Lecture 11
Version control

CS 5150, Spring 2026

Administrative Reminders

- Client Meetings: Meet with your client at least once every sprint!
- Sprint timeline:
  - **Sprint 1**: Feb 5 – Feb 26 (Understanding codebase architecture)
  - **Sprint 2**: Feb 27 – Mar 26 (Work towards first simplest prototype)
  - **Sprint 3**: Mar 27 – Apr 23 (Rapid feature development/testing)
  - **Final Sprint**: Apr 24 – May 15 (Acceptance testing/User Testing/Final PPT)

- Midpoint Presentation: Please sign up soon!
- Peer Review: Please submit response by Feb 28 for Sprint 1!
  - We will weigh the project grades based on this score!
  - Points for submission

Version Control Systems

Version control

- **Software engineering**: "Programming integrated over time"
- **Version control**: Code tracked over time

- Helps answer questions:
  - What was the state of the code when this bug was encountered?
  - Which changes created the bug?
  - How many releases/results are affected by the bug?
  - Who can explain this surprising piece of code?
  - What else had to change when this feature was introduced?
  - Which version of the code should I use for my task?

Who uses version control?

- Everyone should use version control
  - Large teams (100+ devs)
  - Small teams (2-10+ developers)
  - Yourself
    - Multiple features/multiple computers
- Example application domains
  - Software development
  - Hardware development
  - Research & experiments
  - Applications (e.g., cloud-based services)
  - Services that manage artifacts (e.g., legal, accounting, business, …)

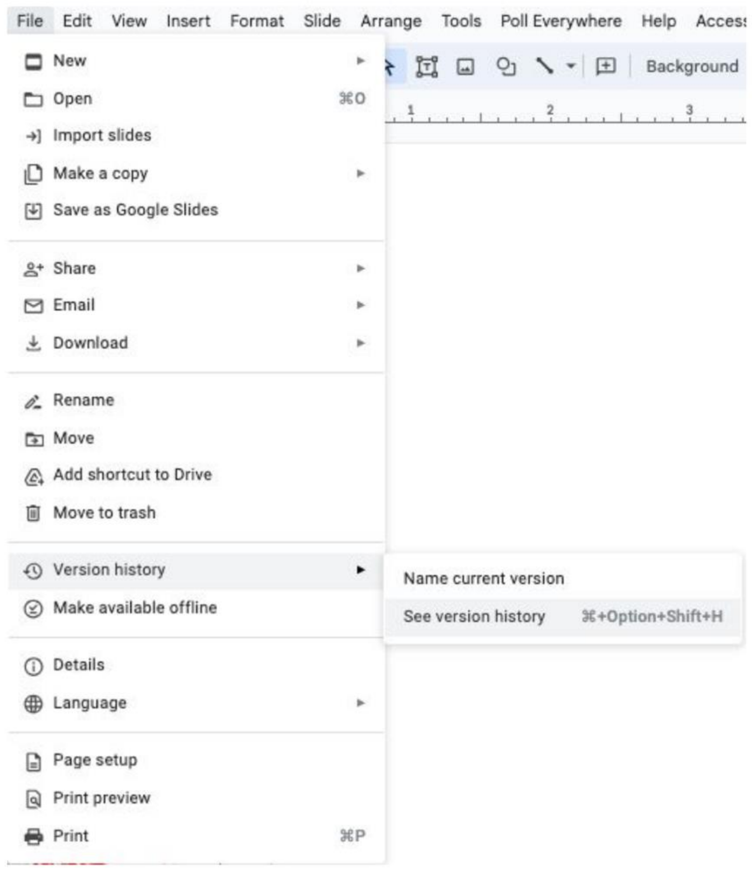
Version control history

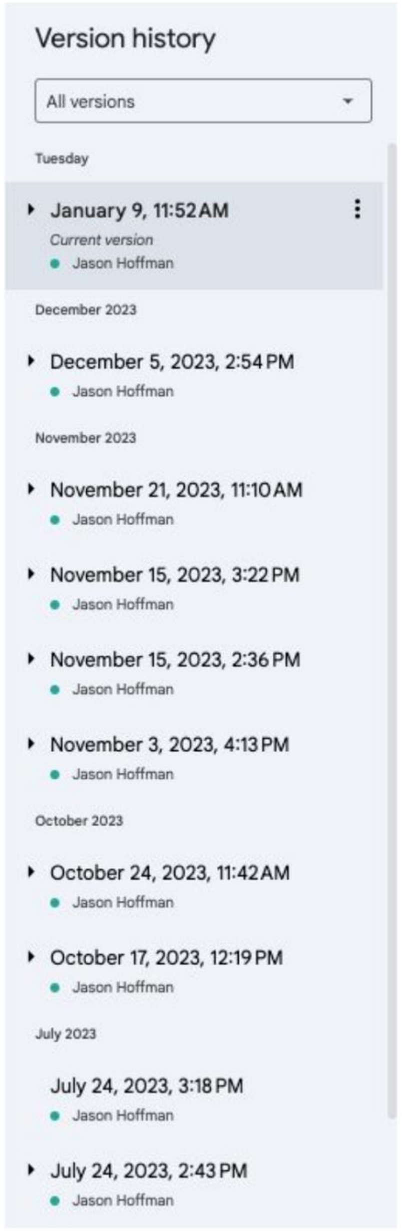
- Today: ubiquitous in software engineering
  - Even though metadata is not part of final product
- Prevents confusion over which version of code to use
  - Avoid confusing filenames
  - Avoid losing changes
- Granularity
  - File locking
  - Atomic commits
  - Line-level merging

Version control for documents

Common App Essay

11:51pm





Version history

All versions

Tuesday

January 9, 11:52 AM
Current version
● Jason Hoffman

December 2023

December 5, 2023, 2:54 PM
● Jason Hoffman

November 2023

November 21, 2023, 11:10 AM
● Jason Hoffman

November 15, 2023, 3:22 PM
● Jason Hoffman

November 15, 2023, 2:36 PM
● Jason Hoffman

November 3, 2023, 4:13 PM
● Jason Hoffman

October 2023

October 24, 2023, 11:42 AM
● Jason Hoffman

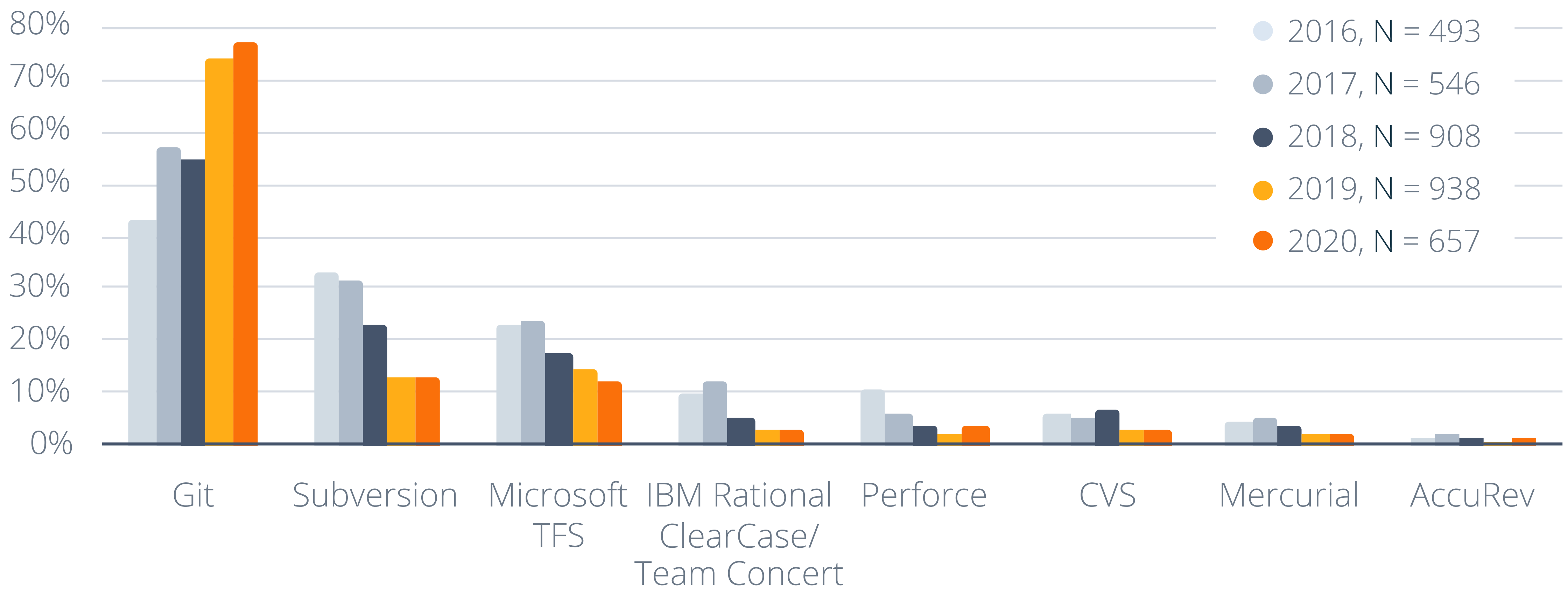
October 17, 2023, 12:19 PM
● Jason Hoffman

July 2023

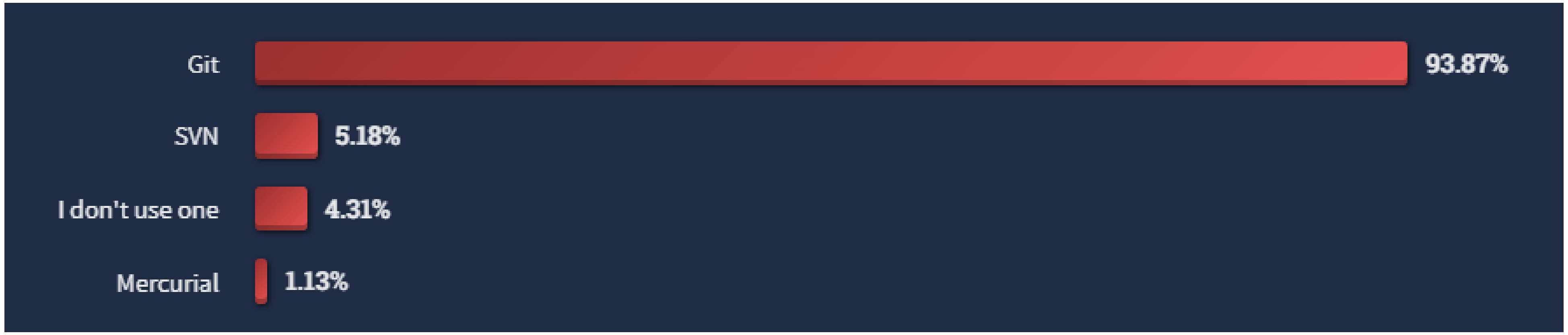
July 24, 2023, 3:18 PM
● Jason Hoffman

July 24, 2023, 2:43 PM
● Jason Hoffman

Which software configuration management system (SCM) do you or your company currently use? *Select all that apply* *fig.16*



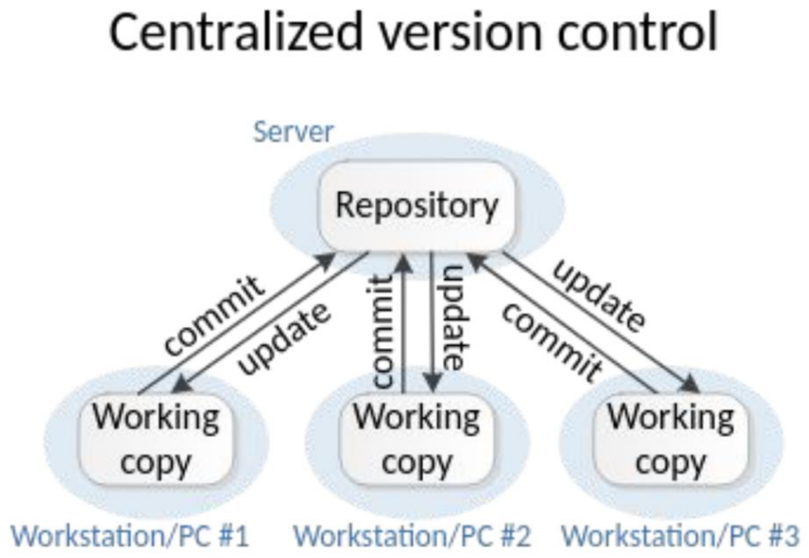
A more recent survey



https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use

Centralized version control (the old way)

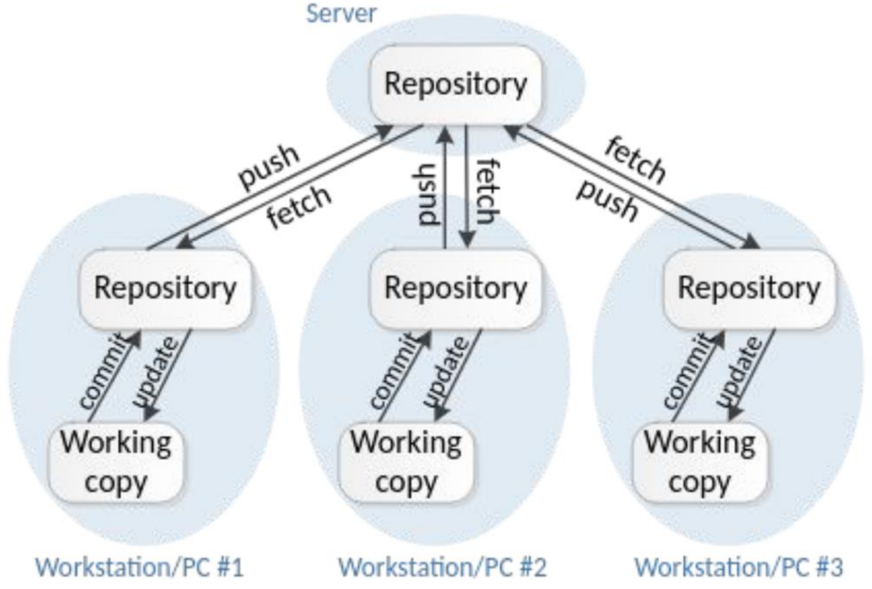
- One central repository
  - Stores a history of project versions
- Each user has a working copy
- A user commits file changes to the repository
- Committed changes are immediately visible to teammates who update
- Examples: SVN (Subversion), CVS
- Problems: Slow!, Single Point of Failure, Branching is expensive, …



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)

Distributed version control in git



Git VCS/SCM

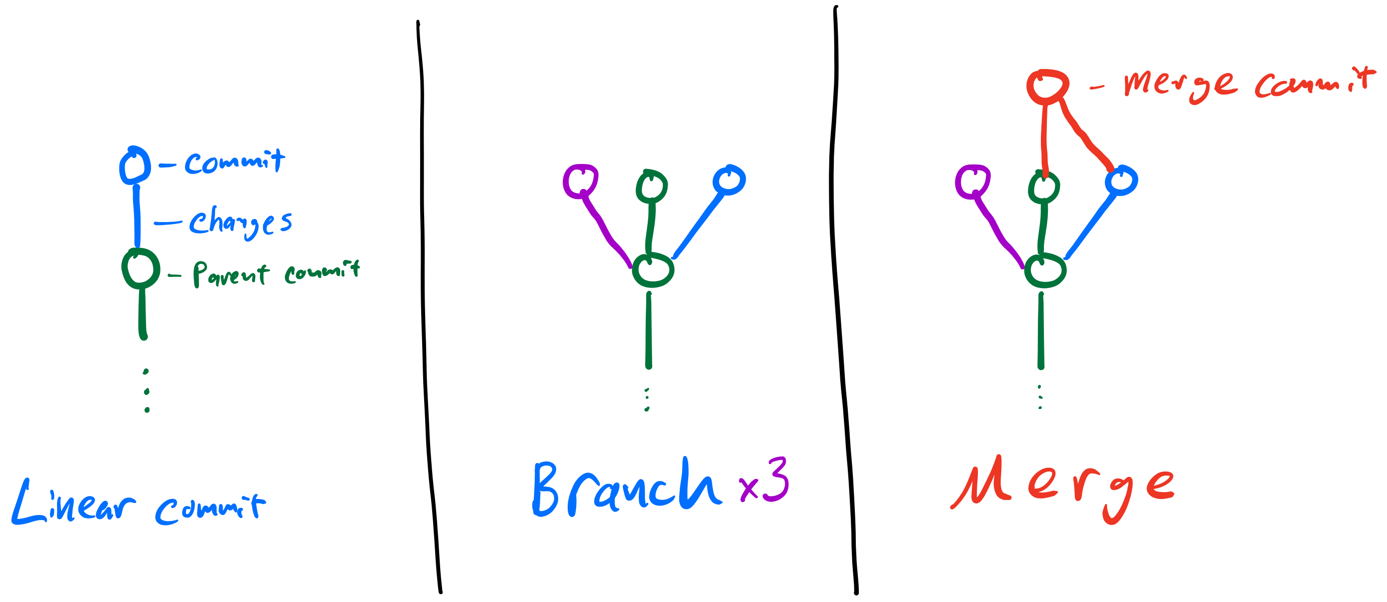
- Created in 2005 by Linus Torvalds
  - Previous system (**bitkeeper**) took 30 seconds to apply a patch!
- Distributed VCS:
  - Geared for Speed, Data Integrity
  - Distributed non-linear workflows running 1000s of parallel branches on different machines
  - Lightweight branching (a branch is only reference to one commit)
- Maintains a “local” copy of entire repo on each machine
  - See “.git” folder
- Other VCS: SVN (centralized), Mercurial (slower), CVS

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Commit basics

- Atomic change affecting any number of files
- **Metadata**: author, timestamp, message
    - Git distinguishes “author” from “committer”
- **Parent commit**: represents state of repository prior to changes
    - **Linear history**: every commit has a single parent and child
    - **Branches**: multiple commits with the same parent
    - **Merge**: commit with two parents
        - Conflict: different changes in the lineages of both parents affect the same lines of code

Commit diagrams



Interesting but important facts:

- Git does not store “files” or “diffs”, but snapshots of trees of blobs identified using hashes!
- Every object is identified by a cryptographic hash (or SHA)
- Immutability, integrity – history is tamper-proof!

Typical Git Workflow

`git pull`

`git branch` *name*

`git checkout -b` *name*

`git switch` *name*

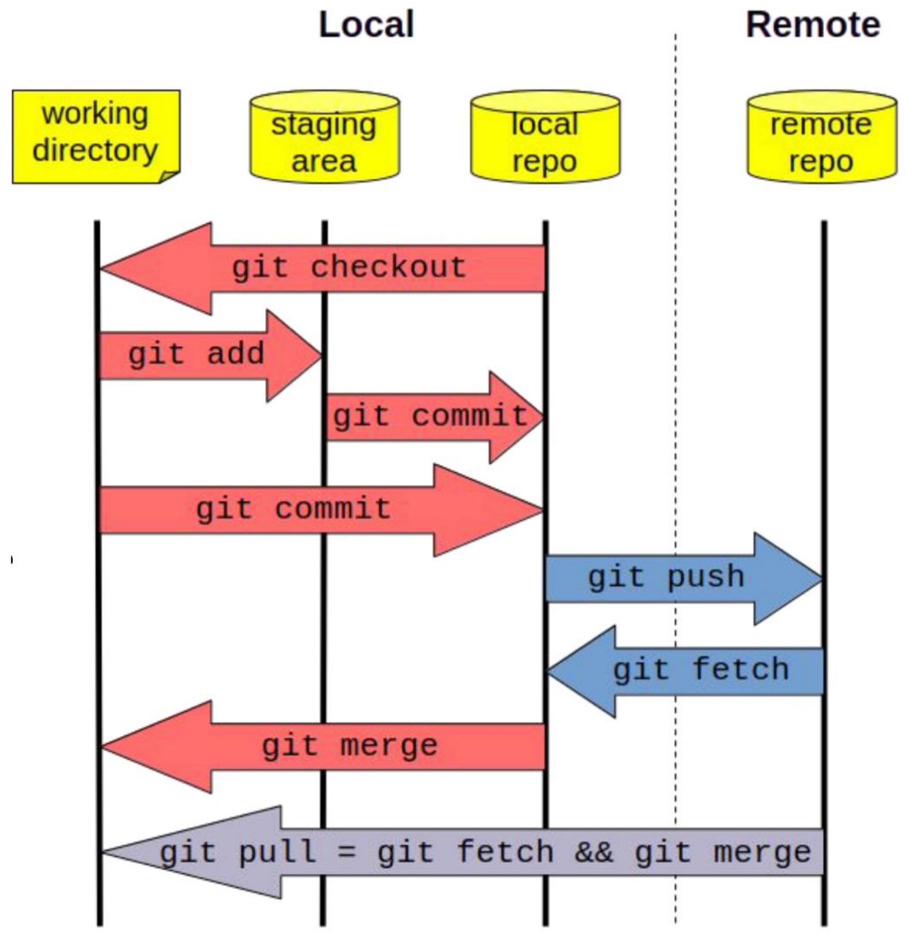
Repeat:

- <edit files, run tests>
- `[git add]`
- `git commit` *filename*
- `git commit`
- `git pull`

<run tests again>

`git push`

<make a GitHub pull request>



Commit message style

PollEv.com/cs5150sp26

- Concise and specific subject
- Separate body from subject
- Link to issues
- Markdown formatting (recommended)

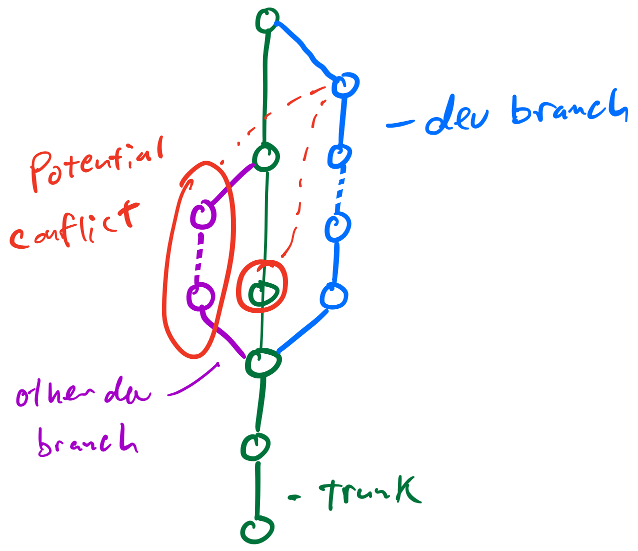
Example: bug fix

1. Create issue
    - Precise description of behavior (actual vs. expected) and context
    - Detailed diagnosis & proposed fixed (when known)
2. Write, test, and commit code
3. Open code review (PR) with changes
    - Describe non-automated testing
4. Respond to feedback
5. Merge changes

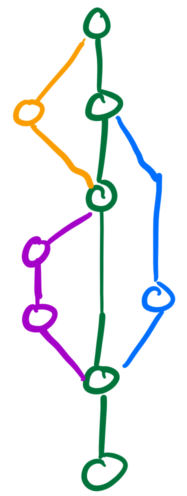
Branch management

- Trunk/master/main
  - Canonical latest version of "ready" code
  - Should be kept in buildable state
- Development branch
  - Long-lived branch for iterating on work in progress
  - Merged with trunk when "finished"
  - Problem: merges are painful; postponing them makes them more painful
  - Shift-left: "if it's painful, do it more often"
- Trunk-based workflow
  - Keep changes small (may queue in issue branch)
  - Merge immediately to trunk
  - Requires continuous testing
- Release branch
  - Tracks version of software released "in the wild" (think hardware products)
  - Provide stability
  - Cherry-pick bugfixes

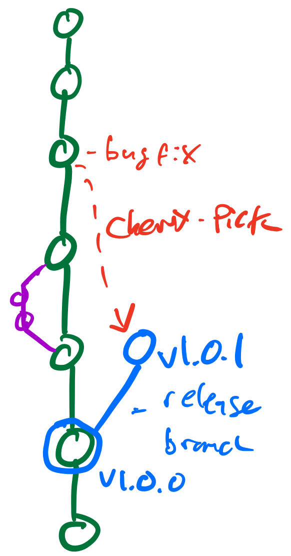
Diagrams



Long-lived development branches



Trunk-based development (short feature branches)



Release branch

Demo

- Commits as DAG
  - git log --oneline --graph --decorate –all
- Blob Trees:
  - git cat-file –p <HASH>
- Whats in the HEAD?

  cat .git/HEAD

  cat .git/refs/heads/main

Merge strategies

- Linear history
  - Total ordering of commits
  - Incorporate new work via rebase
  - Resolve conflicts during rebase

Conceptually simpler

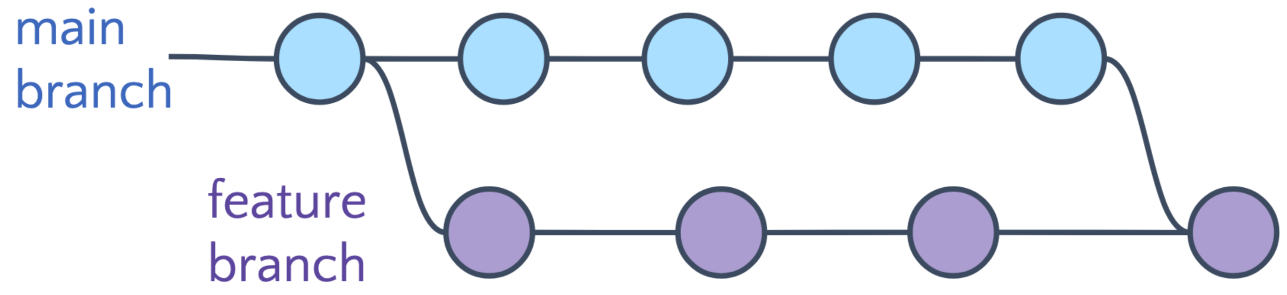
- One-way merging
  - Total ordering of merge commits
  - Incorporate new work via rebase
  - Resolve conflicts during rebase or final merge (latter is not scalable)

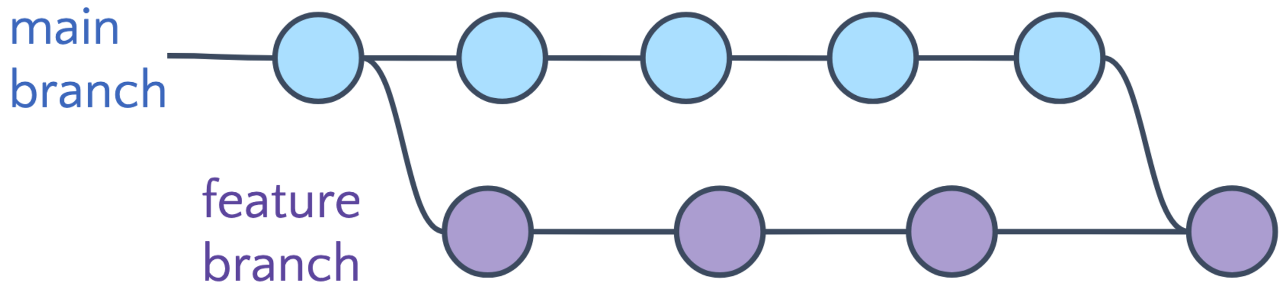
- Two-way merging
  - Incorporate new work by merging trunk into branch
  - Resolve conflicts during back-merges

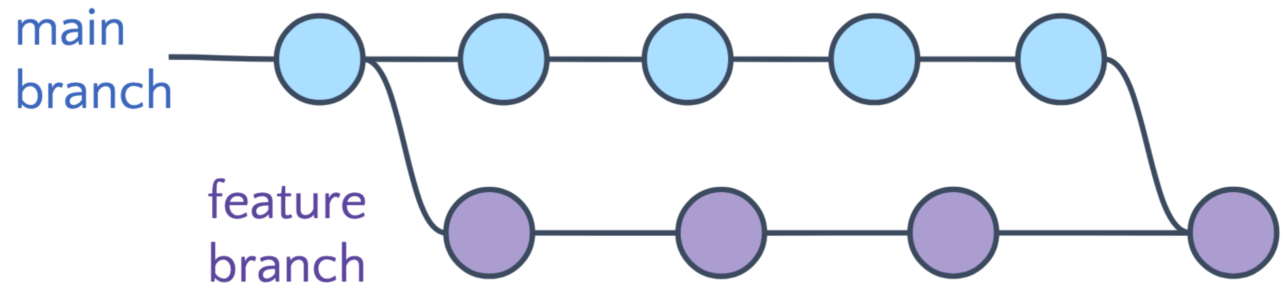
Reflects reality

- Merge vs. rebase
  - Both are opportunities to introduce sneaky bugs
  - Merges are visible in history, rebases are not
  - Rebases replace history – *never rebase a shared branch*

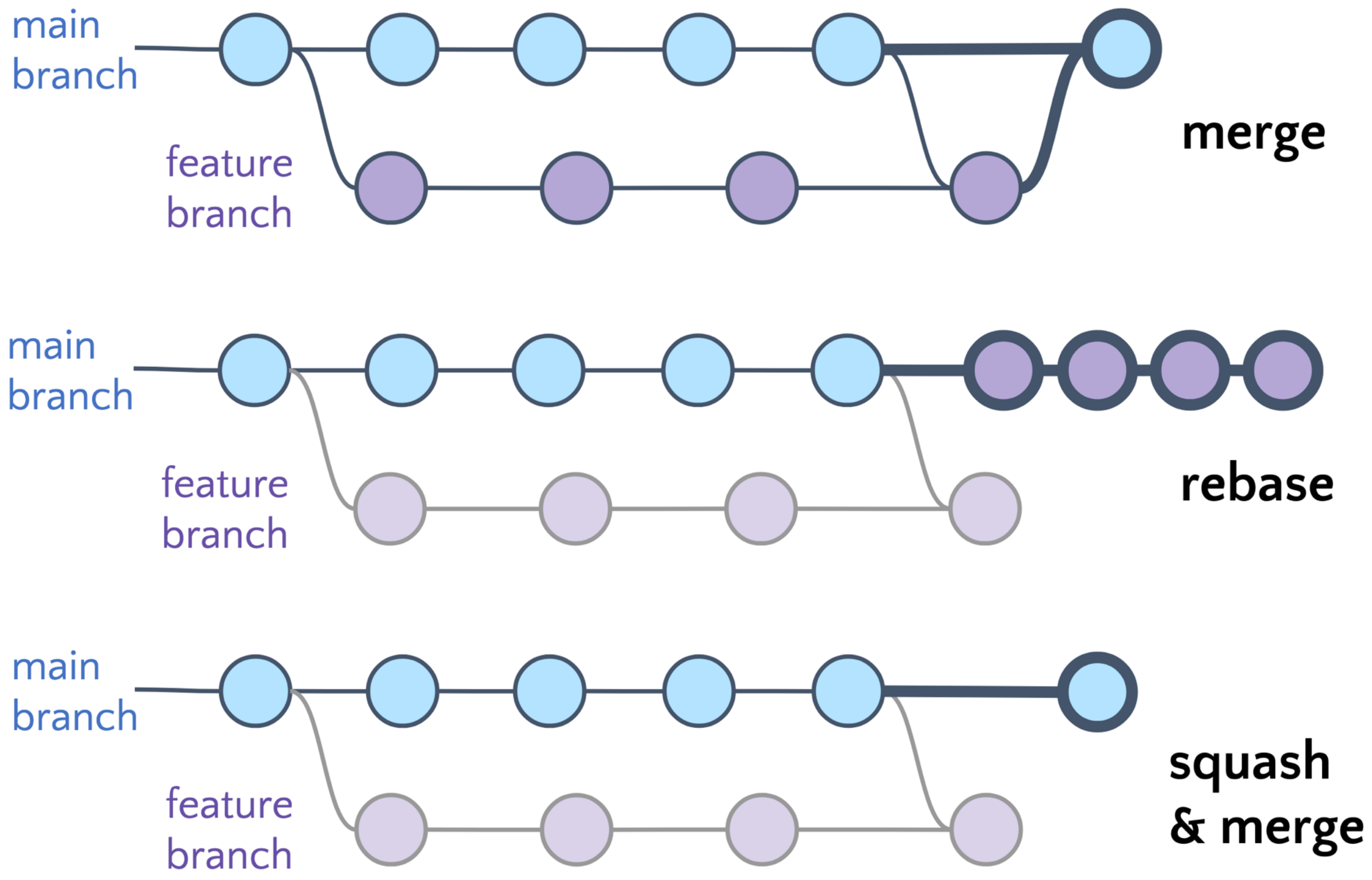
Resolving a pull request



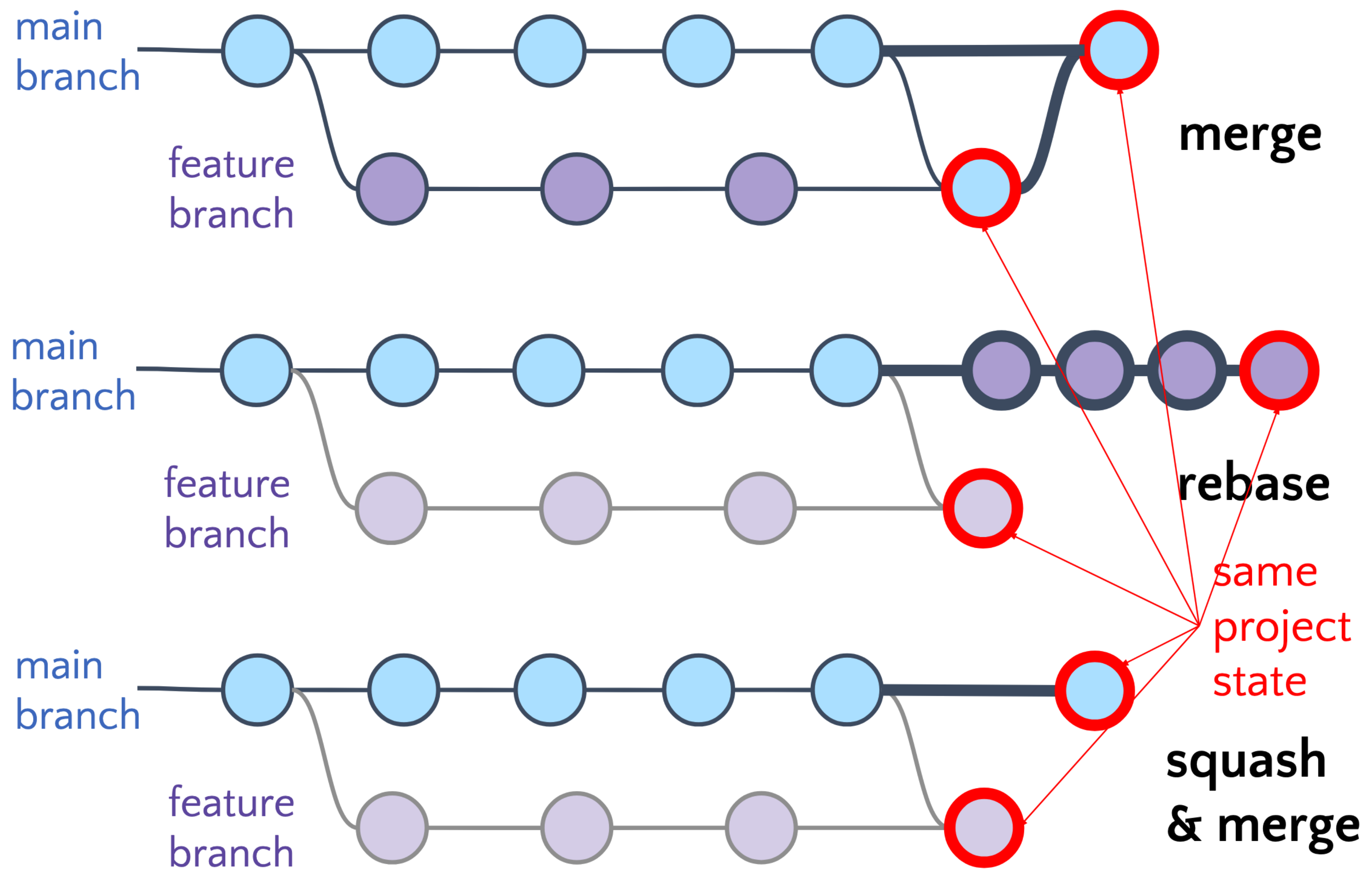




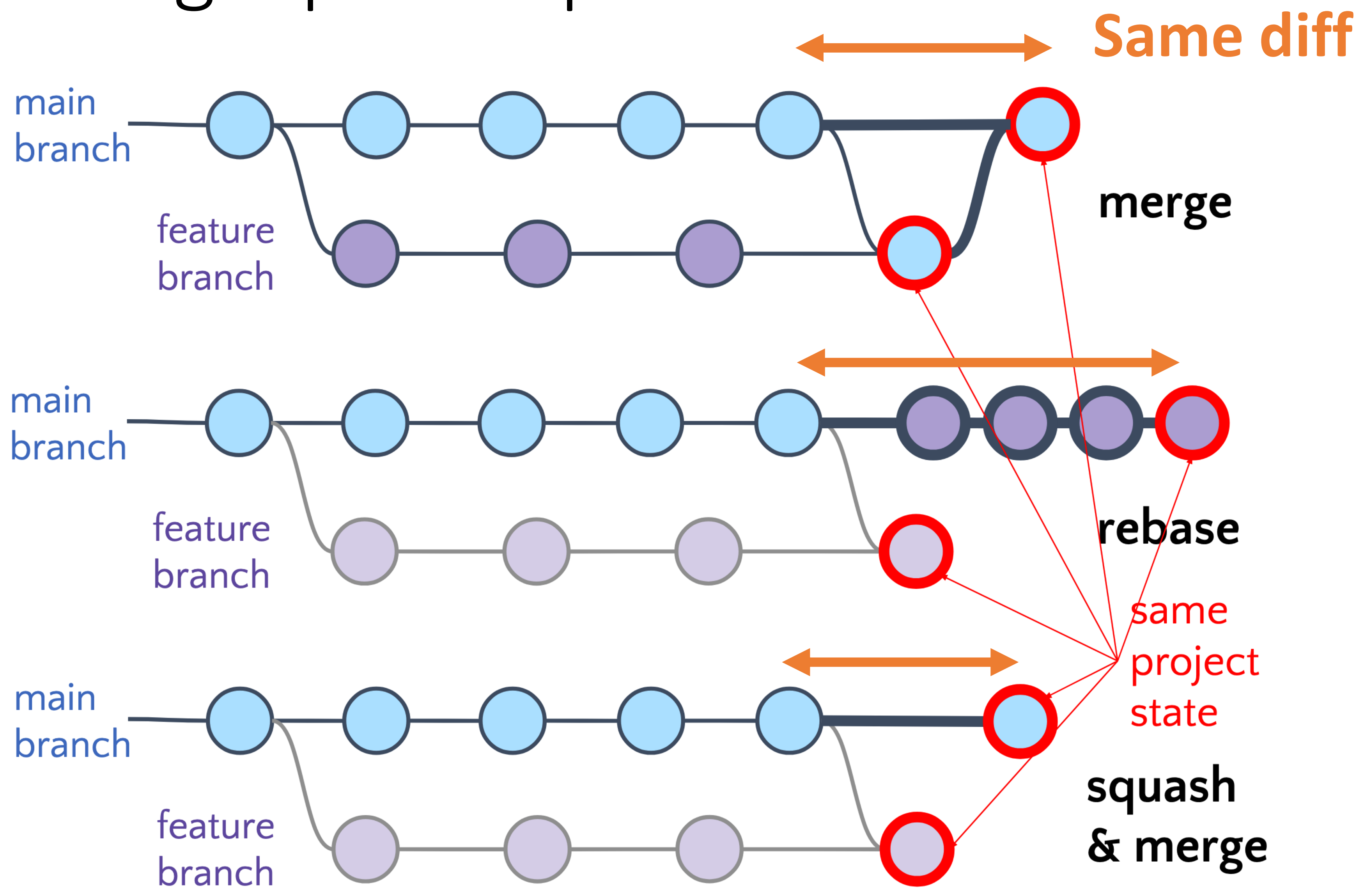
Resolving a pull request



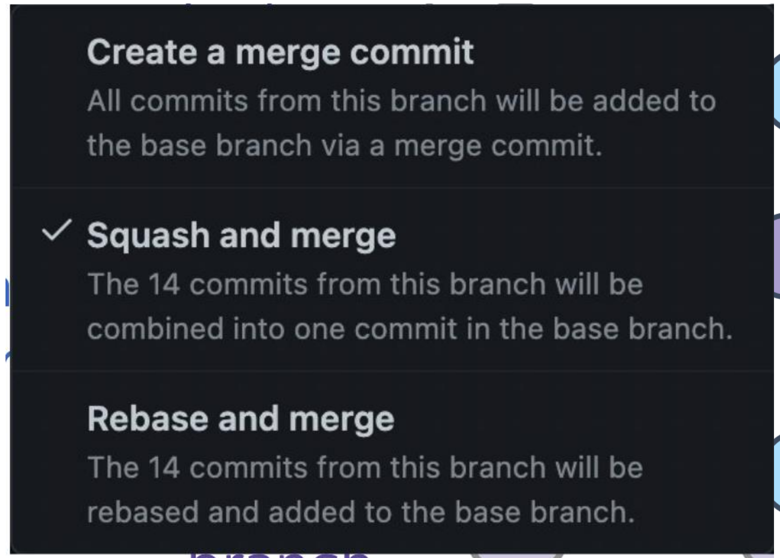
Resolving a pull request



Resolving a pull request



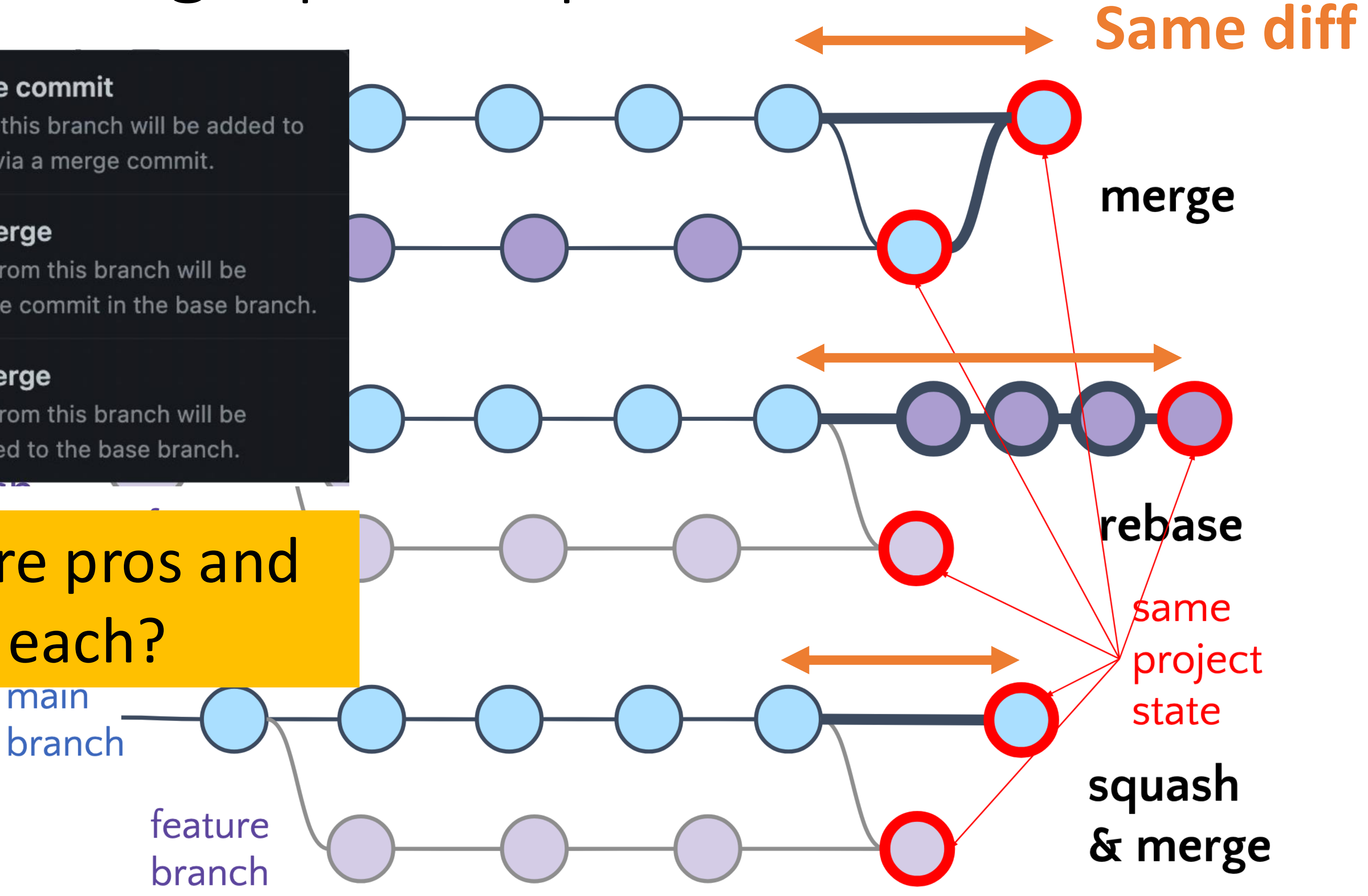
Resolving a pull request



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

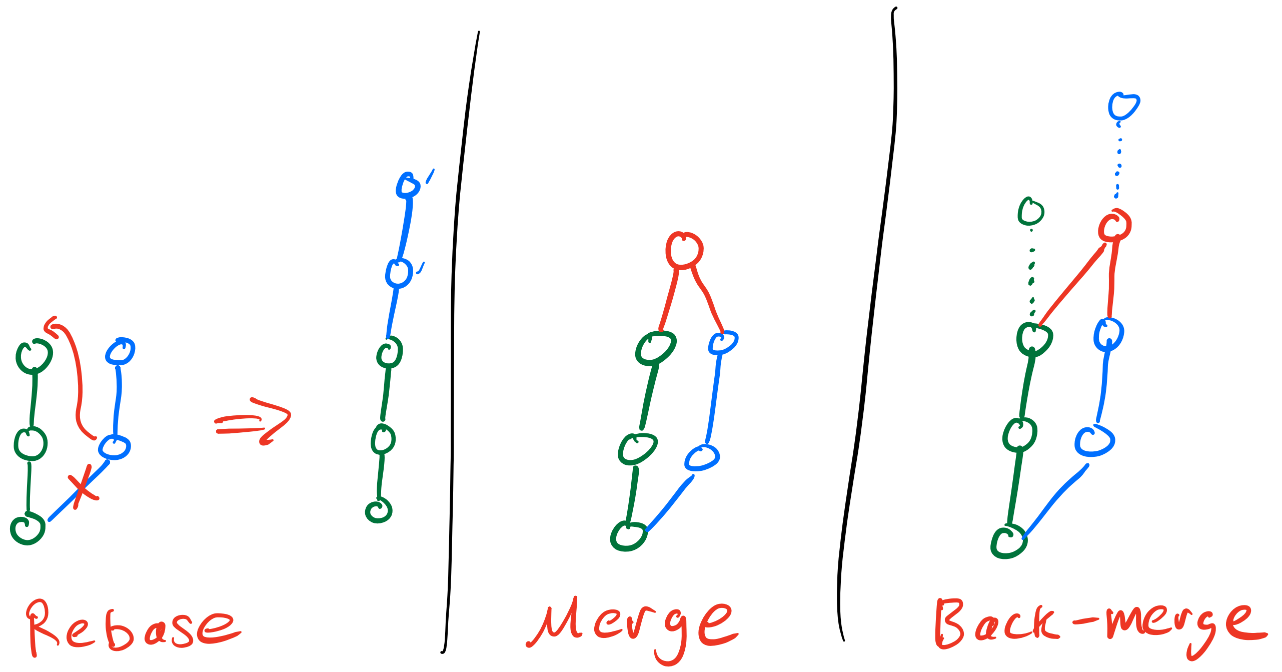
✓ **Squash and merge**
The 14 commits from this branch will be combined into one commit in the base branch.

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



What are pros and cons of each?

Diagrams



Keeping trunk working

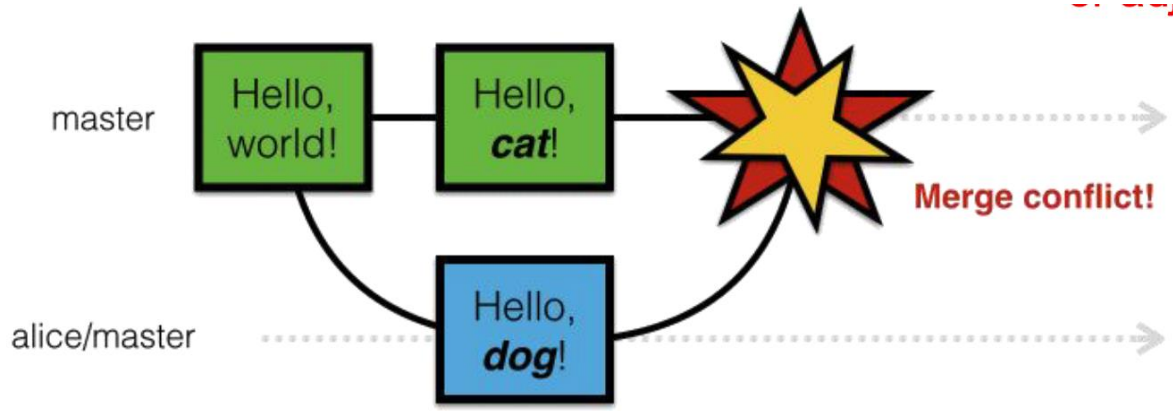
- Testing
  - Ideally automated (CI)
- Peer review

- Every commit should compile and pass tests
  - Facilitates bisection
  - Guides commit boundaries
  - Tests integrating multiple repos are tricky
    - Advantage of monorepo: atomic evolution of integrated system

Conflicts

Git's merge tools can make mistakes!

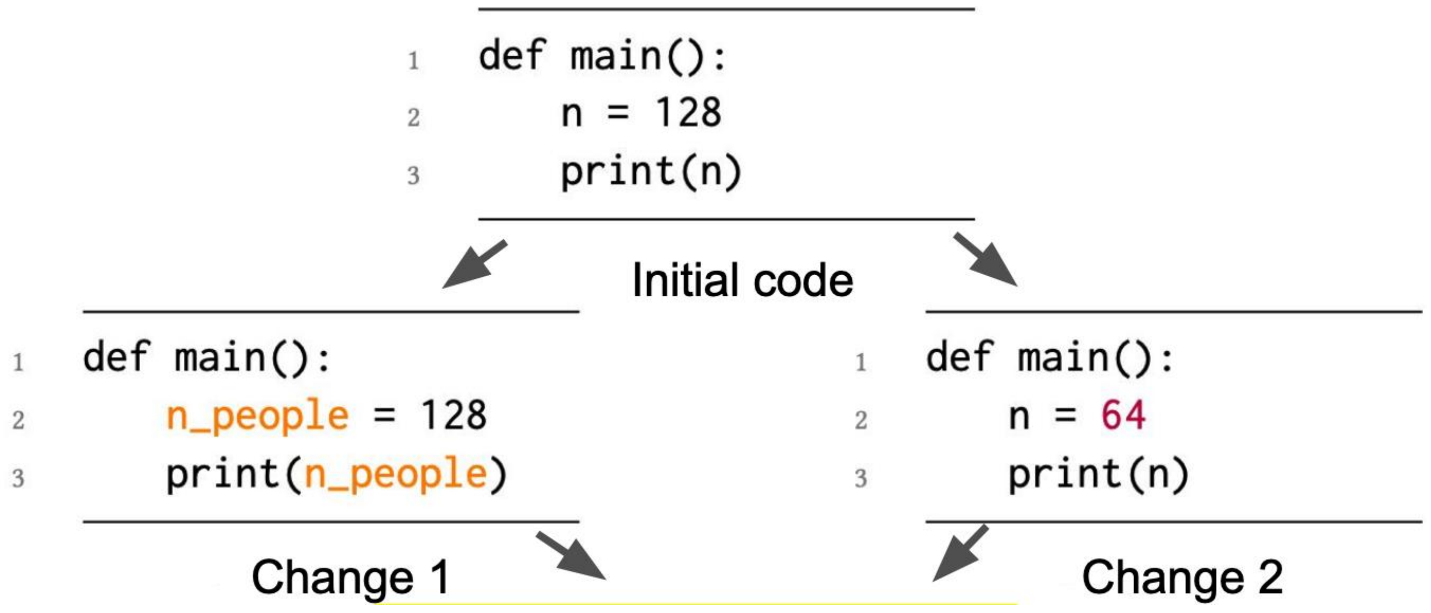
- When you run git merge, git attempts to retain all the changes from each branch
- A "conflict" arises when two users **change the same line (or adjacent lines)** of a file



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

Merge algorithm failure: unable to merge

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



```
def main():
    n = 128
    print(n)
```

```
def main():
    n_people = 128
    print(n_people)
```

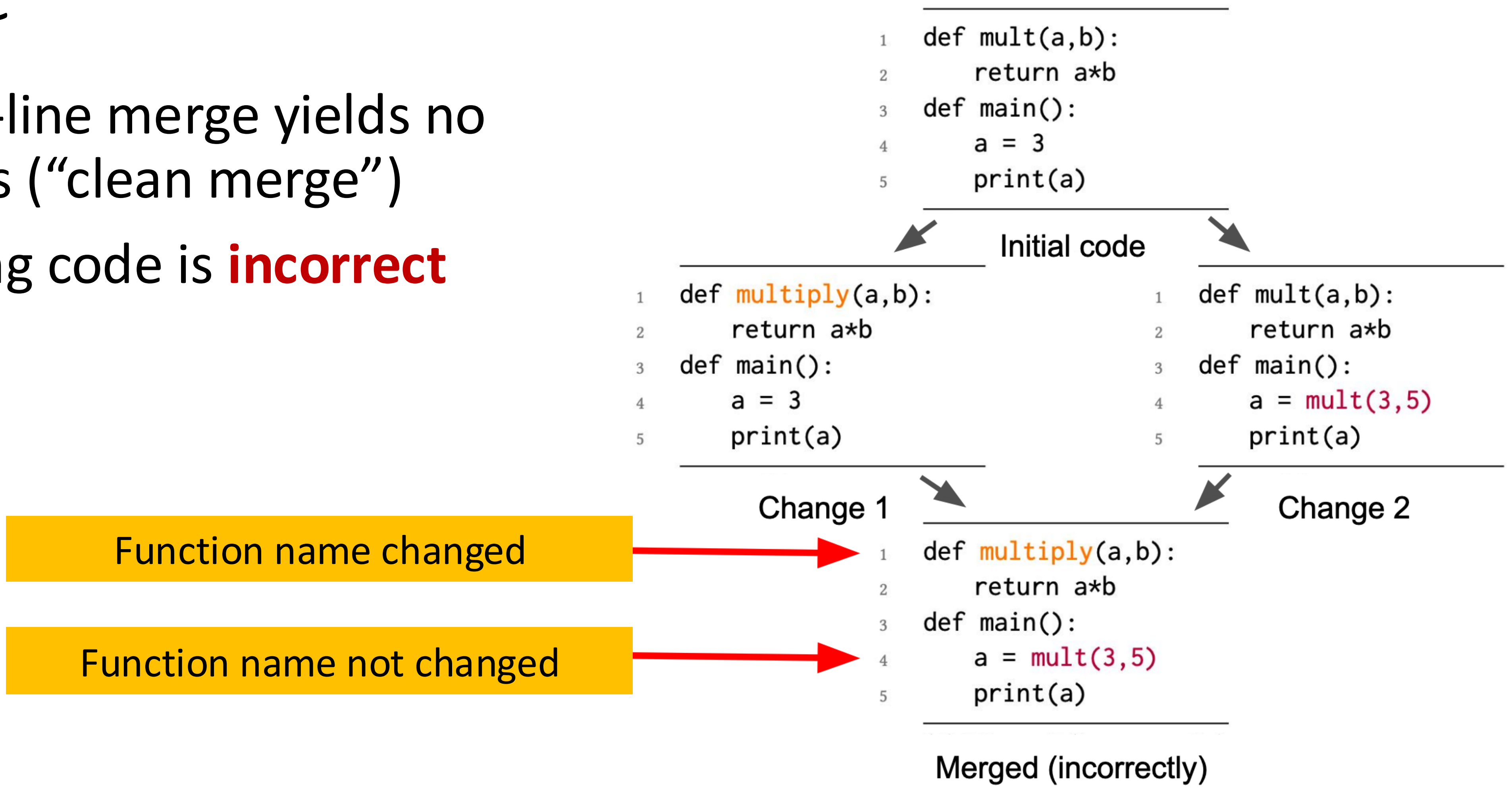
```
def main():
    n = 64
    print(n)
```

Works despite changes on same line

Git's output: "merge conflict"

Merge algorithm failure: clean, incorrect merge

- Line-by-line merge yields no conflicts ("clean merge")
- Resulting code is **incorrect**



Initial code:

```
def mult(a,b):
    return a*b
def main():
    a = 3
    print(a)
```

Change 1:

```
def multiply(a,b):
    return a*b
def main():
    a = 3
    print(a)
```

Change 2:

```
def mult(a,b):
    return a*b
def main():
    a = mult(3,5)
    print(a)
```

Merged (incorrectly):

```
def multiply(a,b):
    return a*b
def main():
    a = mult(3,5)
    print(a)
```

Wrong word substitution!

How to avoid merge conflicts?

- Sync with your teammates often
  - Pull often
  - Push as often as practical
    - Don’t destabilize the main branch (Don’t break the build!)
    - Use continuous integration: Automatic testing on each PR and push, even for branches
  - Avoid long-lived branches (make frequent, small pull requests)
- Commit often
  - Each commit should address one concept; tests should always pass
- Each merged branch should address one concern (Feature/fix)
- Don’t worry about commit history!
- Get changes in main via PR:
  - Squash and merge! What about rebase?