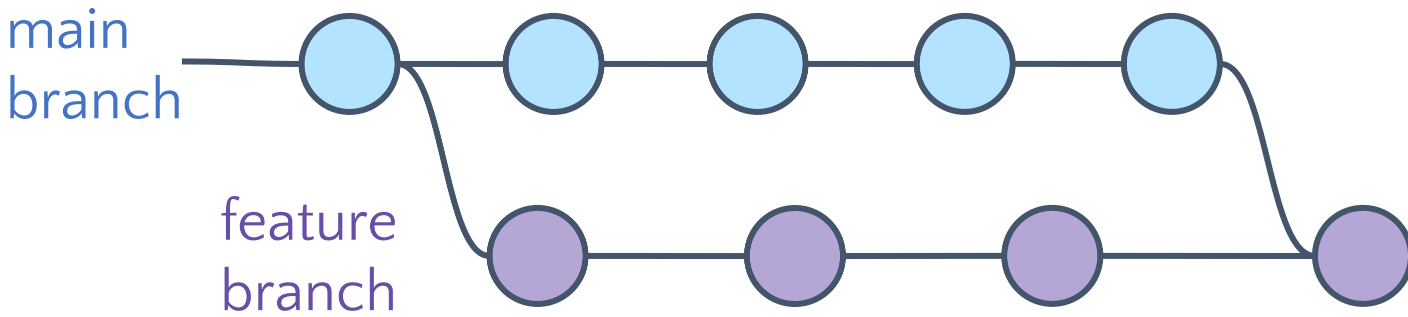


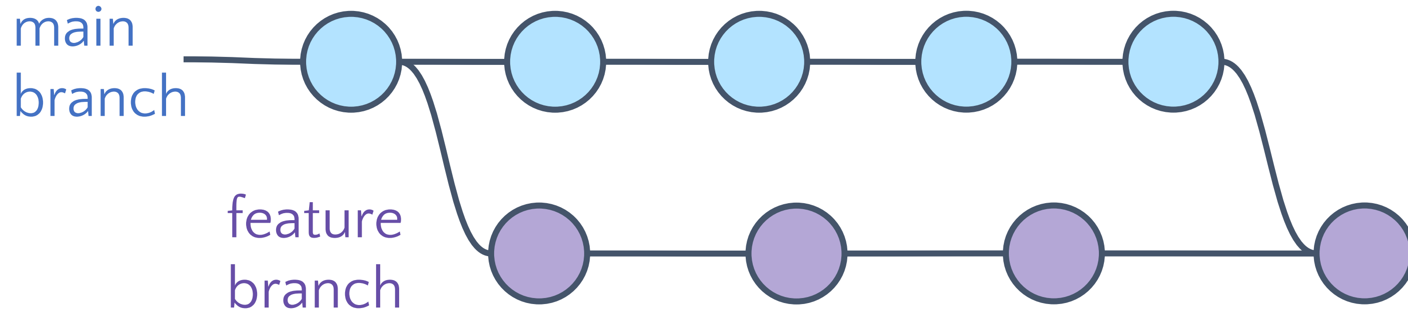




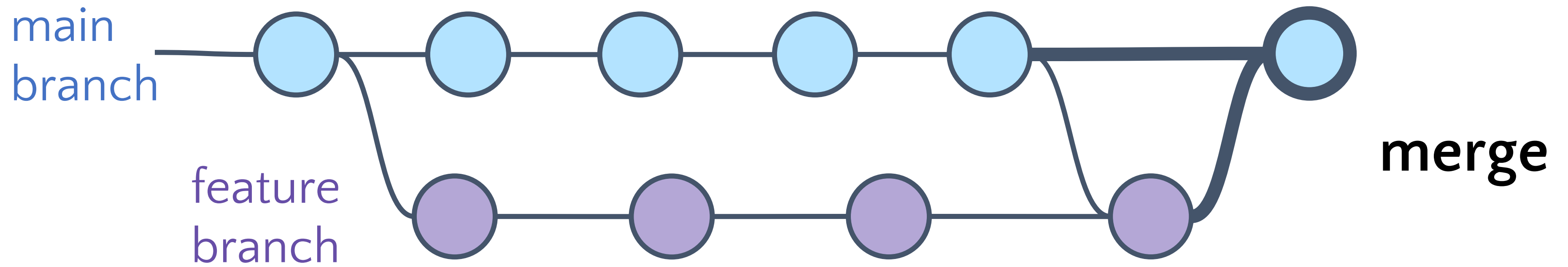
3 ways to resolve a pull request

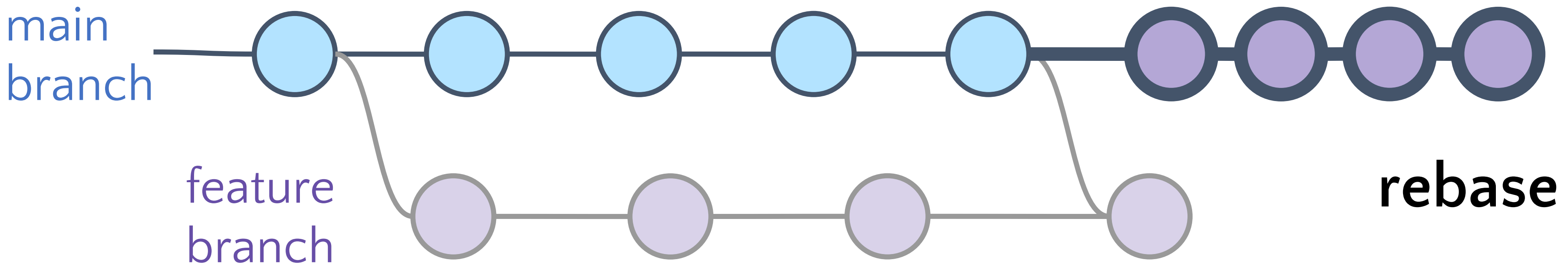


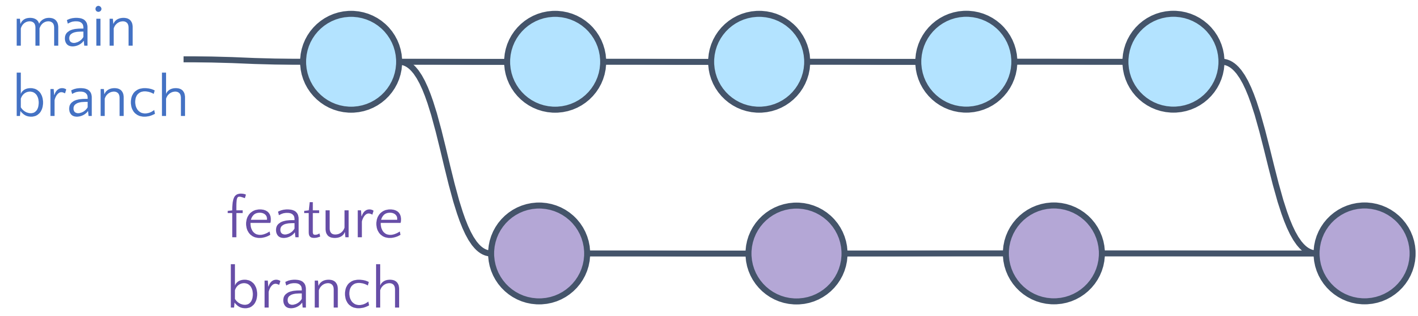




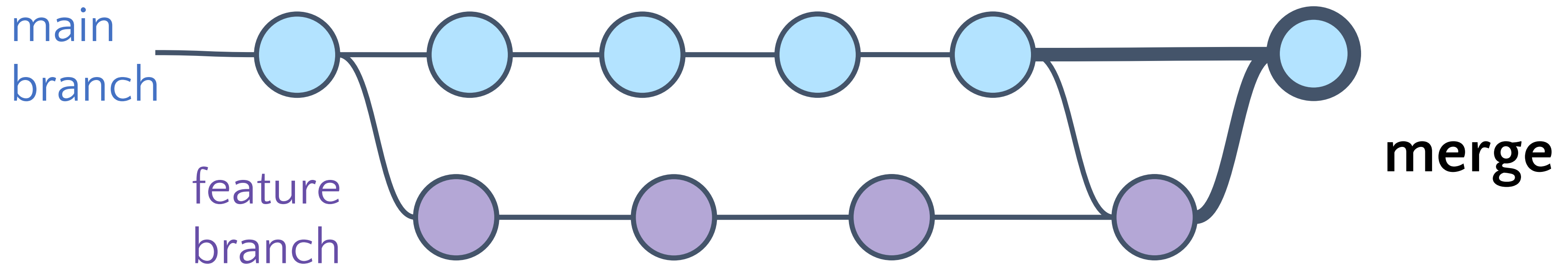
3 ways to resolve a pull request

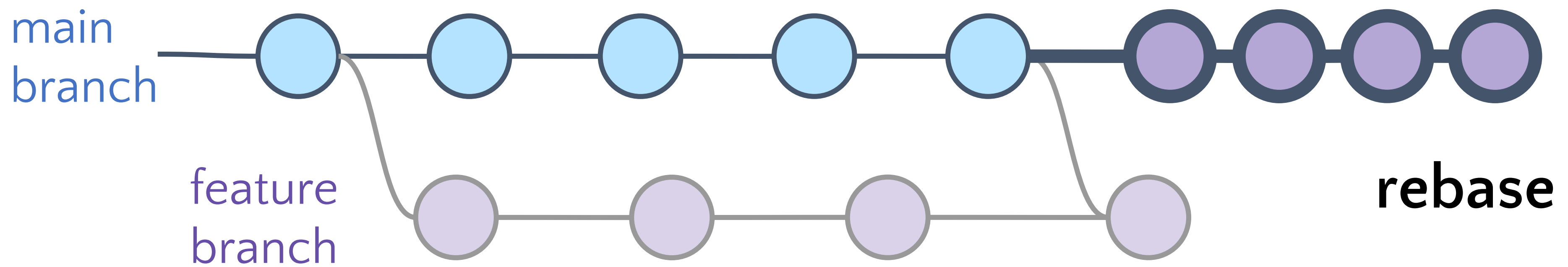


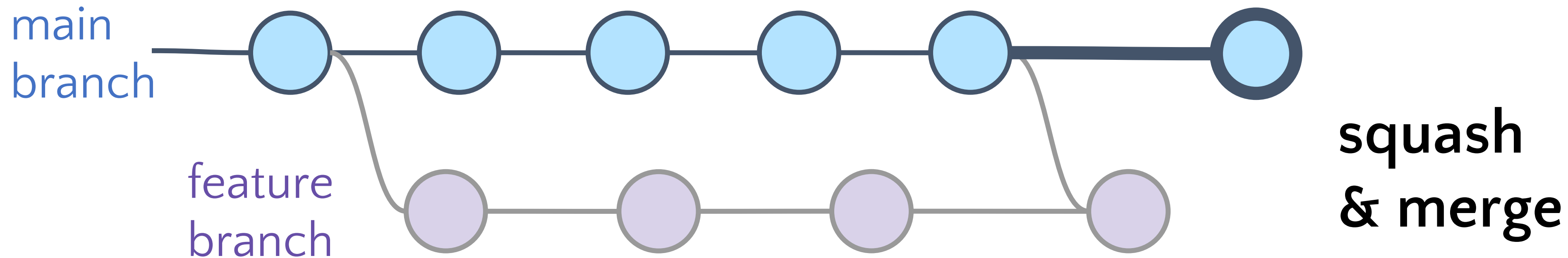




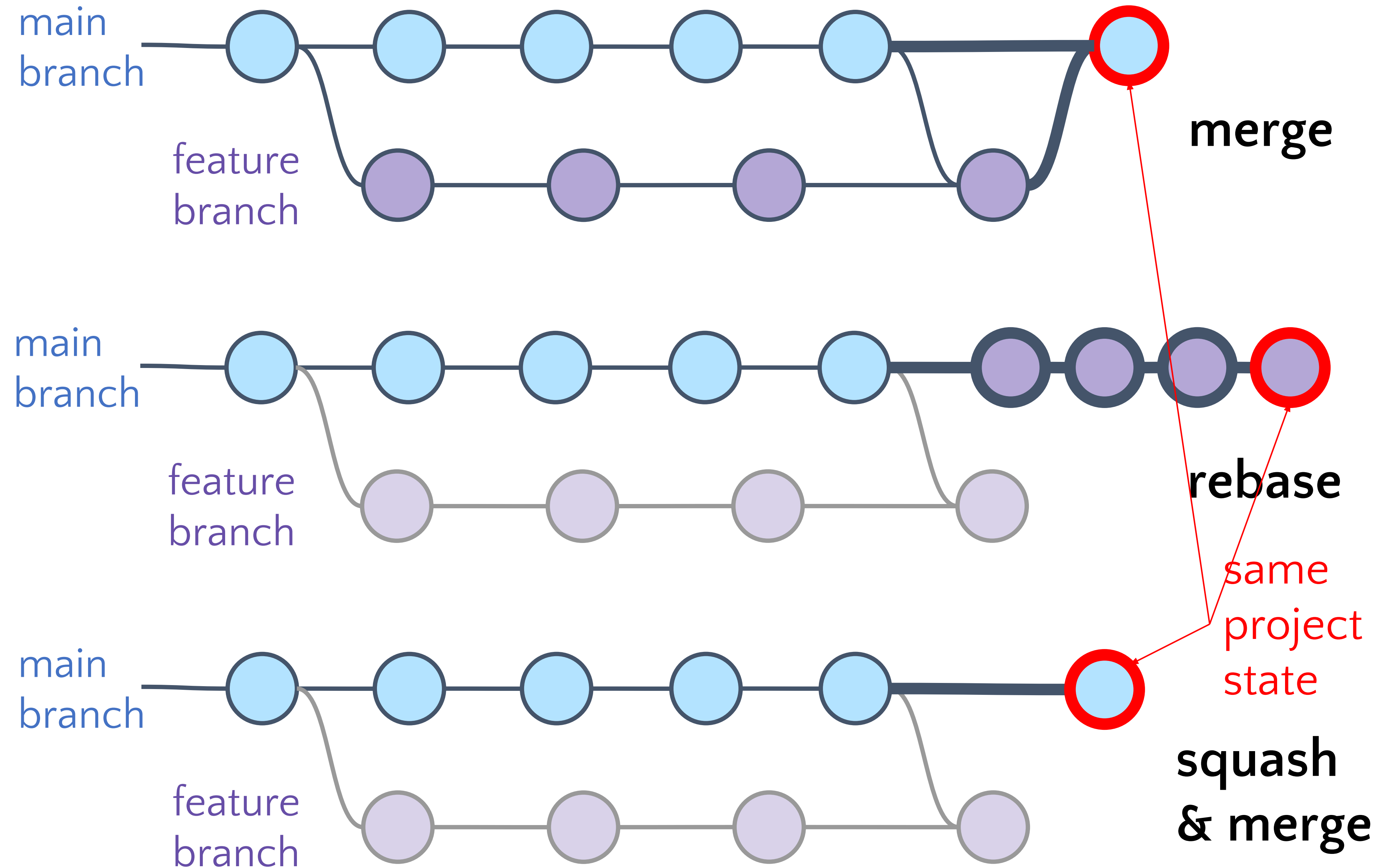
3 ways to resolve a pull request



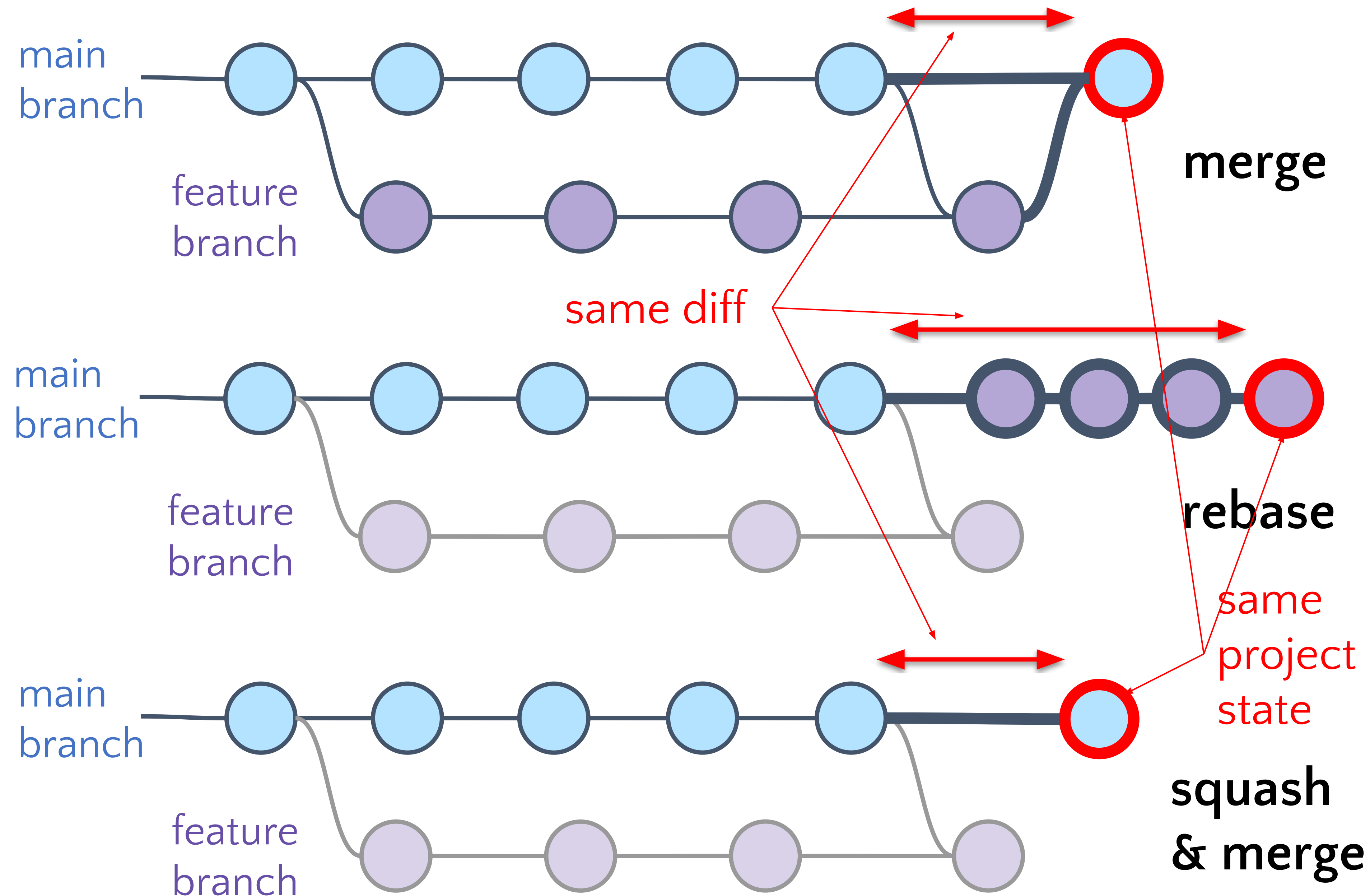




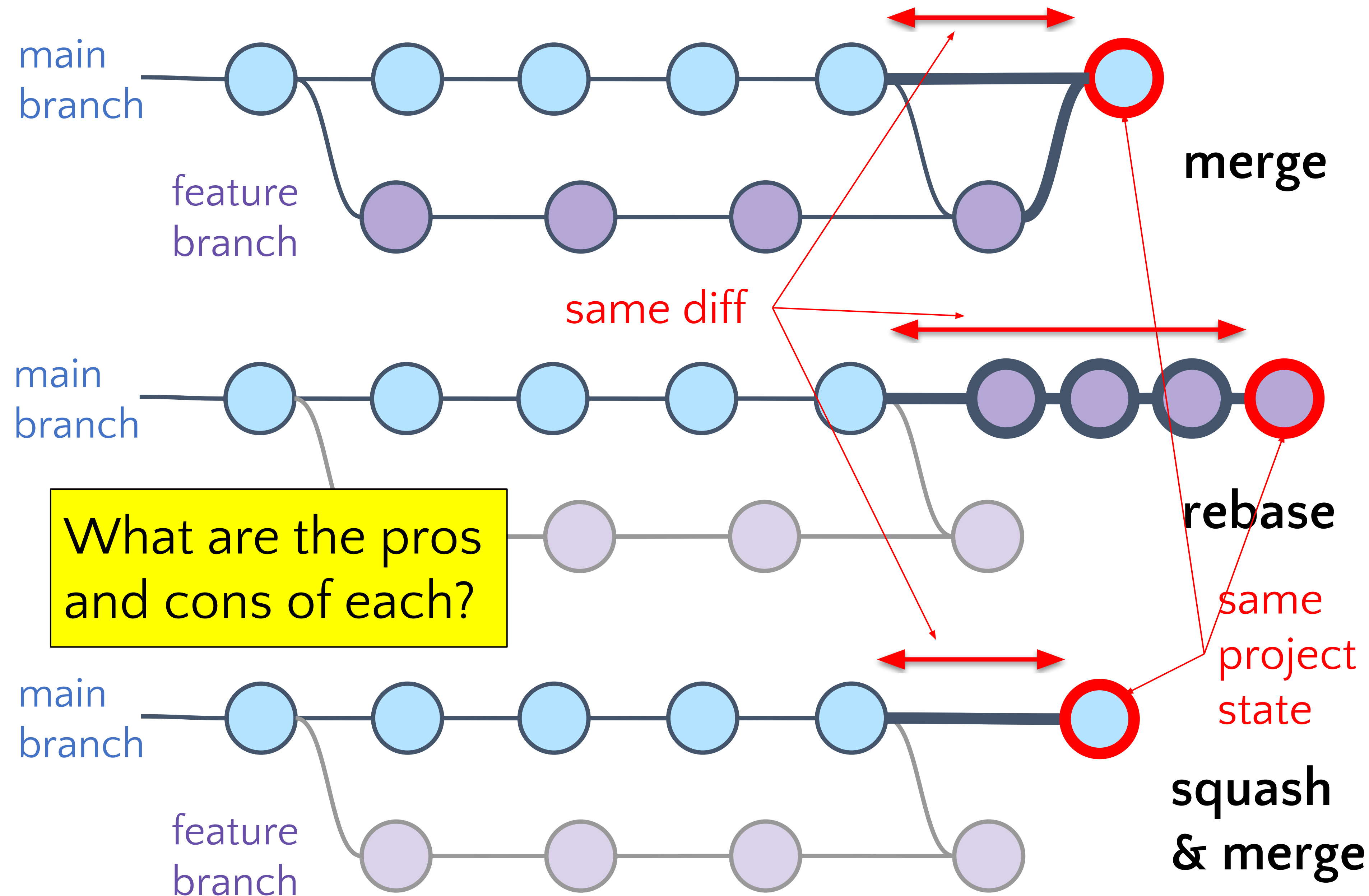
3 ways to resolve a pull request



3 ways to resolve a pull request



3 ways to resolve a pull request



3 ways to resolve a pull request

main branch

feature branch

merge

same diff

main branch

rebase

same project state

squash & merge

branch

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

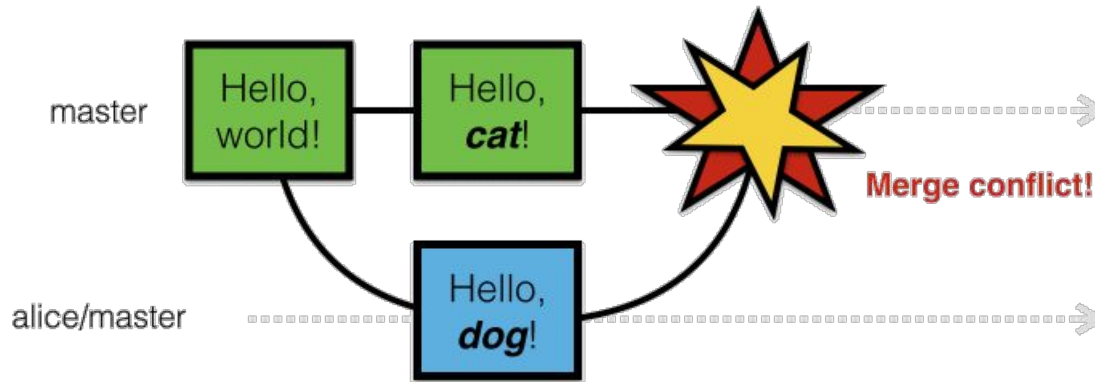
✓ **Squash and merge**
The 14 commits from this branch will be combined into one commit in the base branch.

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

Merge conflicts

Conflicts

- When you run git merge, git attempts to retain all the changes from each branch
- A **conflict** arises when two users **change the same line** of a file

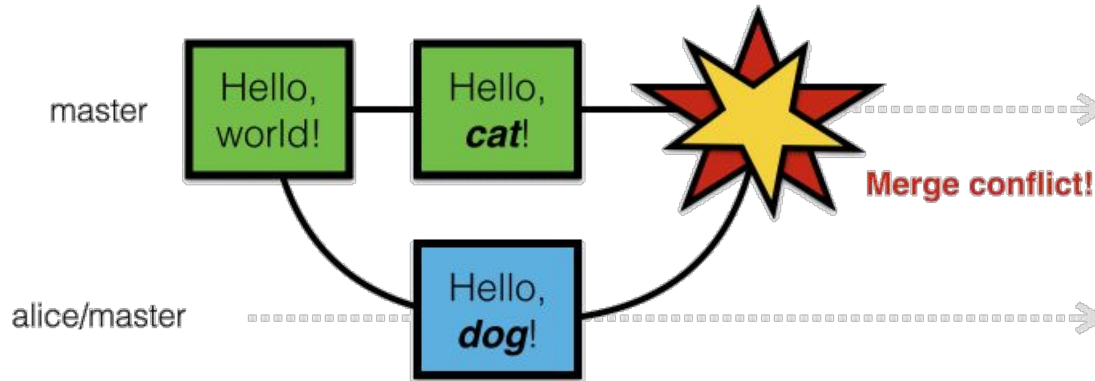


- The person doing the merge needs to resolve the conflict by manual inspection

git's merge tools
can make mistakes

Conflicts

- When you run git merge, git attempts to retain all the changes from each branch
- A **conflict** arises when two users **change the same line** of a file

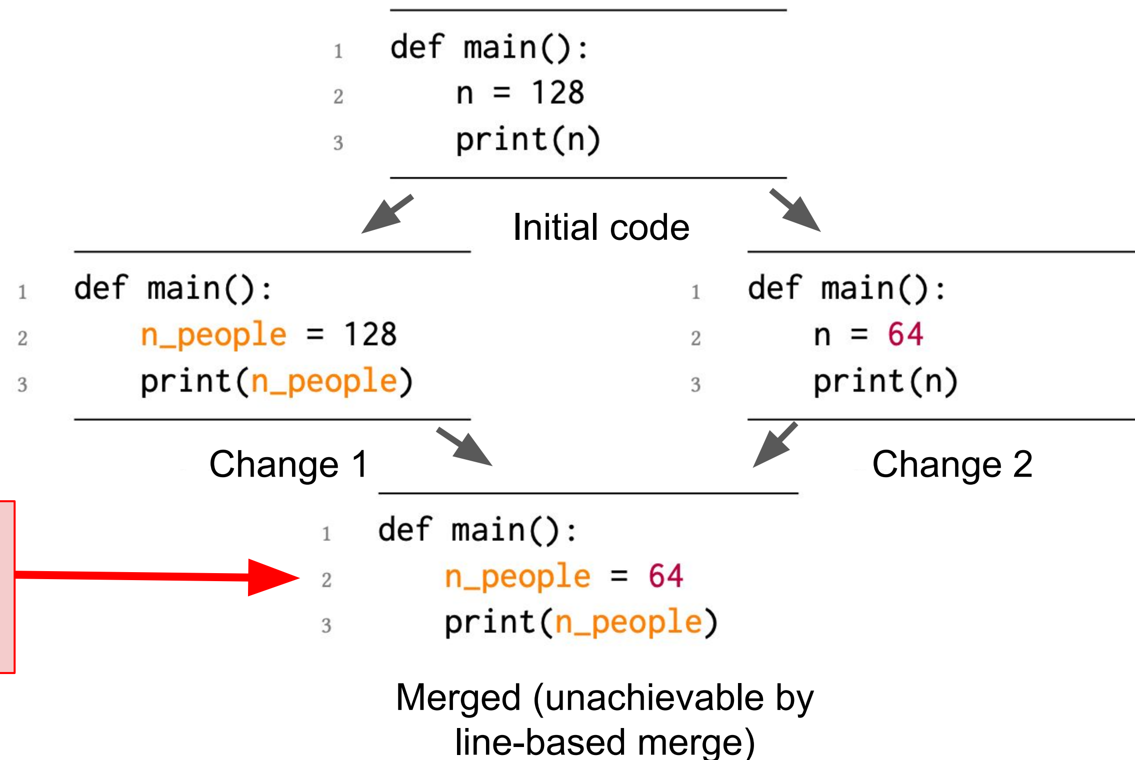


- The person doing the merge needs to resolve the conflict by manual inspection

Merge Algorithm: May Fail to Make a Merge

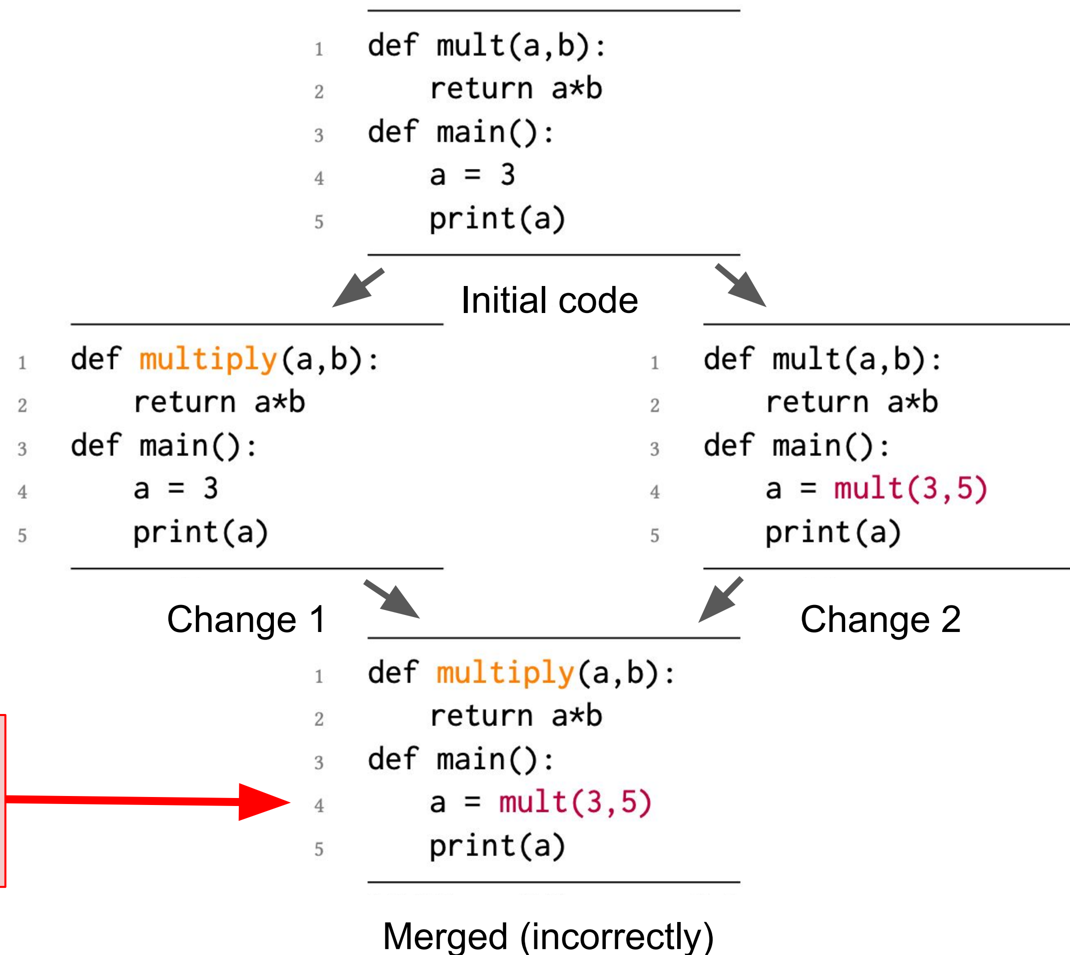
- Line-by-line merge yields a conflict
- Inspection reveals they can be merged

Still works despite 2 changes



Merge Algorithm: Falsely Successful Merge

- Line-by-line merge yields no conflicts (“clean merge”)
- Resulting code is incorrect

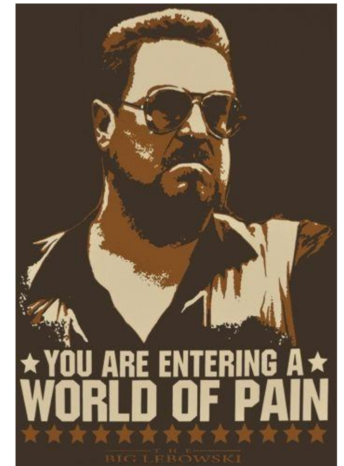


Function name changed

Rebasing
(= rewriting the commit history)

Don’t.

Any questions?



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 (or any long-lived branch):
  1. Every commit should address one concept (see next slide)
  2. Every concept should be in one commit
  3. Tests should always pass
- On feature/bugfix branches:
  1. Don’t worry about the commit history
  2. From branch back into main: squash and merge

Make single-concern branches and commits

They are easier to understand, review, merge, revert.

Ways to achieve single-concern branches and commits:

- Do only one task at a time
  - Commit after each one
- Create a branch for each simultaneous task
  - Easier to share work with teammates
  - Single-concern branch ⇒ Single-concern commit on main
  - Requires a bit of bookkeeping to keep track of them all
- Do multiple tasks in one working copy with multiple branches
  - Commit only specific files, or only specific parts of files (use Git's "staging area" with `git add`; can interactively choose parts of files)

I create a working copy per branch.

Do not commit all files

Use a `.gitignore` file

Don't commit:

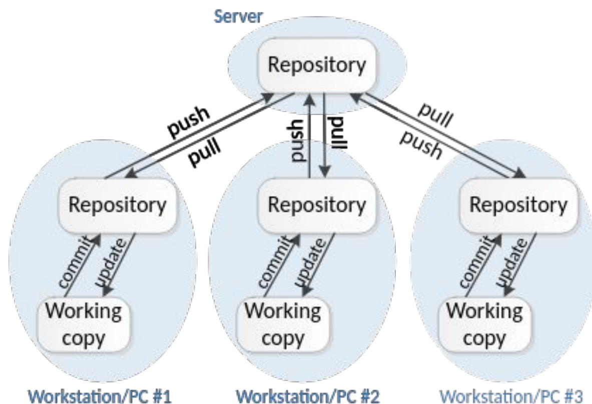
- Binary files
- Log files
- Generated files
- Temporary files

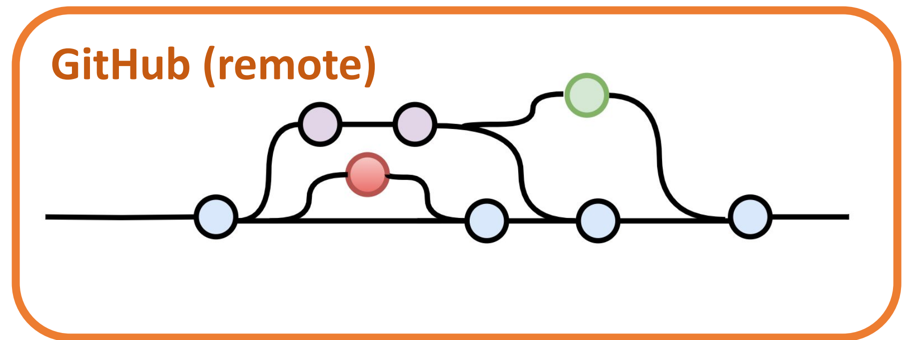
Plan ahead to avoid merge conflicts

- Modularize your work
  - Divide work so that individuals or subteams “own” parts of the code
  - 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

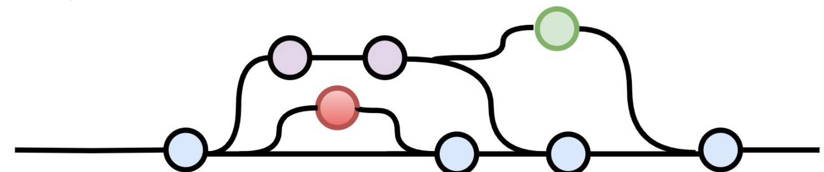
Cloning

- **`git clone`** creates a **local copy** of the repo and a working copy of the files for editing
- Ideal for contributing to a repo alongside other developers
- **`git push`** sends local changes to remote repo





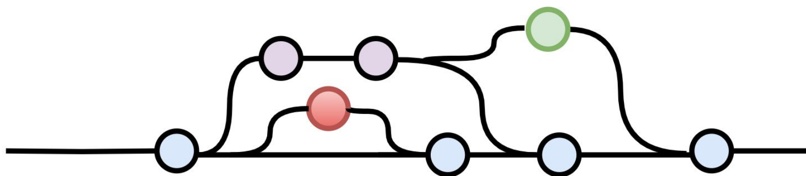
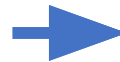
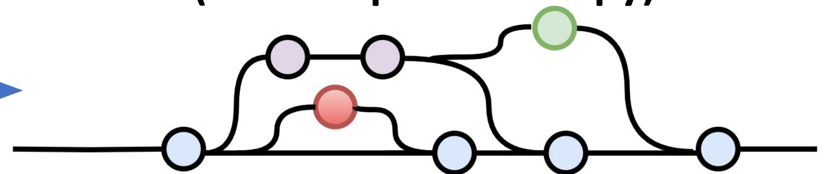
Clone
(copy on local host)



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

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

Typical workflow

`git pull`

`git branch` *`name`*

`git checkout` *`name`*

Repeat:

<edit files, run tests>

`[git add]`

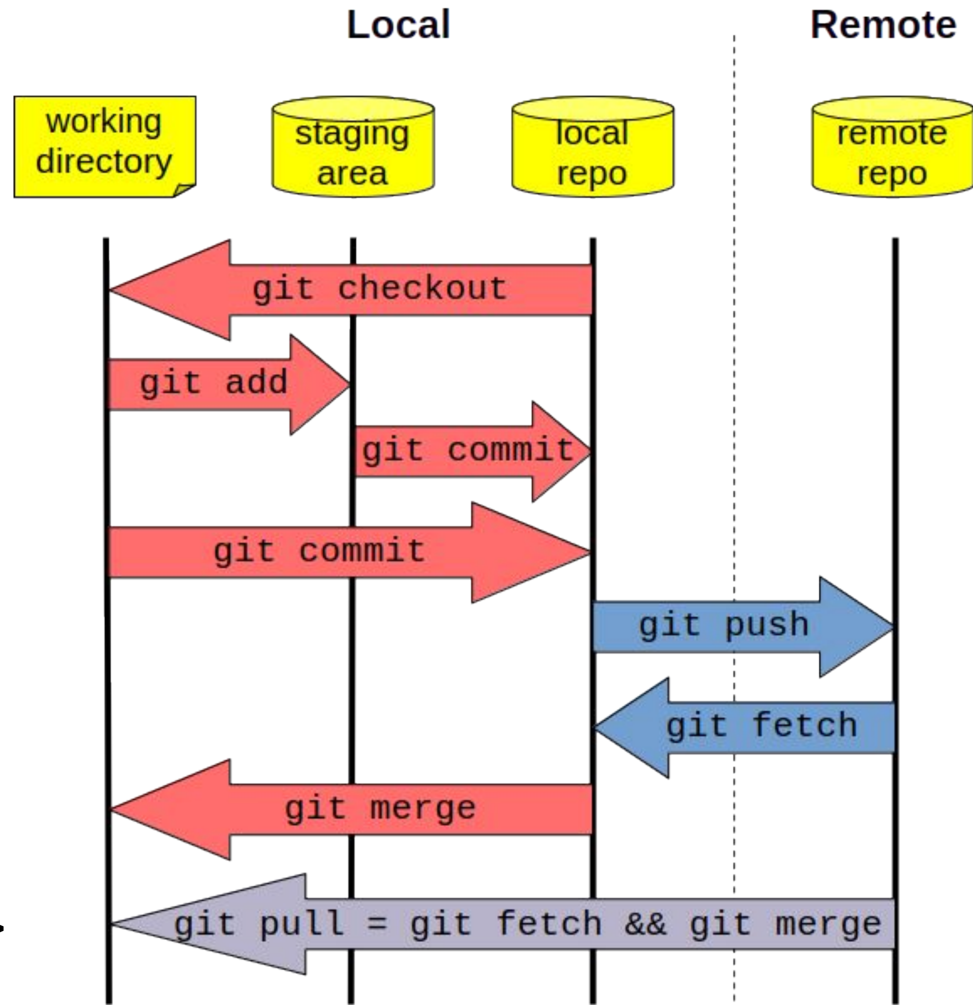
`git commit`

`git pull`

<run tests again>

`git push`

<make a GitHub pull request>



Git’s confusing vocabulary

- **index**: staging area (located .git/index)
- **content**: git tracks **what is in a file, not the file itself**
- **tree**: git's representation of a file system
- **working tree**: tree representing the local working copy
- **staged**: ready to be committed
- **commit**: a snapshot of the working tree (a database entry)
- **ref**: pointer to a commit object
- **branch**: just a (special) ref; semantically: represents a line of dev
- **HEAD**: a ref pointing to the working tree

Learn more!

- Other resources: explanations, tips, best practices
  - Michael Ernst: VC Concepts and Pull Requests
  - Atlassian merge vs rebase
  - Git branching and merging
  - Video tutorial "Git, GitHub, & GitHub Desktop"
  - Learn Git Branching