

Git: Miscellaneous Topics

Computer Science and Engineering ■ College of Engineering ■ The Ohio State University

Lecture 4

Basic Workflow: Overview

1. Configure git locally (everyone)
2. Create central repo (1 person)
3. Create local repo (everyone)
4. As you work (everyone):
 - Commit locally
 - Fetch/merge as appropriate
 - Push to share

Step 1: Configure Git Locally

- ❑ Each team member, in their own VM
 - Req'd: Set identity for authoring commits

```
$ git config --global user.name "Brutus Buckeye"
```

```
$ git config --global user.email bb@osu.edu
```
 - Tips
 - ❑ Add to your GitHub account (Settings > Email)
 - ❑ Or use GitHub-generated fake address (Settings > Email > Keep my address private):
`<username>@users.noreply.github.com`
 - ❑ Add your SSH key to your GitHub account
 - Optional: set default initial branch name

```
$ git config --global init.defaultBranch main
```

Step 2: Initialize Central Rep

- ❑ One person, once per project
- ❑ Hosting services (GitHub, BitBucket...) use a web interface for this step
- ❑ Alternative: a location that the group has access to (*e.g.* stdlinux):
 - Create central repository in group's project directory (/project/c3901aa03)
\$ cd /project/c3901aa03
\$ **mkdir** proj1 *# an ordinary directory*
■ Initialize this directory as a *bare* git repository, with group permissions
\$ git **init** --bare --shared proj1

Step 3: Create Local Repository

- Each team member, once, in their VM
 - Create local repository by *cloning* the central repository

```
$ git clone git@github.com:bb/proj1.git
```
 - Copies contents of remote repo, including store, and sets a remote called "origin"

```
proj1$ git remote -v # display info
origin git@github.com:bb/proj1.git (fetch)
origin git@github.com:bb/proj1.git (push)
```
- Different ways to clone
 - HTTPS: Easy to set up, but you need to enter your password every time you fetch/push
 - SSH: Set up by adding your SSH key to the remote host, then easy to fetch/push

Step 4: Local Development

- Each team member repeats:
 - Edit and commit (to local repository) often
`$ git status/add/rm/commit`
 - Pull others' work when can benefit
`$ git fetch origin # bring in changes`
`$ git log/checkout # examine new work`
`$ git merge, commit # merge work`
 - Push to central repository when confident
`$ git push origin main # share`

Demo

- <https://git-school.github.io/visualizing-git/#upstream-changes>
- Try:
 - `git commit`
 - `git fetch origin # see origin/feature`
 - `git merge origin/feature # see feature`
 - `git push origin feature # see remote`

Alternative: In-class Activity

- ❑ Navigate to class org on GH and find the repo called *first-commits*
- ❑ Clone the repo to your VM
- ❑ Do some development!
 - Edit
 - Inspect the store's DAG

```
git log --graph --oneline --all
```
 - Commit, fetch, merge, push...

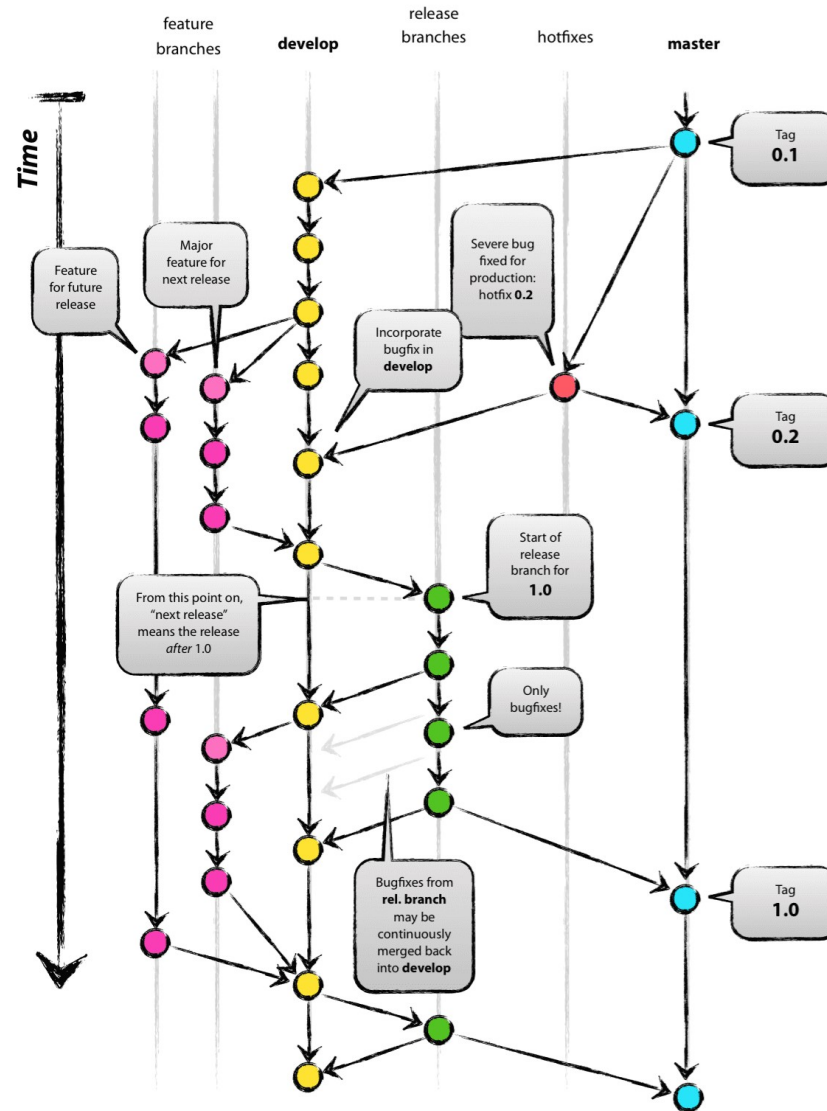
Professional Git

- ❑ Commit/branch conventions
- ❑ Deciding what goes in, and what stays out of the store
 - Share all the things that should be shared
 - Only share things that should be shared
- ❑ Normalizing contents of the store
 - Windows vs linux line endings

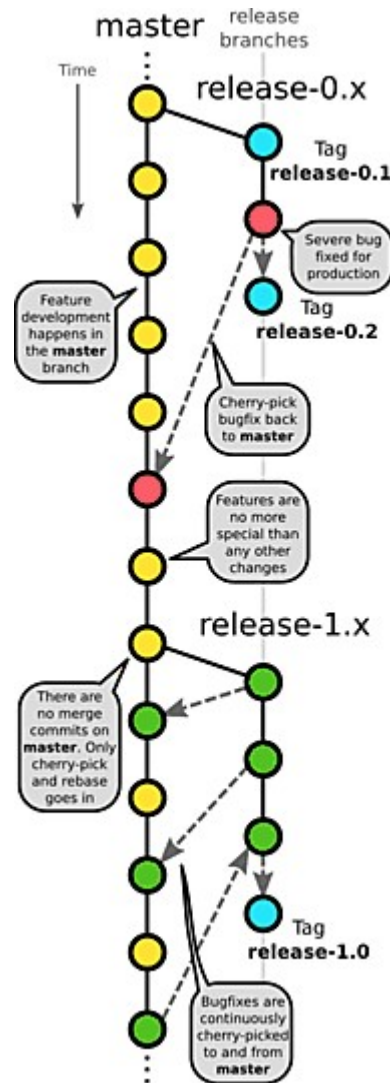
Commit/Branch Conventions

- Team strategy for managing the structure of the DAG (ie the store)
- Examples:
 - “Main is always deployable”
 - All work is done on other branches, merged with main only when result is executable
 - “Feature branches”, “developer branches”
 - Each feature developed on its own branch vs. each developer works on their own branch
 - “Favor rebase over merge”
 - Always append to latest origin/branch

Example: Branch-Based Dev



Example: Trunk-Based Dev



What Goes Into Central Repo?

- ❑ Avoid developer-specific environment settings
 - Hard-coded file/directory paths from local machine
 - Passwords
 - Better: Use *variables* instead
 - OK to include a sample config (each developer customizes but keeps their version out of store)
- ❑ Avoid living binaries (docx, pdf)
 - Meaningless diffs
- ❑ Avoid generated files
 - compiled files, the build
- ❑ Avoid IDE-specific files (.settings)
 - Some generic ones are OK so it is easier to get started by cloning, especially if the team uses the same IDE
- ❑ Agree on code formatting
 - Auto-format is good, but only if everyone uses the same format settings!
 - Spaces vs tabs, brace position, etc

Ignoring Files from Working Tree

- Use a .gitignore file in root of project
 - Committed as part of the project
 - Consistent policy for everyone on team
- Examples: <https://github.com/github/gitignore>

```
# github:gitignore/Java.gitignore
```

```
# Compiled class file
```

```
*.class
```

```
# Log file
```

```
*.log
```

```
# Package Files #
```

```
*.jar
```

```
*.war
```

```
*.ear
```

```
*.zip
```

```
*.tar.gz
```

```
*.rar
```

Problem: End-of-line Confusion

- Differences between OS's in how a new line is encoded in a text file
 - Windows: 2 bytes, CR + LF ("`\r\n`", 0x0D 0x0A)
 - Unix/Mac: 1 byte, LF ("`\n`", 0x0A)
- Difference is hidden by most editors
 - An IDE might recognize either when opening a file, but convert all to `\r\n` when saving
 - Demo: hexdump (or VSCode hex editor)
- But difference matters to git when comparing files!
- Problem: OS differences within team
 - Changing 1 line causes every line to be modified
 - Flood of spurious changes masks the real edit

Solution: Normalization

- Git convention: use `\n` in the store
 - Working tree uses OS's native eol
 - Convert when moving data between the two (e.g., commit, checkout)
- Note: Applies to *text* files only
 - A “binary” file, like a jpg, might contain these bytes (0x0D and/or 0x0A), but they should not be converted
- How does git know whether a file is text or binary?
 - Heuristics: auto-detect based on contents
 - Configuration: filename matches a pattern

Normalization With .gitattributes

- Use a .gitattributes file in root of project
 - Committed as part of the project
 - Consistent policy for everyone on team

- Example:

```
# Auto detect text files and perform LF normalization  
* text=auto
```

```
# These files are text, should be normalized (crlf=>lf)  
*.java      text  
*.md        text  
*.txt       text  
*.classpath text  
*.project   text
```

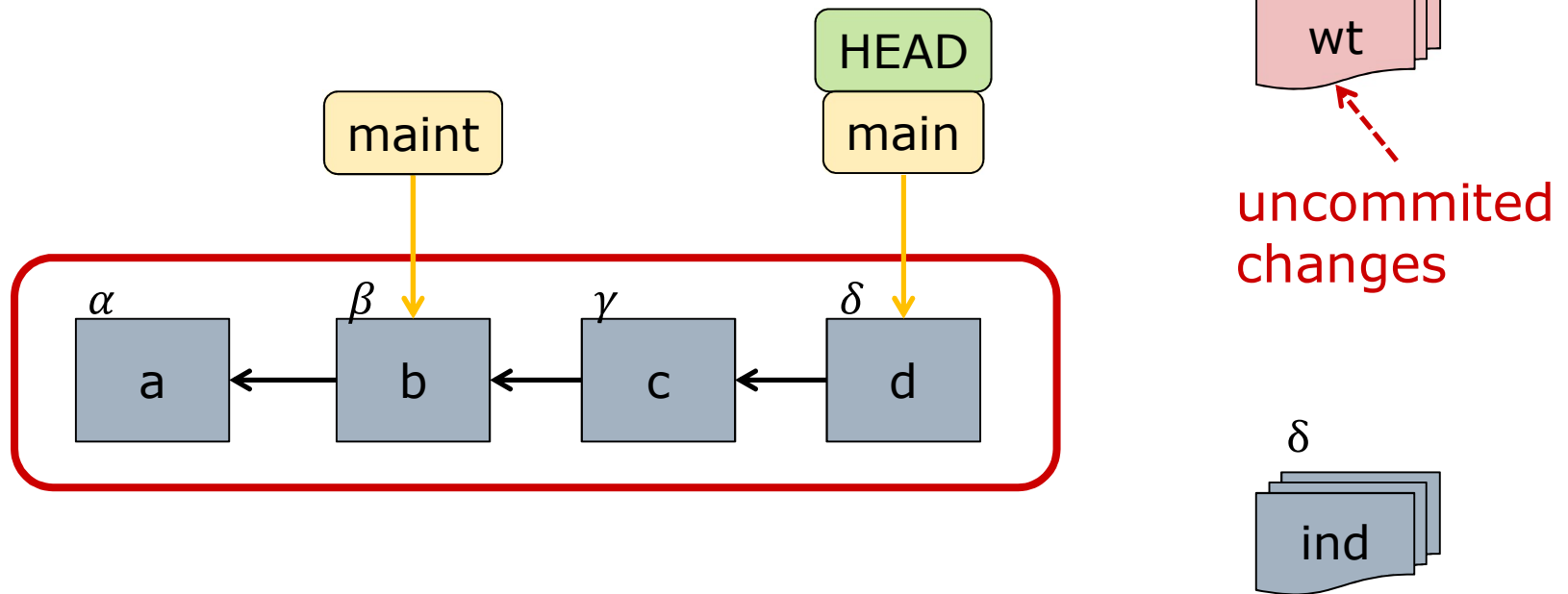
```
# These files are binary, should be left untouched  
*.class     binary  
*.jar       binary
```

Ninja Git: Advanced Moves

- ❑ Temporary storage
`stash`
- ❑ Undoing big and small mistakes in the working tree
`reset, checkout`
- ❑ Undoing mistakes in store
`amend`
- ❑ DAG surgery
`rebase`

Advanced: Temporary Storage

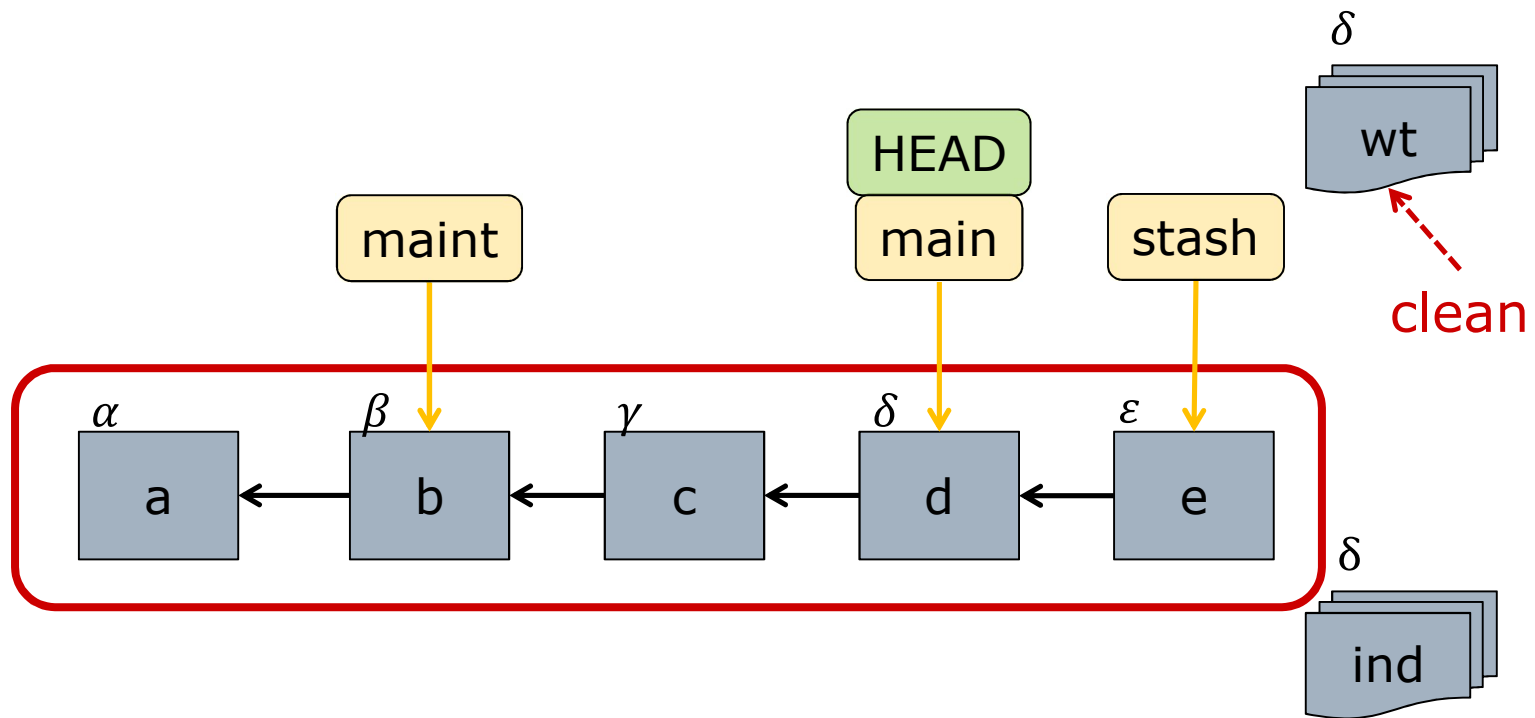
- Say you have uncommitted work and want to look at a different branch
- Checkout won't work!



Stash: Push Work Onto A Stack

```
$ git stash # repo now clean
```

```
$ git checkout ...etc... # feel free to poke around
```



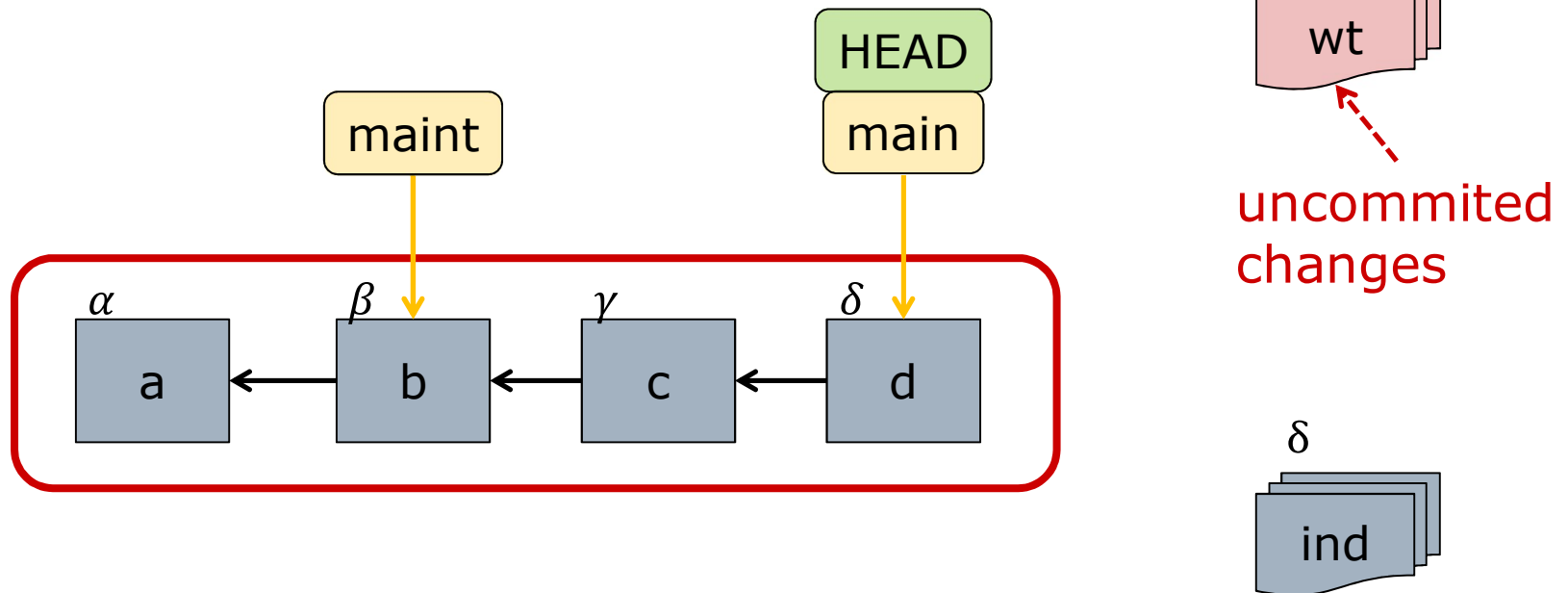
Stash: Pop Work Off the Stack

```
$ git stash pop # restores state of wt (and store)
```

```
# equivalent to:
```

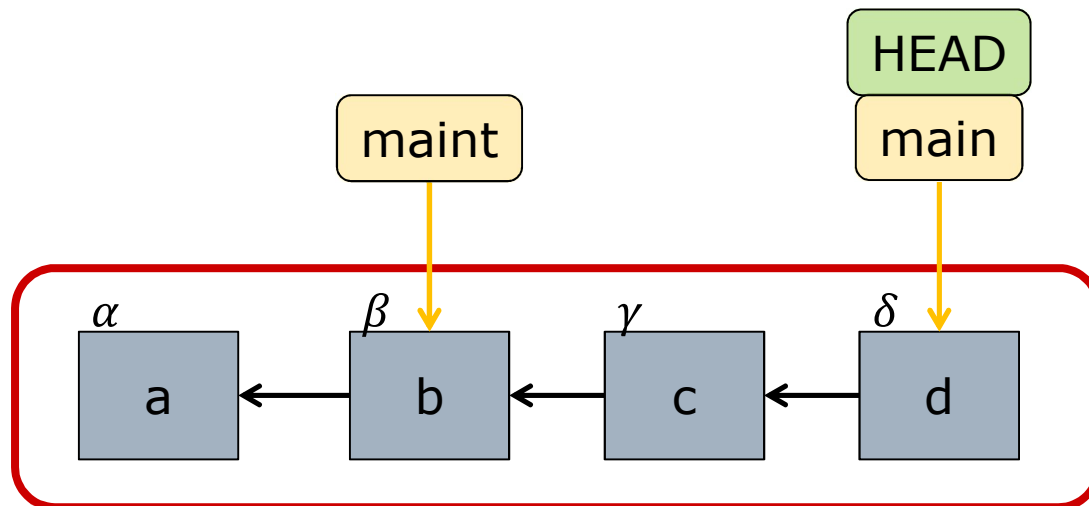
```
$ git stash apply # restore wt and index
```

```
$ git stash drop # restore store
```



Advanced: Undoing Big Mistakes

- Say you want to throw away *all* your uncommitted work
 - ie “Roll back” to last committed state
- Checkout HEAD won't work!



ϵ

wt

uncommitted changes

The diagram shows a stack of three pink rectangular boxes representing uncommitted changes. The top box is labeled 'wt'. A red dashed arrow points from the text 'uncommitted changes' to the 'wt' box. A red Greek letter epsilon (ϵ) is positioned above the stack.

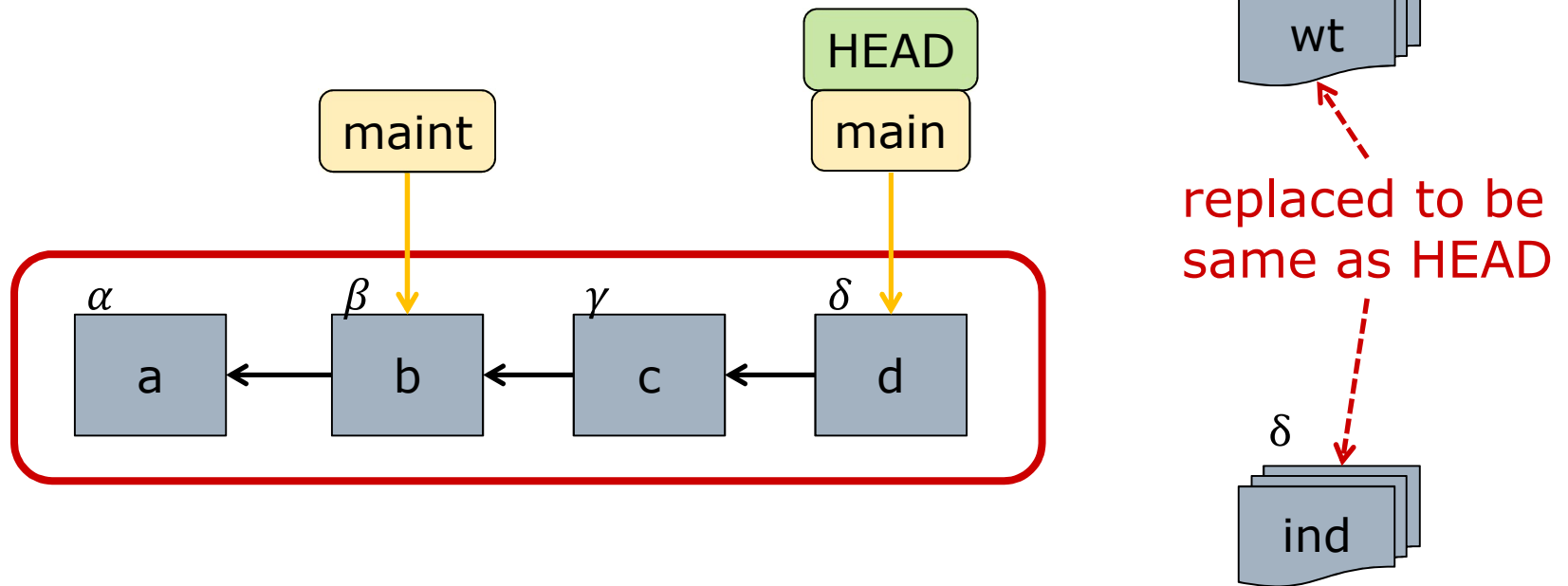
δ

ind

The diagram shows a stack of three blue rectangular boxes representing the index. The top box is labeled 'ind'. A blue Greek letter delta (δ) is positioned above the stack.

Reset: Discarding Changes

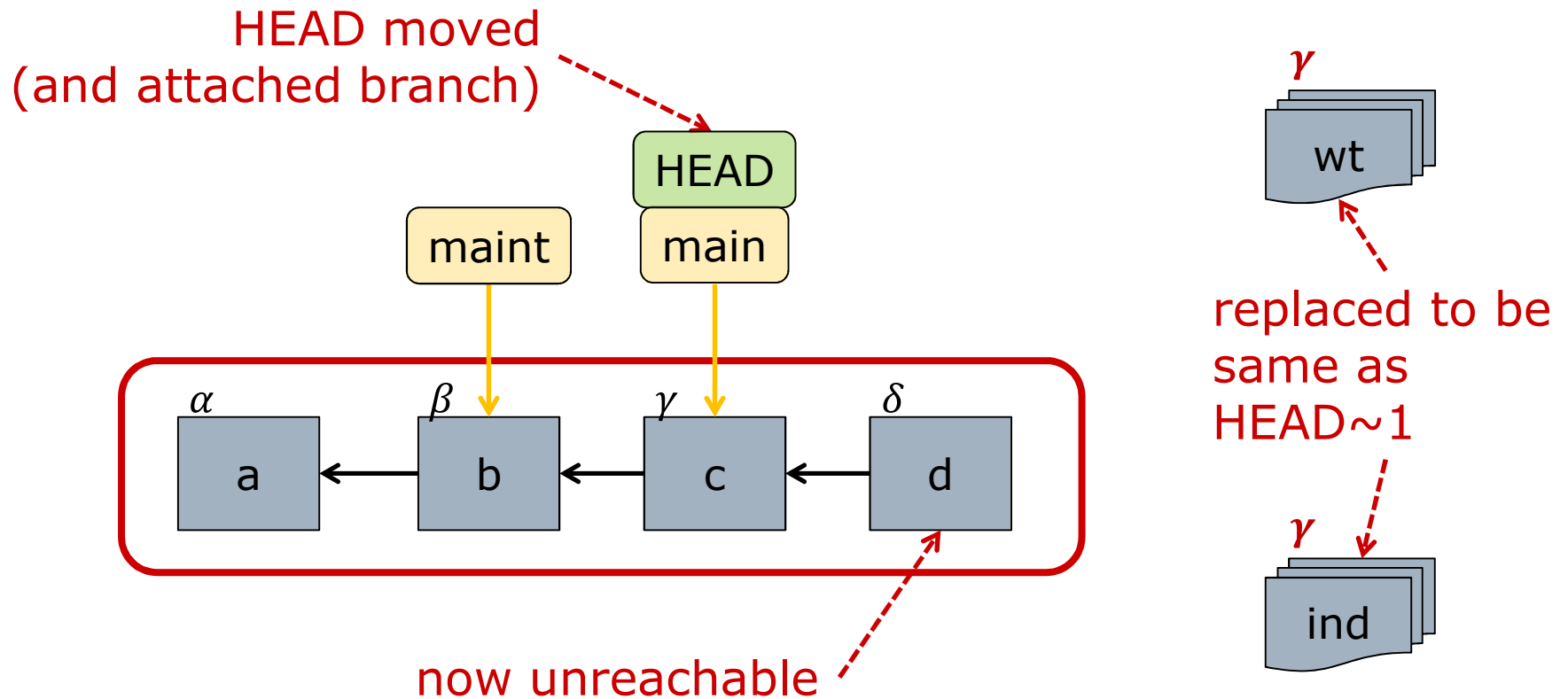
```
$ git reset --hard  
$ git clean --dry-run # list untracked files  
$ git clean --force # remove untracked files
```



Reset: Discarding Commits

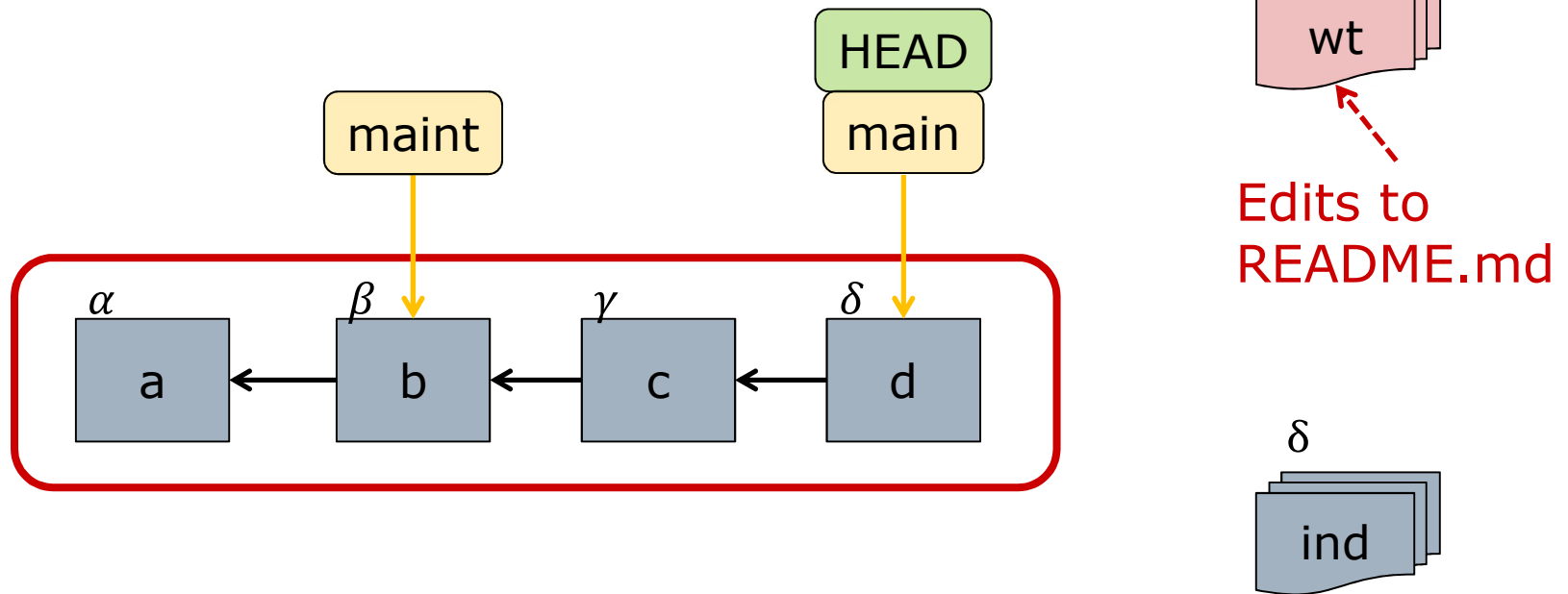
```
$ git reset --hard HEAD~1
```

no need to git clean, since wt was already clean



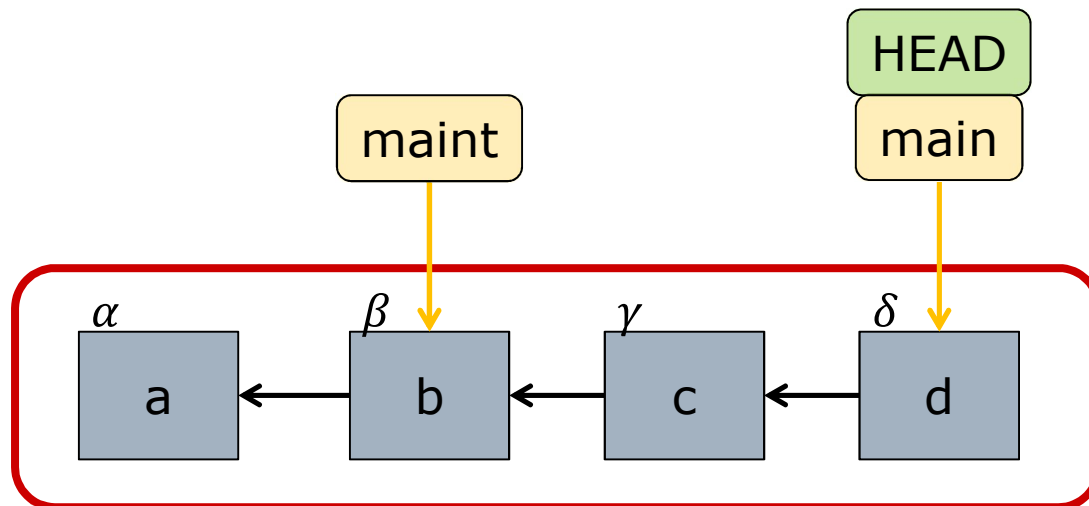
Advanced: Undo Small Mistakes

- Say you want to throw away *some of* your uncommitted work
 - Restore a file to last committed version



Advanced: Undo Small Mistakes

```
$ git checkout -- README.md  
# -- means: rest is file/path (not branch)  
# git checkout README.md ok, if not ambiguous
```



ϵ'

wt

README.md matches δ

δ

ind

Advanced: Rewriting History

Computer Science and Engineering ■ The Ohio State University



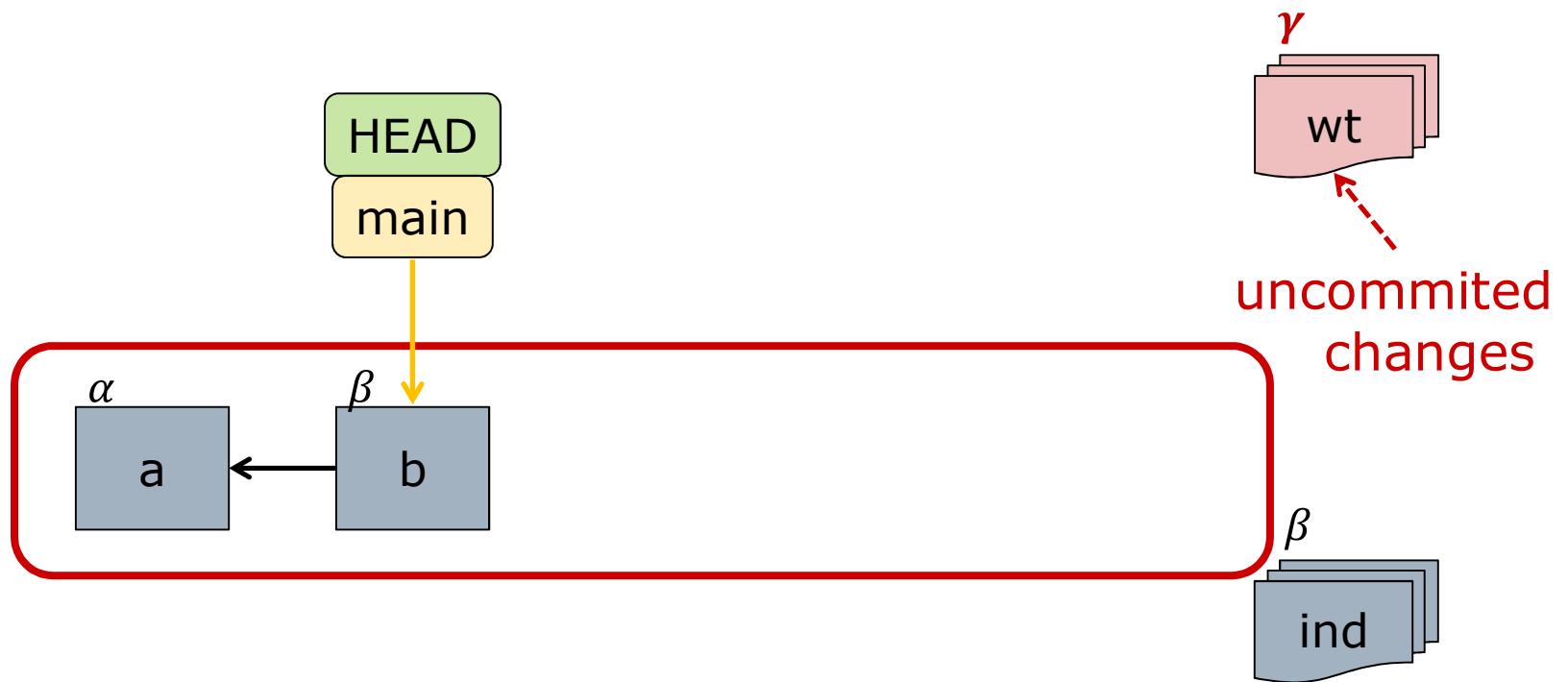
The Power to Change History

- Changing the store lets us:
 - Fix mistakes in recent commits
 - Clean up messy DAGs to make history look more linear

- Rule: Never change *shared* history
 - Once something has been pushed to a remote repo (e.g., origin), do not change that part of the DAG
 - So: A *push* is really a *commitment*!

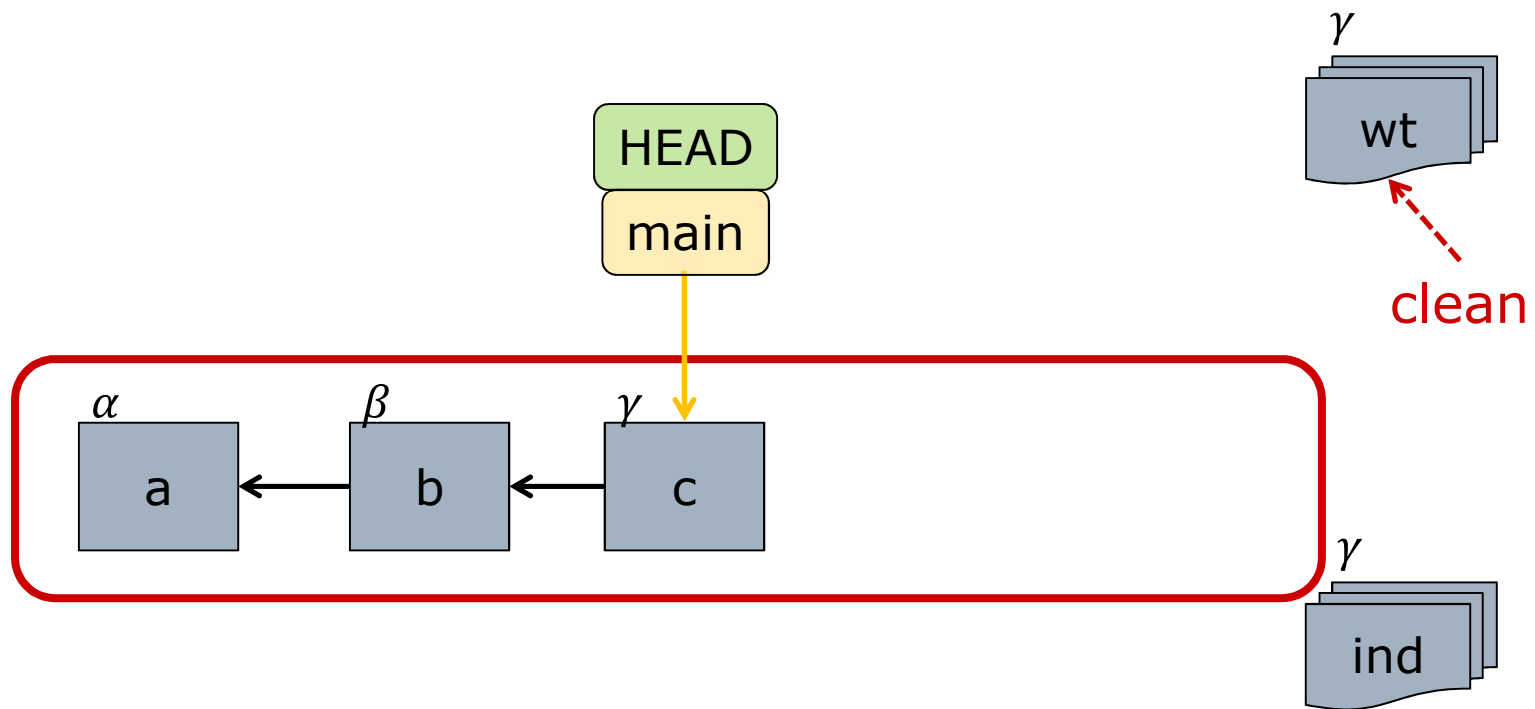
Advanced: Rewriting History

- ❑ Problem 1: Wrong or incomplete commit



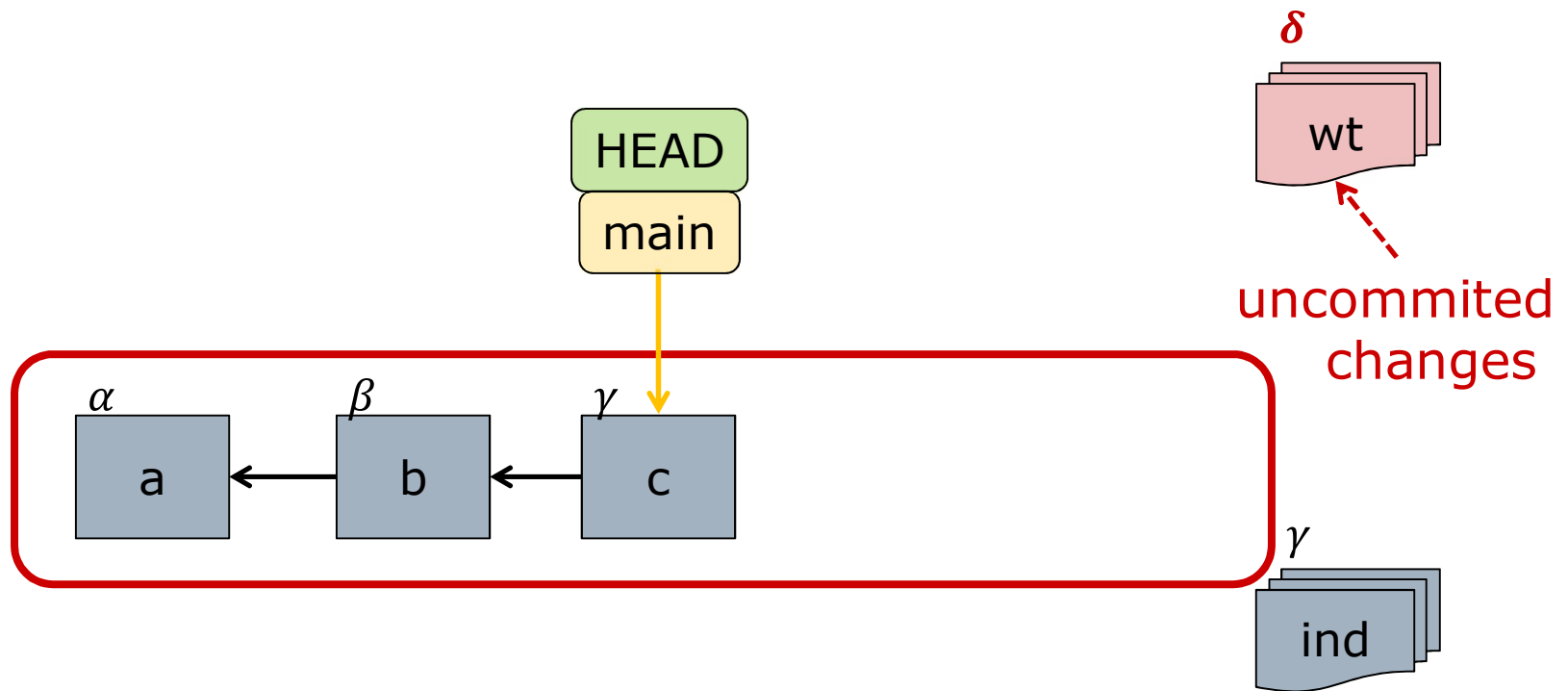
Advanced: Rewriting History

- ❑ Problem 1: Wrong or incomplete commit



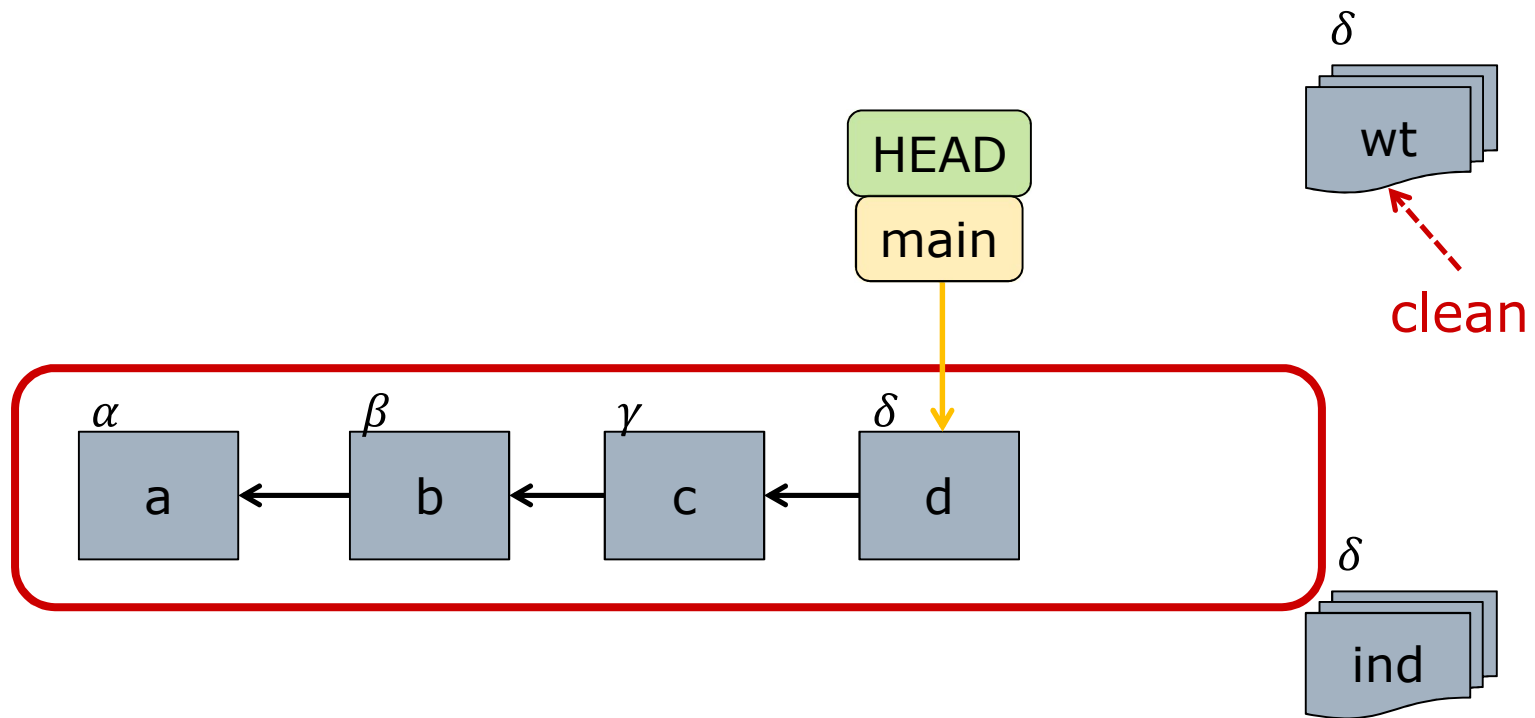
Advanced: Rewriting History

- ❑ Problem 1: Wrong or incomplete commit
 - Oops! That wasn't quite right...



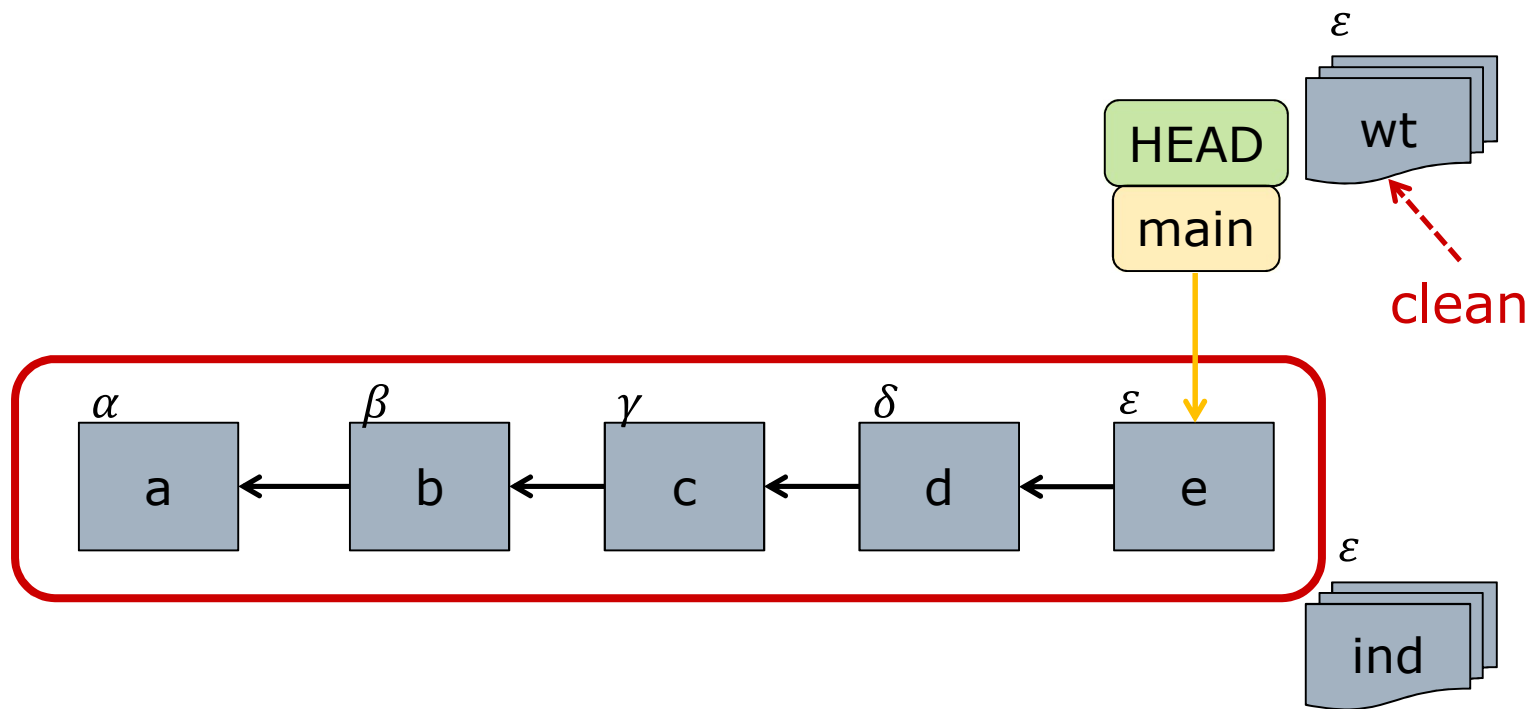
Advanced: Rewriting History

- ❑ Problem 1: Wrong or incomplete commit
 - Oops! That wasn't quite right...



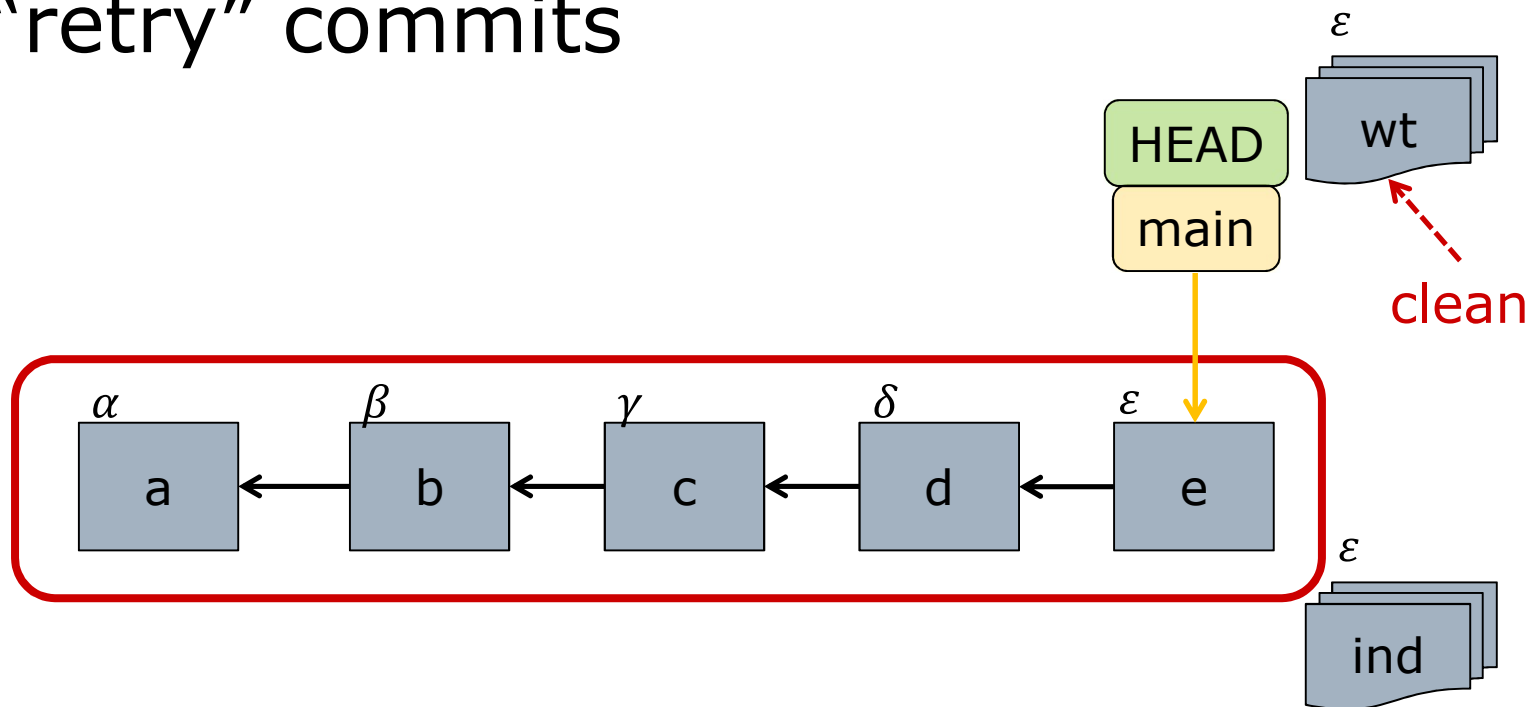
Advanced: Rewriting History

- ❑ Problem 1: Wrong or incomplete commit
 - Oops! That wasn't quite right...



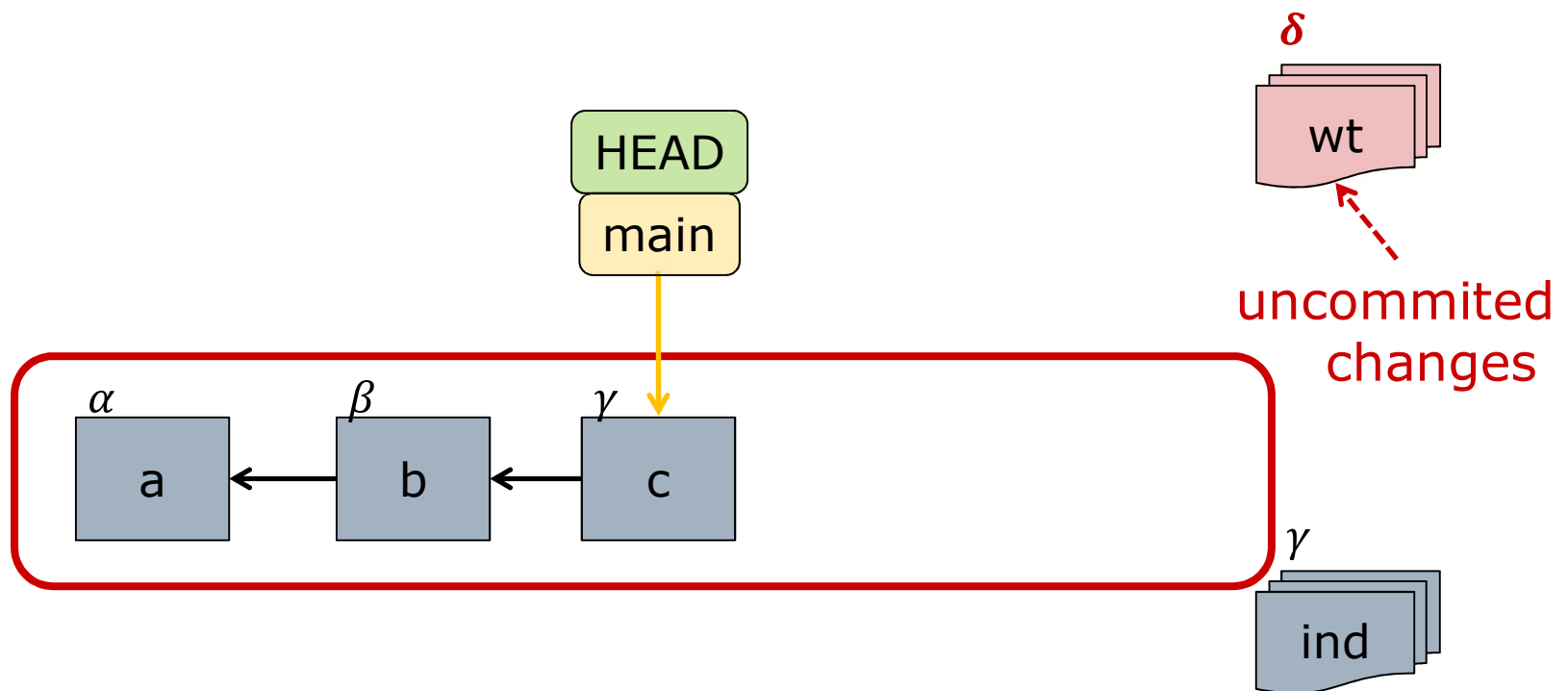
Advanced: Rewriting History

- ❑ Problem 1: Wrong or incomplete commit
- ❑ Result: Lots of tiny “fix it”, “oops”, “retry” commits



Commit --amend: Tip Repair

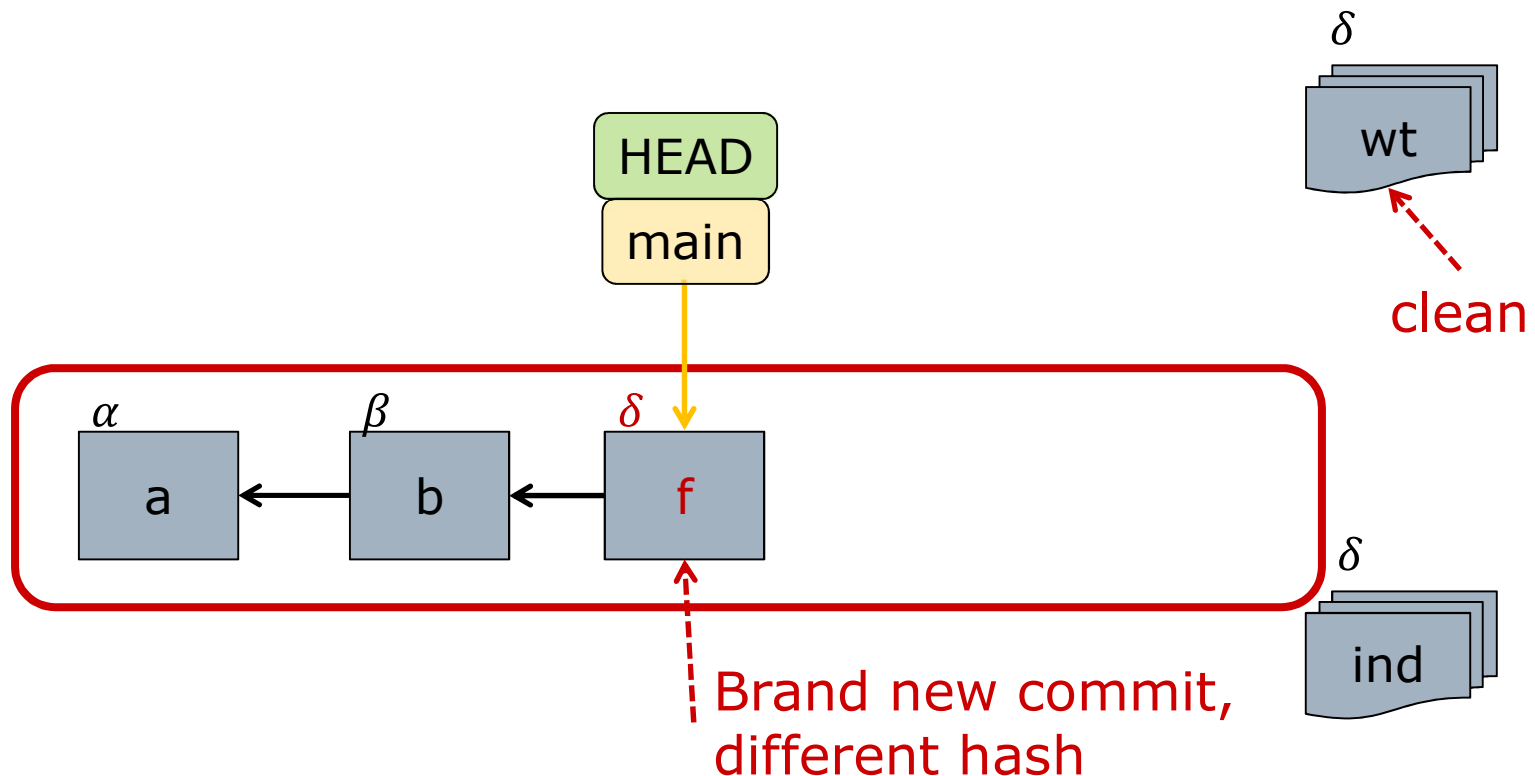
- Alternative: Change most recent commit(s)



Commit --amend: Tip Repair

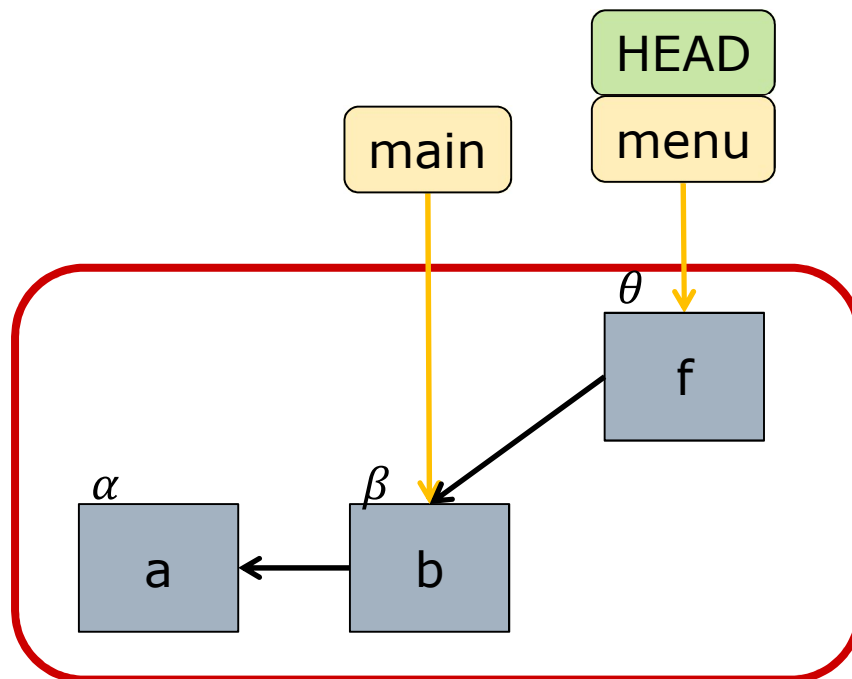
Computer Science and Engineering ■ The Ohio State University

```
$ git add --all .  
$ git commit --amend --no-edit  
# no-edit keeps the same commit message
```



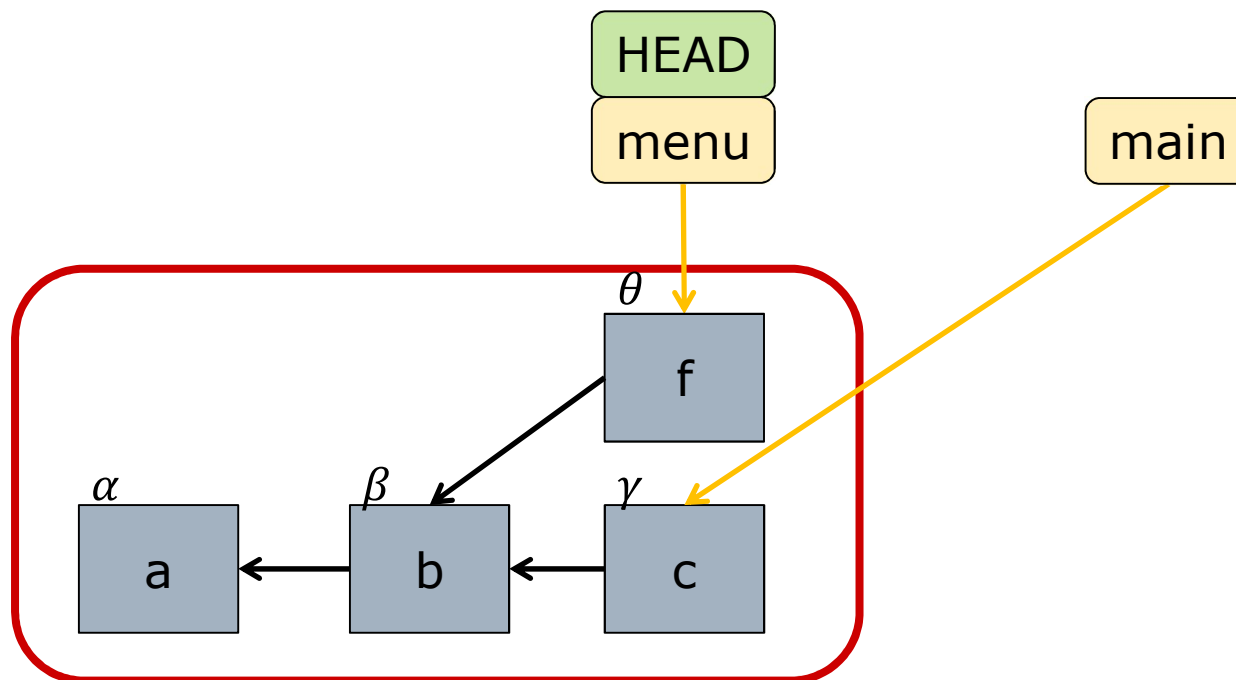
Advanced: Rewriting History

- Problem 2: As an independent branch is being developed, main also evolves



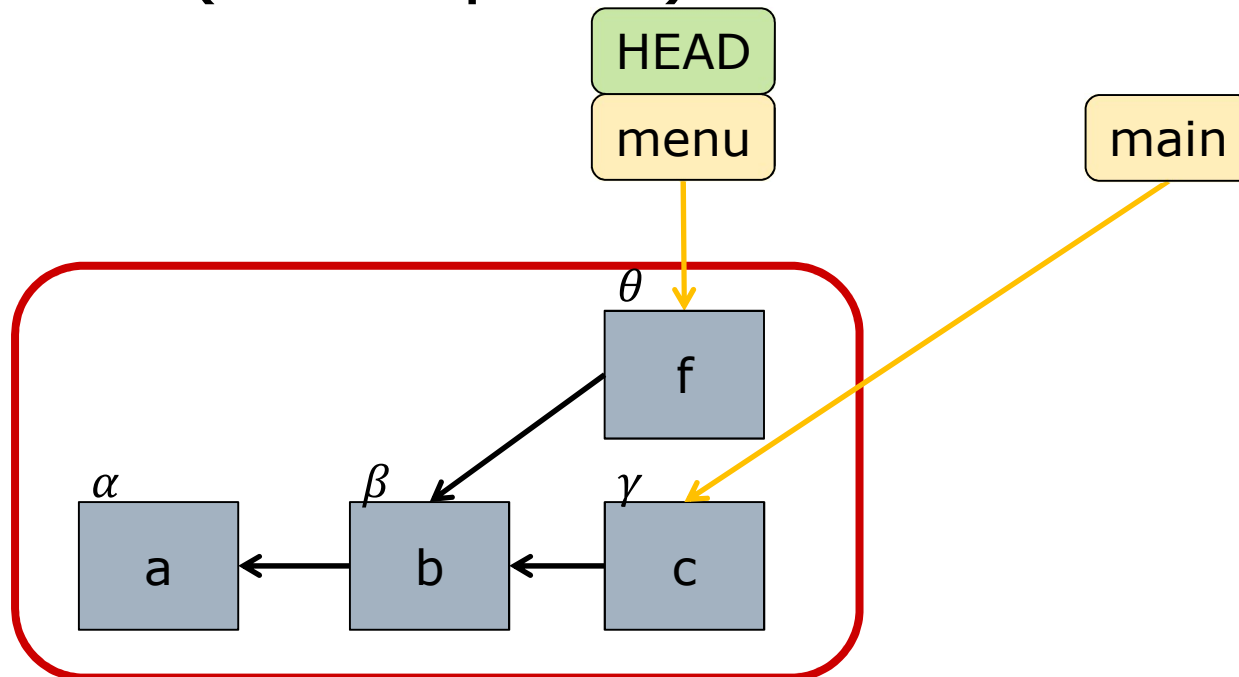
Advanced: Rewriting History

- Problem 2: As an independent branch is being developed, main also evolves



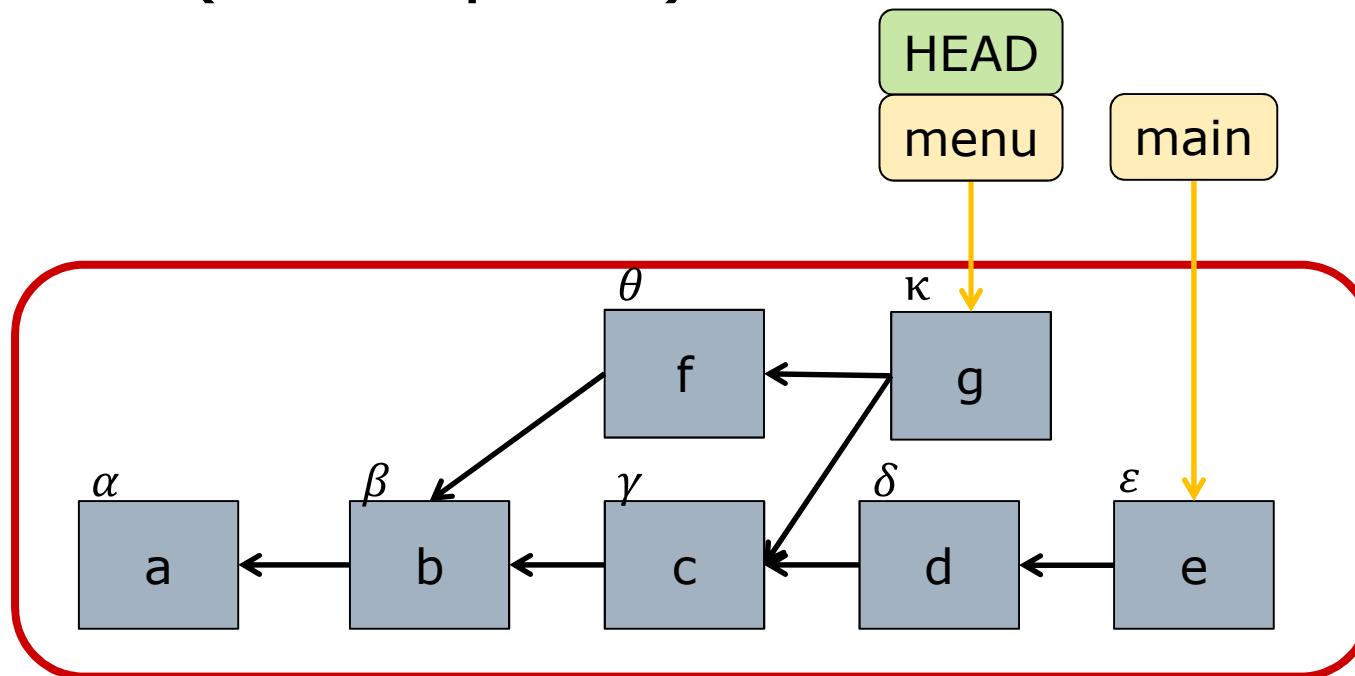
Advanced: Rewriting History

- ❑ Problem 2: As an independent branch is being developed, main also evolves
- ❑ Result: Need periodic merges of main with (incomplete) branch



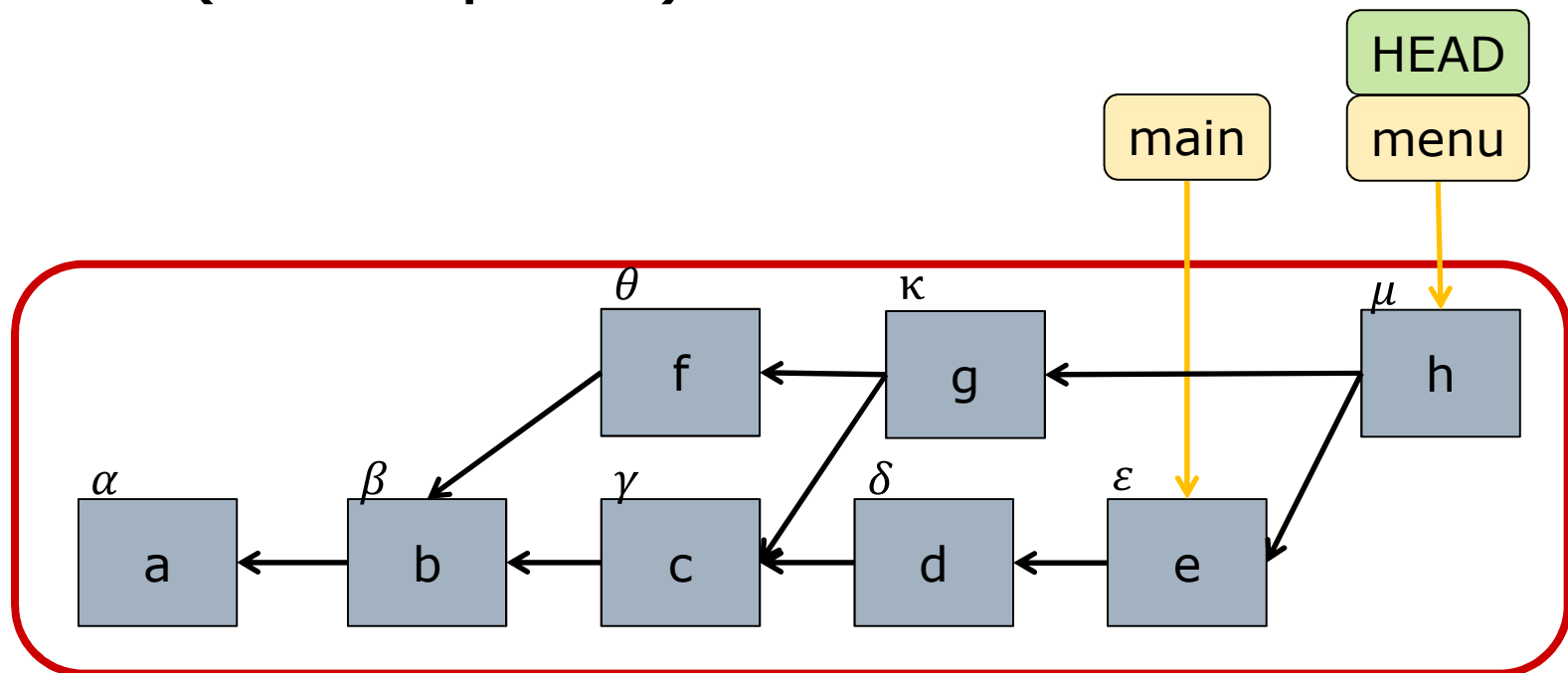
Advanced: Rewriting History

- ❑ Problem 2: As an independent branch is being developed, main also evolves
- ❑ Result: Need periodic merges of main with (incomplete) branch



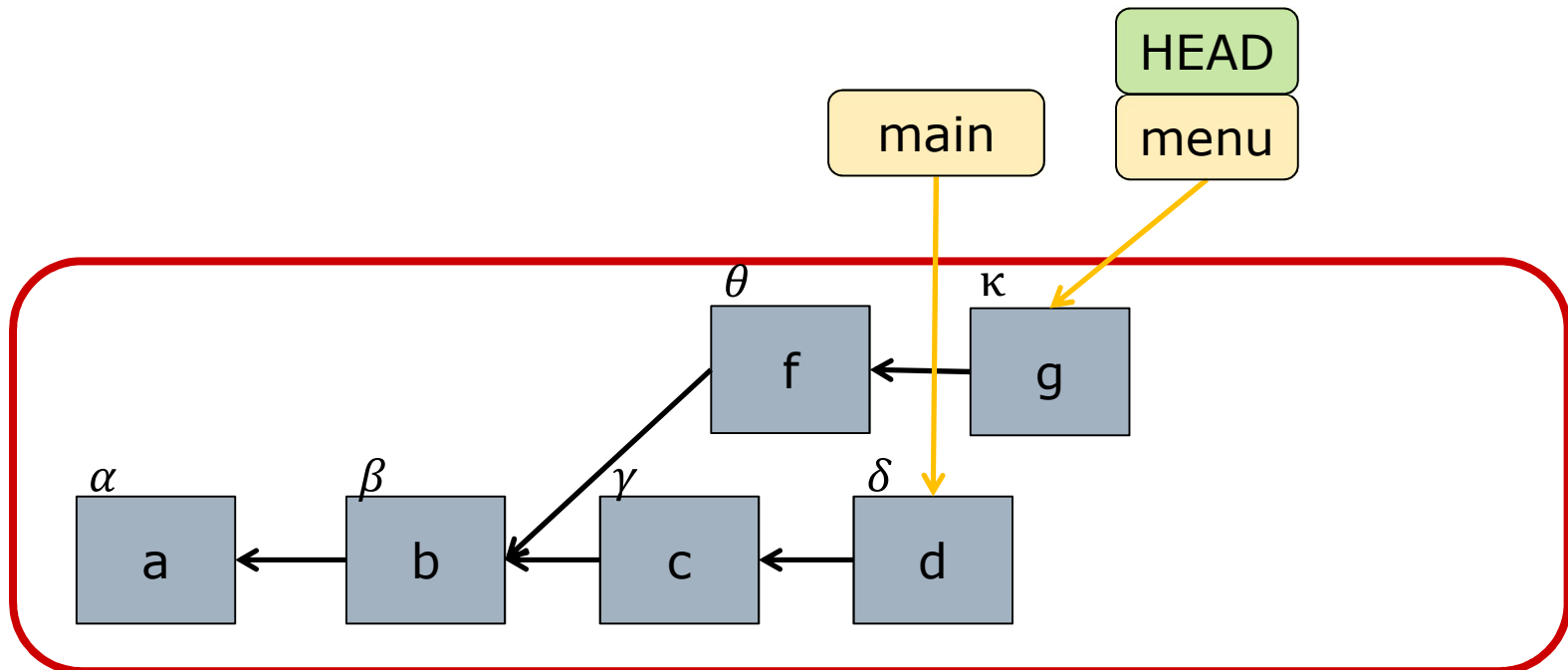
Advanced: Rewriting History

- ❑ Problem 2: As an independent branch is being developed, main also evolves
- ❑ Result: Need periodic merges of main with (incomplete) branch



Rebase: DAG Surgery

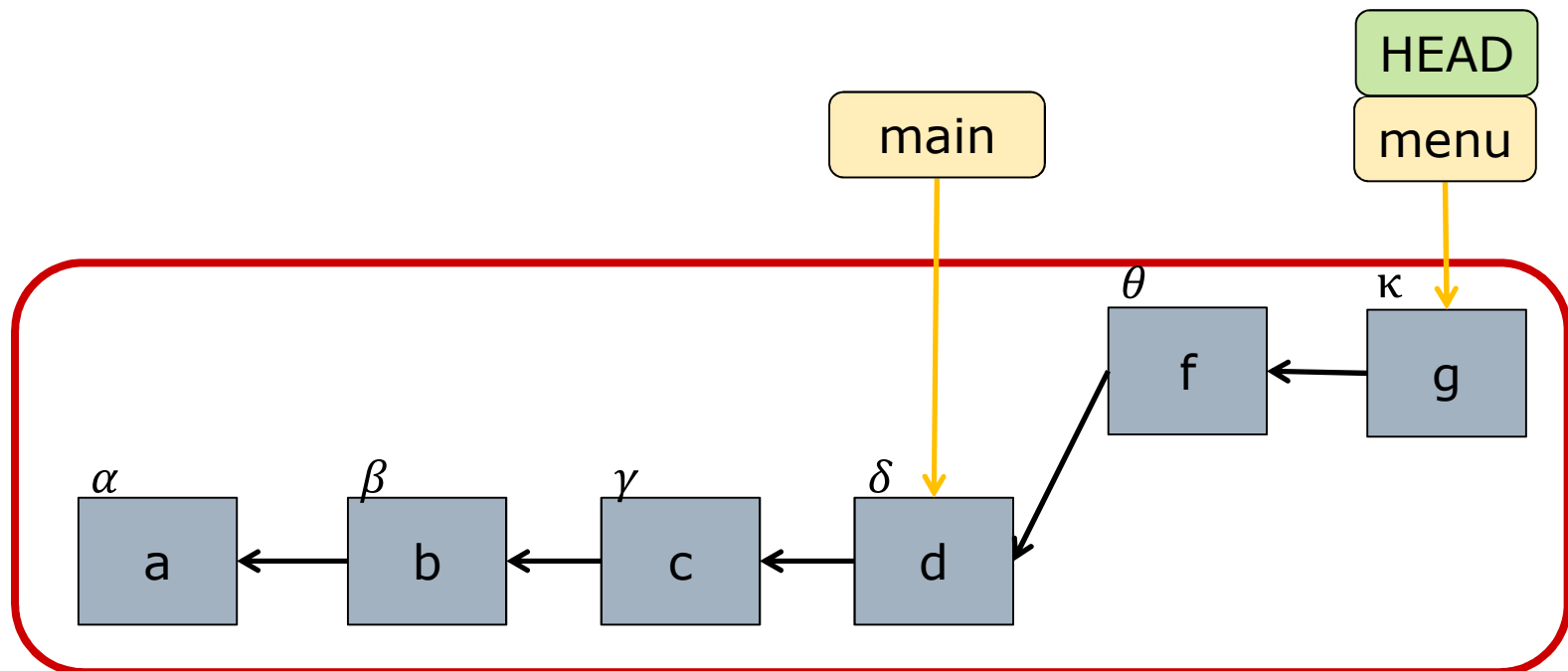
- Alternative: Move commits to a different part of the DAG



Rebase: DAG Surgery

```
$ git rebase main
```

```
# merging main into menu is now a fast-forward
```



Git Clients and Hosting Services

- ❑ Recommended client: Command line!
- ❑ Alternative: IDEs
 - VSCode, plus Git Graph extension
- ❑ Lots of sites for hosting your repos:
 - GitHub, GitLab, Bitbucket, SourceForge...
 - See: git.wiki.kernel.org/index.php/GitHosting
- ❑ These cloud services provide
 - Storage space, account/access management
 - Pretty web interface
 - Issues, bug tracking
 - Workflow (eg forks) to promote contributions from others

Clarity

git != GitHub



Warning: Academic Misconduct

- ❑ GitHub is a very popular service
 - New repos are *public* by default
 - But even free plan allows unlimited *private* repo's (and collaborators)
 - 3901 has an "organization" for your private repo's and team access
- ❑ Other services (e.g. GitLab, Bitbucket) have similar issues
- ❑ Public repo's containing coursework can create academic misconduct issues
 - Problems for poster
 - Problems for plagiarist

Mercurial (hg): Another DVCS

- Slightly simpler mental model
- Some differences in terminology
 - git fetch/pull $\sim$ hg pull/fetch
 - git checkout $\sim$ hg update
- Some (minor) differences in features
 - No rebasing (only merging)
 - No octopus merge ($\# \text{parents} \leq 2$)
- But key ideas are identical
 - Repository = working directory + store
 - Send/Receive changes between stores

Summary

- ❑ Workflow
 - Fetch/push frequency
 - Respect team conventions for how/when to use different branches
- ❑ Central repo is a shared resource
 - Contains common (source) code
 - Normalize line endings and formats
- ❑ Advanced techniques
 - Stash, reset, rebase
- ❑ Advice
 - Learn by using the command line
 - Beware academic misconduct