CMPS 270: Software Construction
Fall 2022

VERSION CONTROL SYSTEMS

Adapted from: 6.031 Fall 2019 at MIT

Rida Assaf
Department of Computer Science

MOTIVATION

- Keeping track of previous file/codebase/… versions
- Being able to revert to any of them
- Pushing full version history to another location
- Pulling history from that location
- Comparing different versions
- Merging versions that are variations of some earlier version

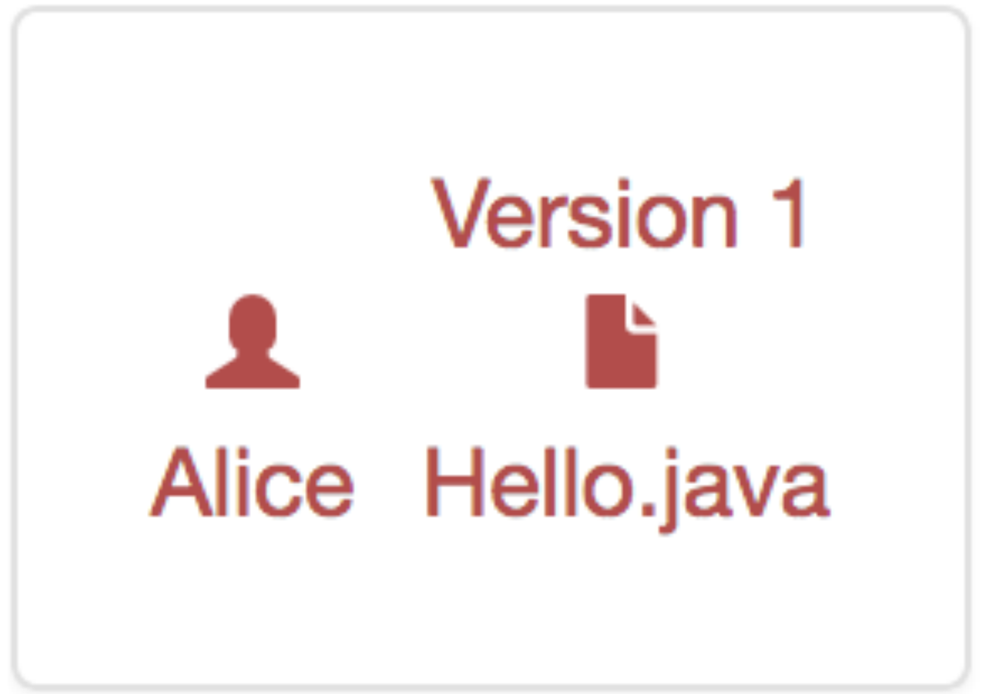
FAMILIAR EXAMPLES

Dropbox

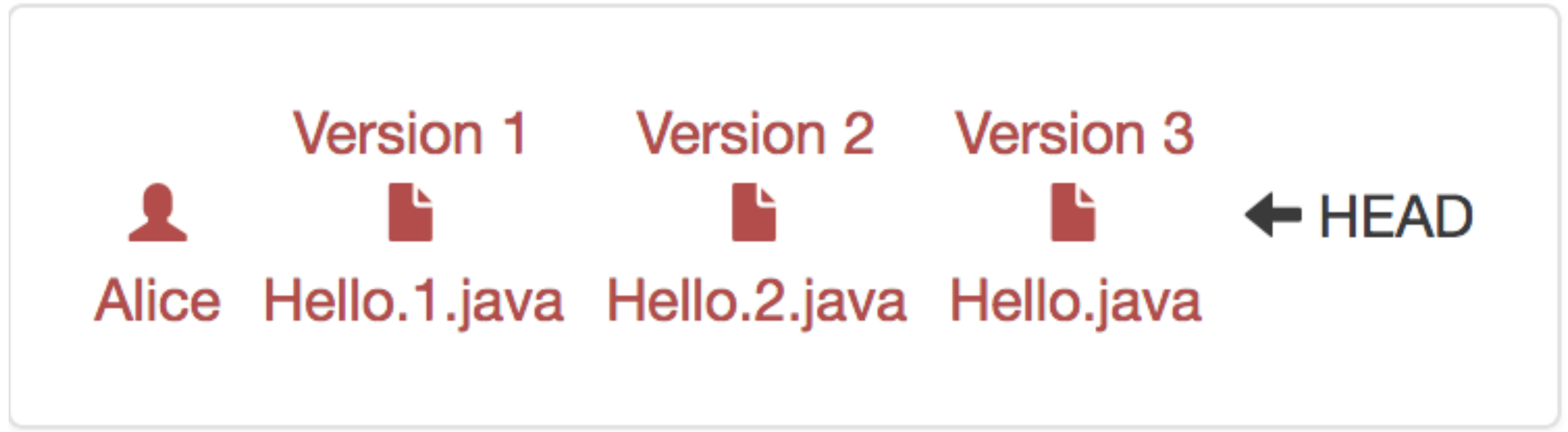
Keeping multiple local copies of files with version numbers

USE CASE SCENARIO

- Imagine working on a project for a while and keeping track of different versions in different files. Think of adding new files as a stack, the most recent is the head. You realize the most recent version introduces bugs.



- **Solution:** copy some old version.



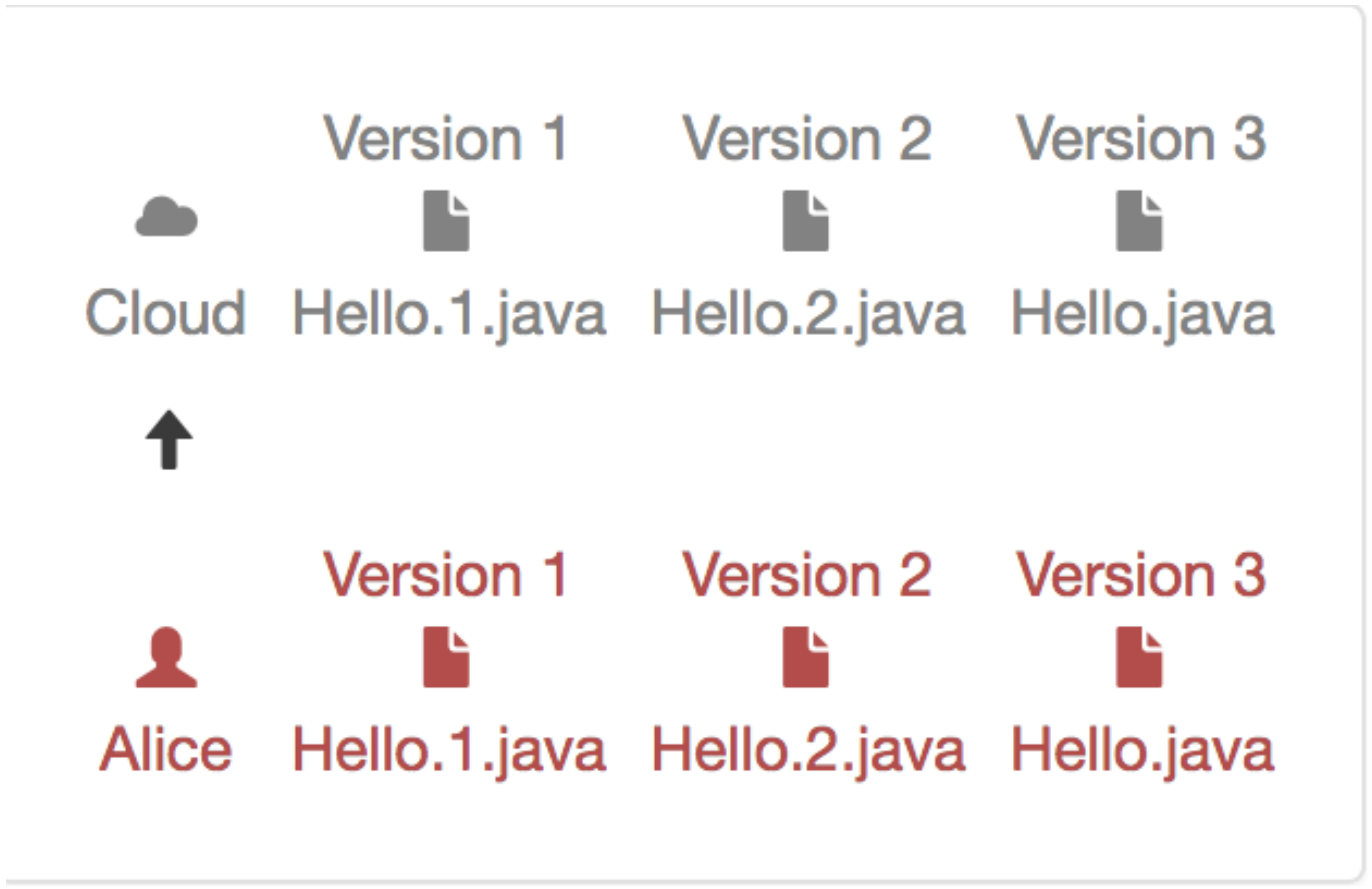
USE CASE SCENARIO CONT'D

- This may introduce a new problem: new features in the most recent version will be lost.
- **Solution:** Manually compare and copy the wanted pieces of code.

Too much work though. We need a way to quickly find differences between versions.

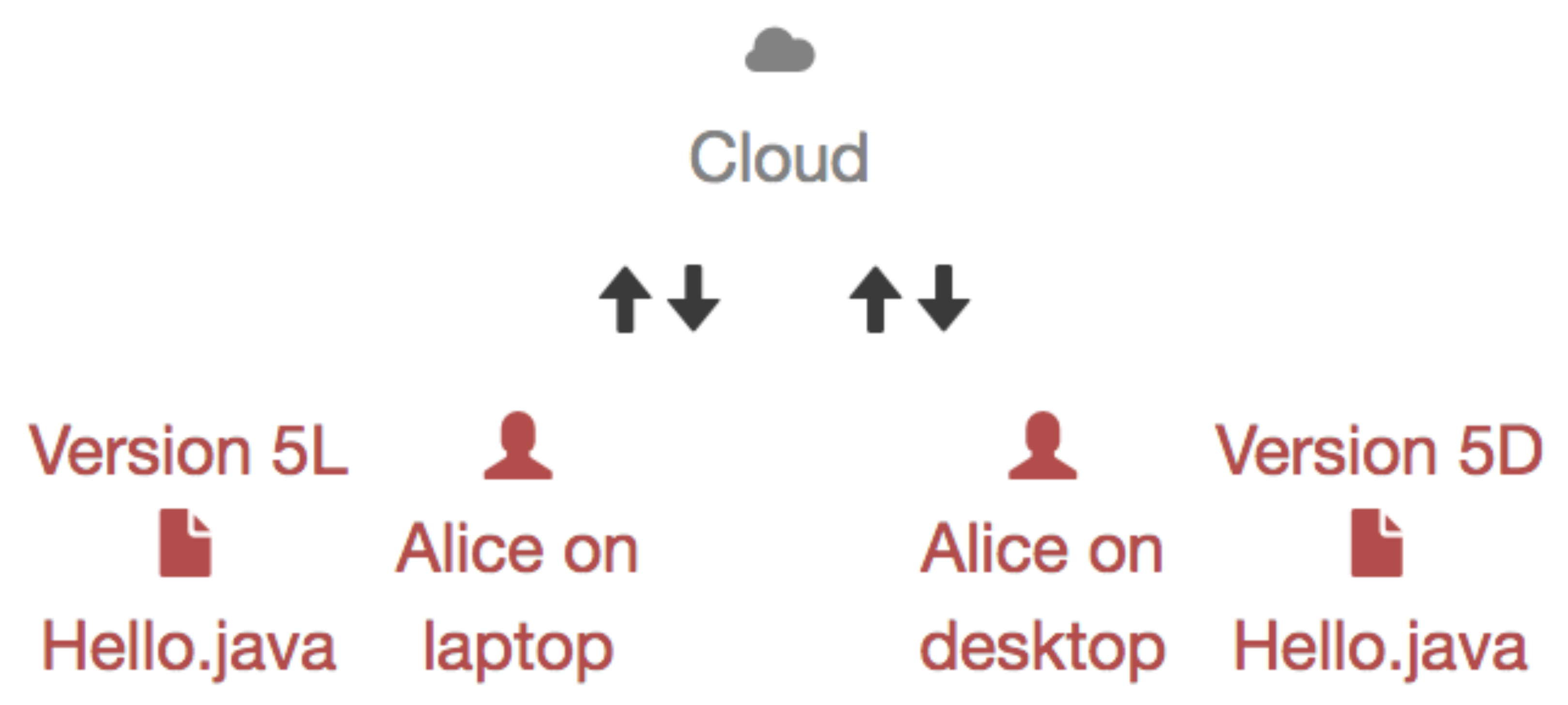
USE CASE SCENARIO CONT'D

- Alice has lost work due to machine failure before.
- **Solution:** save a copy on the cloud. We need a way to quickly keep a most updated version in the cloud.



USE CASE SCENARIO CONT'D

This also solves the issue of working on the code from multiple machines, for example home computer and university computer. But this is a delicate matter: work started on one computer and not pushed to the cloud before resuming on another computer might cause conflicts or overwriting. So we need a good way to synchronize.



SUMMARY OF USES SO FAR

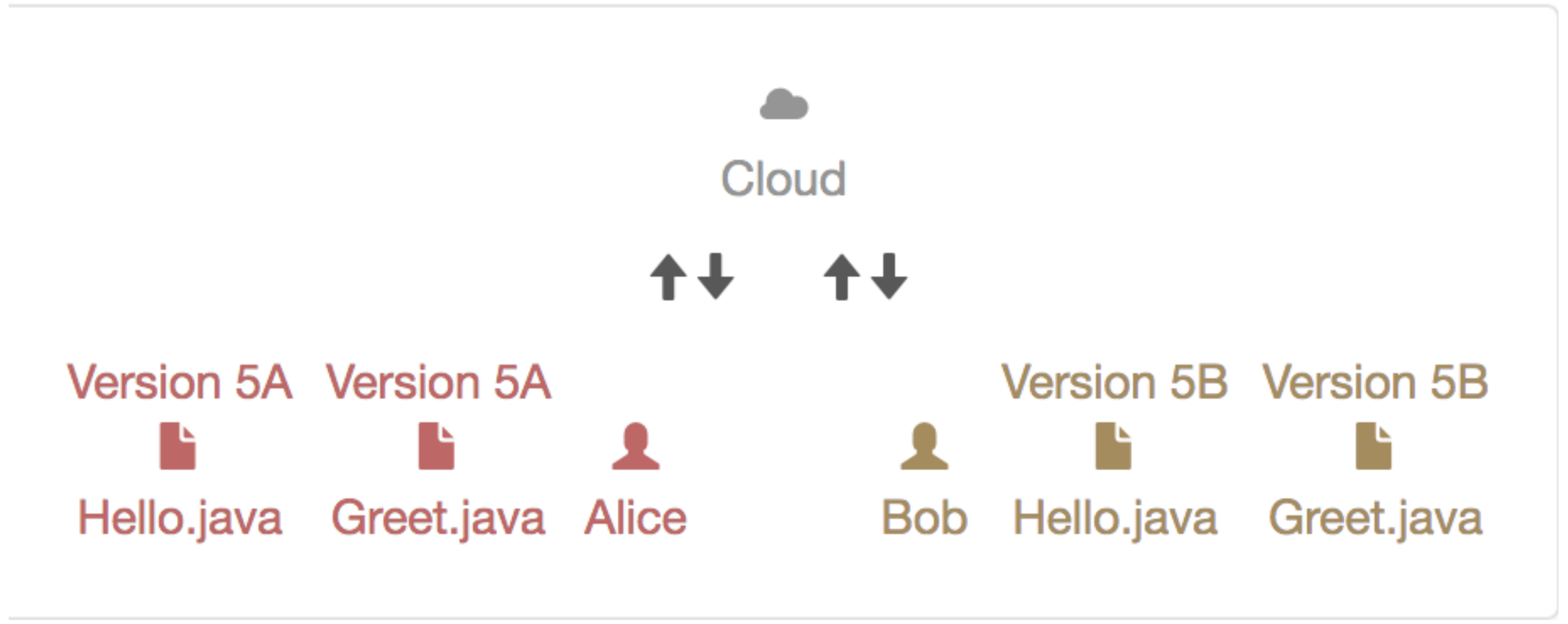
At this point, considering just the scenario of one programmer working alone, we already have a list of operations that should be supported by a version control scheme:

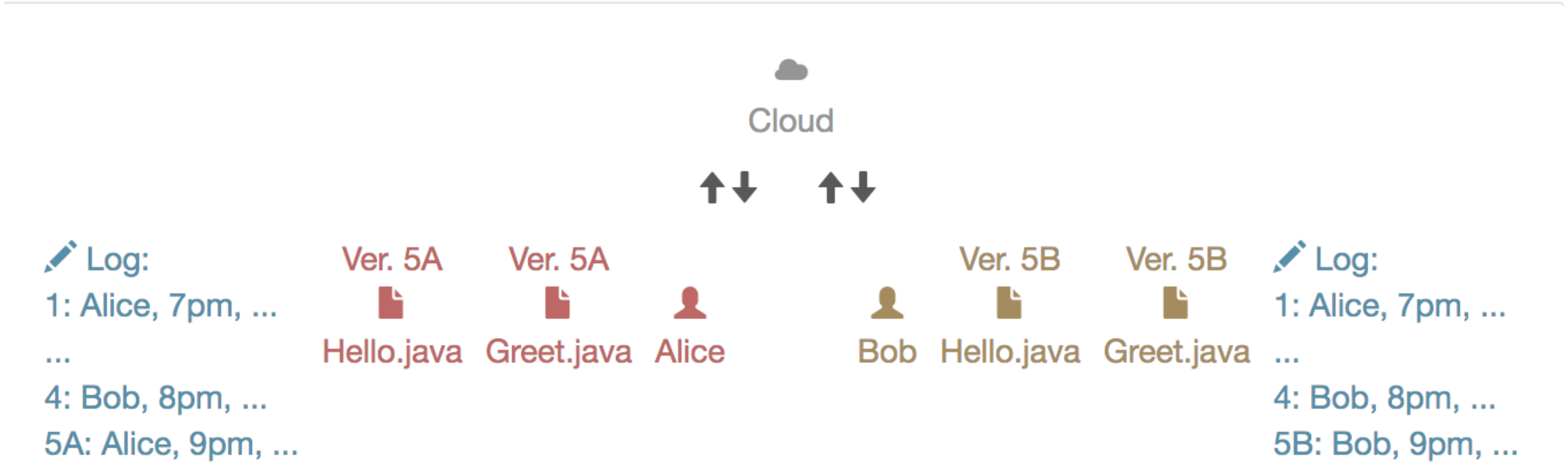
- Reverting to a past version.
- Comparing two different versions.
- Pushing full version history to another location.
- Pulling history back from that location.
- Merging versions that are offshoots of the same earlier version.

USE CASE SCENARIO CONT'D

- Now if the project includes **multiple developers,** it's harder to agree on a scheme to **save** multiple versions, and to **synchronize** all updates without messing things up. One thing we need is to **log** changes made by every developer every time. Another thing we need is the ability to work on **different features** of the project **at the same time**, and to make it easy to **merge** updates.
- It may also be necessary and useful to work on different **branches** of the project. Say we have some **stable version** that is **deployed** and we want to **add** or **modify** features without breaking things. Then what we need is a **copy** of the stable version that we can change **locally** but only **push** to the **cloud** when done and properly tested. The same developer or multiple ones may need multiple branches.

USE CASE SCENARIO CONT'D





VERSION CONTROL SYSTEMS (VCS) TO THE RESCUE

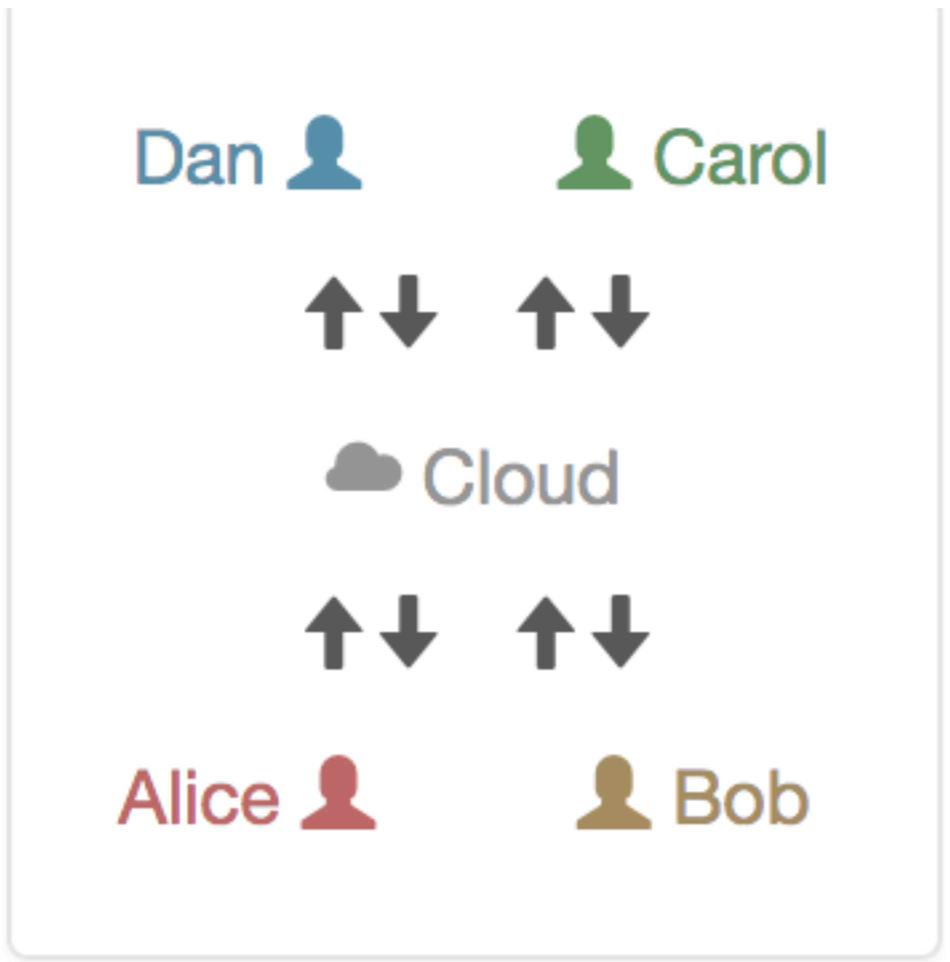
There are mainly two types of version control systems

Central
Distributed

CENTRALIZED VCS

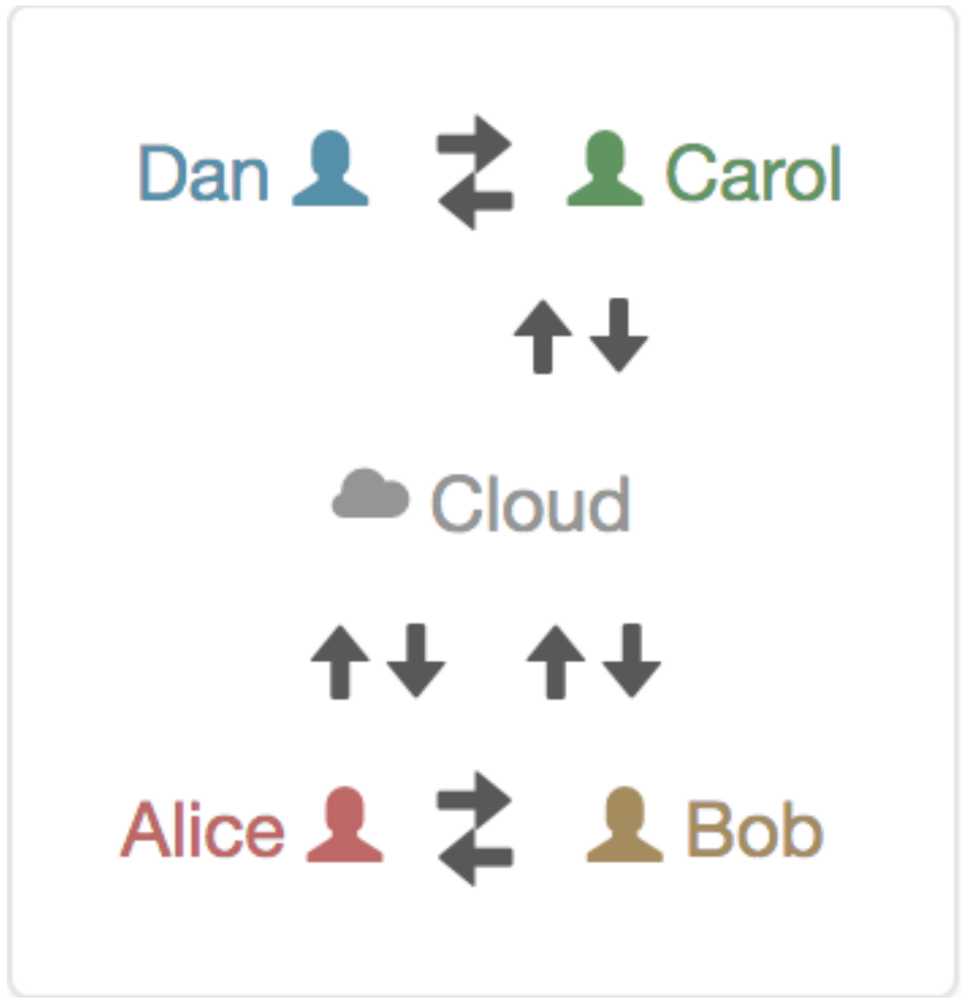
- Traditional centralized version control systems like CVS and Subversion do a subset of the things we've imagined above. They support a collaboration graph – who's sharing what changes with whom – with one master server and copies that only communicate with the master.

- In a centralized system, everyone must share their work to and from the **master repository.** Changes are safely stored in version control if they are in the master repository, because that's the **only repository.**



DISTRIBUTED VCS

- In contrast, distributed version control systems like Git and Mercurial allow all sorts of different collaboration graphs, where teams and subsets of teams can experiment easily with alternate versions of code and history, merging versions together as they are determined to be good ideas.
- In a distributed system, all repositories are created equal, and it's up to users to assign them different roles. Different users might share their work to and from different repos, and the team must decide what it means for a change to be in version control.



GIT/GITHUB

- Git is the leading VCS available. It does all the things we mentioned earlier and more.
- There are centralized and distributed version control systems. **Git is distributed.**
- Github is a cloud server that offers services relevant (storing code synchronizing etc)

VERSION CONTROL TERMINOLOGY

- **Repository:** a local or remote store of the versions in our project.
- **Working copy:** a local, editable copy of our project that we can work on.
- **File:** a single file in our project.
- **Version or revision:** a record of the contents of our project at a point in time.
- **Change or diff:** the difference between two versions.
- **Head:** the current version.

FEATURES OF A VCS

- **Reliable:** keep versions around for as long as we need them; allow backups.
- **Multiple files:** track versions of a project, not single files.
- **Meaningful versions:** what were the changes, why were they made?
- **Revert:** restore old versions, in whole or in part.
- **Compare versions.**
- **Review history:** for the whole project or individual files.
- **Not just for code:** prose, images, …

FEATURES OF A VCS CONT'D

It should allow multiple people to work together:

- **Merge:** combine versions that diverged from a common previous version.
- **Track responsibility:** who made that change, who touched that line of code?
- **Work in parallel:** allow one programmer to work on their own for a while (without giving up version control).
- **Work-in-progress:** allow multiple programmers to share unfinished work (without disrupting others, without giving up version control).

GIT

Git has some visual UI you can use, but you should pretend that does not exist. You should be comfortable using the Command Line (CMD) version of it because that’s how you’ll likely encounter it in a work setting.

GITHUB

- GitHub is a server you can store your project on. Or surf others and clone them. It's useful for you and more useful than a resume. I advise you to make a habit out of keeping your course projects on Github.
- GitHub.com is a site for online storage of Git repositories:
  - You can create a remote repo there and push code to it.
  - Many open source projects use it, such as the Linux kernel. Free for open source, pay for private and extra space.
  - Free private repos for educational use: github.com/edu

Question: Do I always have to use GitHub to use Git?

- Answer: No! You can use Git locally for your own purposes. Or you or someone else could set up a server to share files. Or you could share a repo with users on the same file system, as long everyone has the needed file permissions.

BASIC GIT USAGE

- **Initialize** or **clone** some repo.
- **Add** the files you want to track.
- **Commit** the changes you make with some meaningful message.
- **Push** those changes to the cloud.
- **Pull** those changes when needed.

THE BIG THREE

Safe from bugs

- Find when and where something broke
- Look for other, similar mistakes
- Gain confidence that code hasn’t changed accidentally

Ready for change

- All about managing and organizing changes.
- Accept and integrate changes from other developers.
- Isolate speculative work on branches.

Easy to understand

- Why was a change made?
- What else was changed at the same time?
- Who can I ask about this code?

INITIAL GIT CONFIGURATION

- Set the name and email for Git to use when you commit:

  – **git config --global user.name "Bugs Bunny"**

  – **git config --global user.email bugs@gmail.com**

  You can call **git config –list** to verify these are set.

- Set the editor that is used for writing commit messages:

  – **git config --global core.editor nano** (it is vim by default)

SETTING UP A GIT REPO

Two common scenarios (only do one of these):

To create a new local Git repo in your current directory:

- **`git init`**
  This will create a **.git** directory in your current directory. Then you can commit files in that directory into the repo.
- **`git add filename`**
- **`git commit -m "commit message"`**

To clone a remote repo to your current directory:

- **`git clone url localDirectoryName`**

This will create the given local directory, containing a working copy of the files from the repo, and a .git directory (used to hold the staging area and your actual local repo)

SOME GIT COMMANDS

command	description
`git clone` ***url [dir]***	copy a Git repository so you can add to it
`git add` ***file***	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help` ***[command]***	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others: `init, reset, branch, checkout, merge, log, tag`	

VIEW HISTORY

- **git log -p** command shows all the commits along with the changes done in each commit.
- **Git show** allows you to check all the changes done on a Specific Commit ID.

BRANCHING

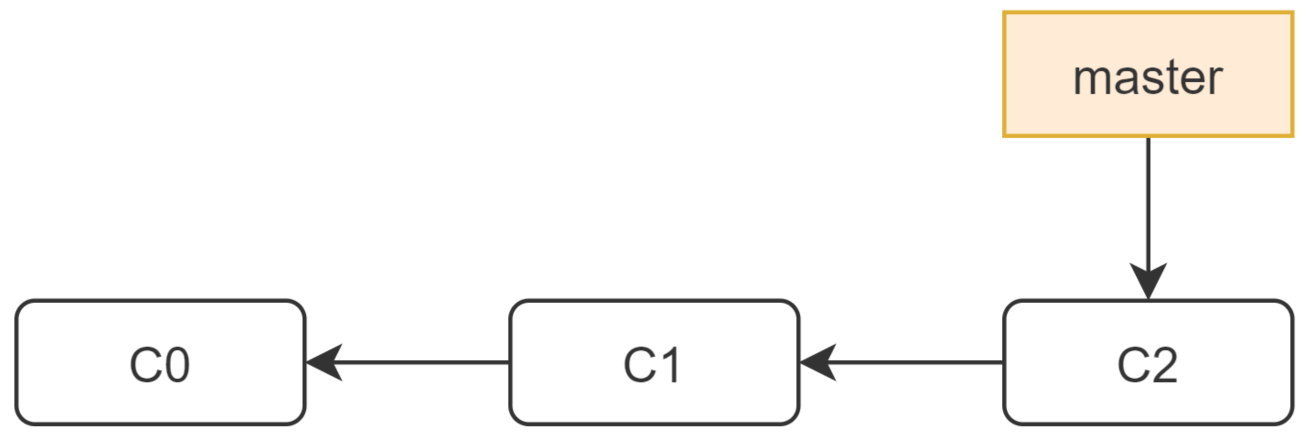
- Branches in Git are simply pointers to a specific commit -- nothing more.
- A branch essentially says "I want to include the work of this commit and all parent commits.“
- A branch allows us to branch off, develop a new feature, and then combine it back in

MERGING

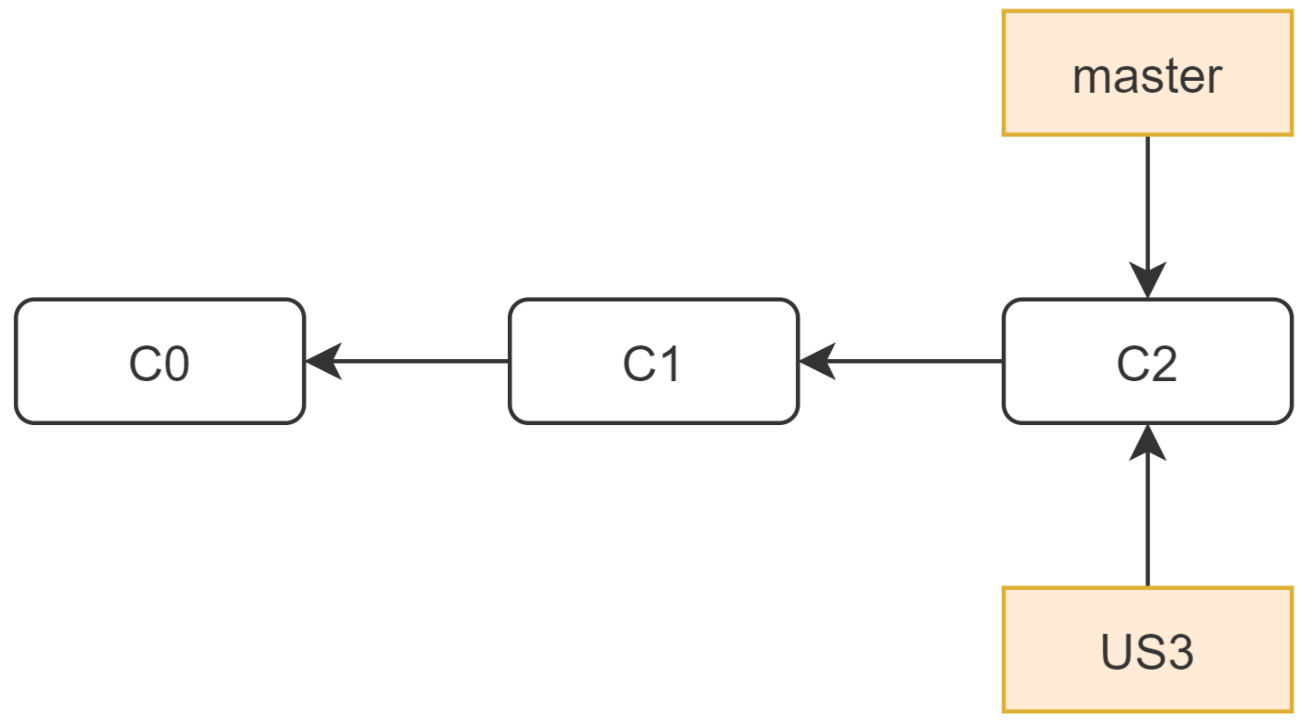
- Merge is combining the work from two different branches together:
    1. Branch off.
    2. Develop A new feature.
    3. Combine it back in.
- Merging in Git creates a special commit that has two unique parents.
- A commit with two parents essentially means "I want to include all the work from this parent over here and this one over here, and the set of all their parents."

GIT BRANCHES SCENARIO

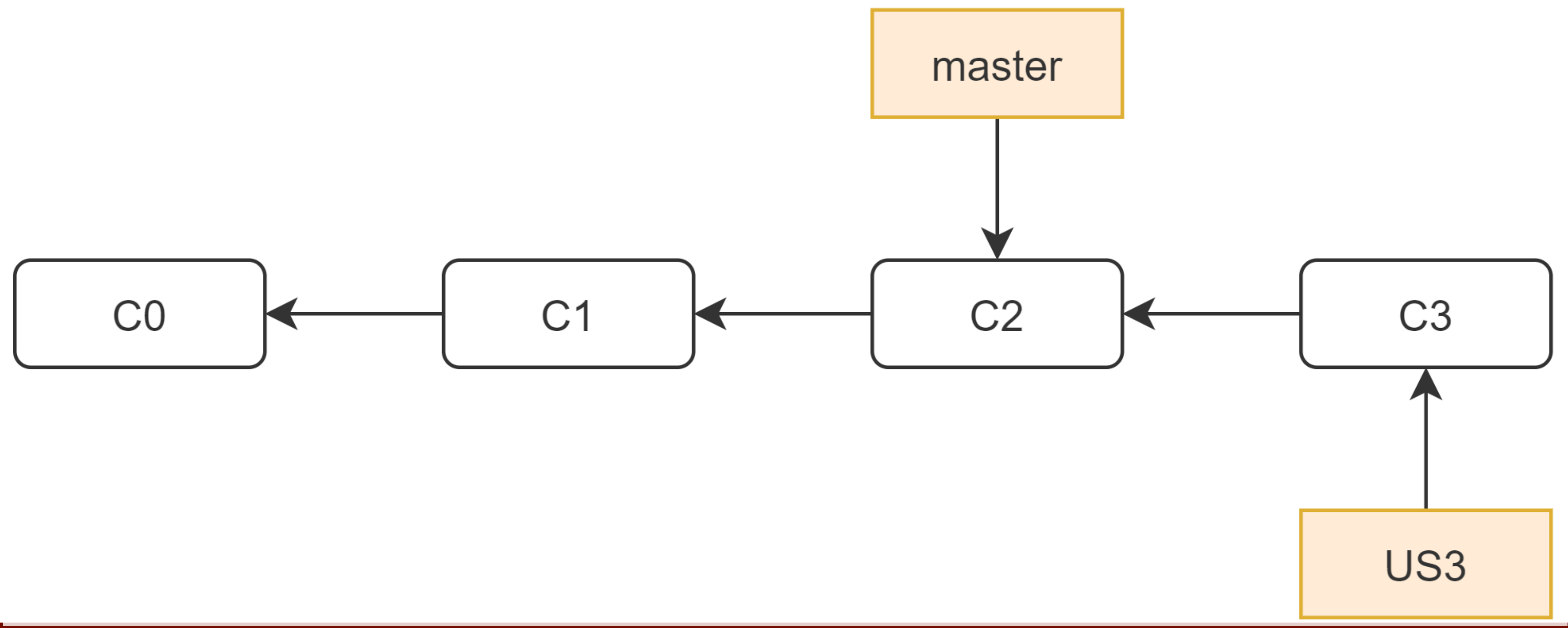
- Assume this is your current git log, you have made three commits on the master branch:



- Then you start working on a user story (US3), so you branch to complete the code for that story, you name it US3:

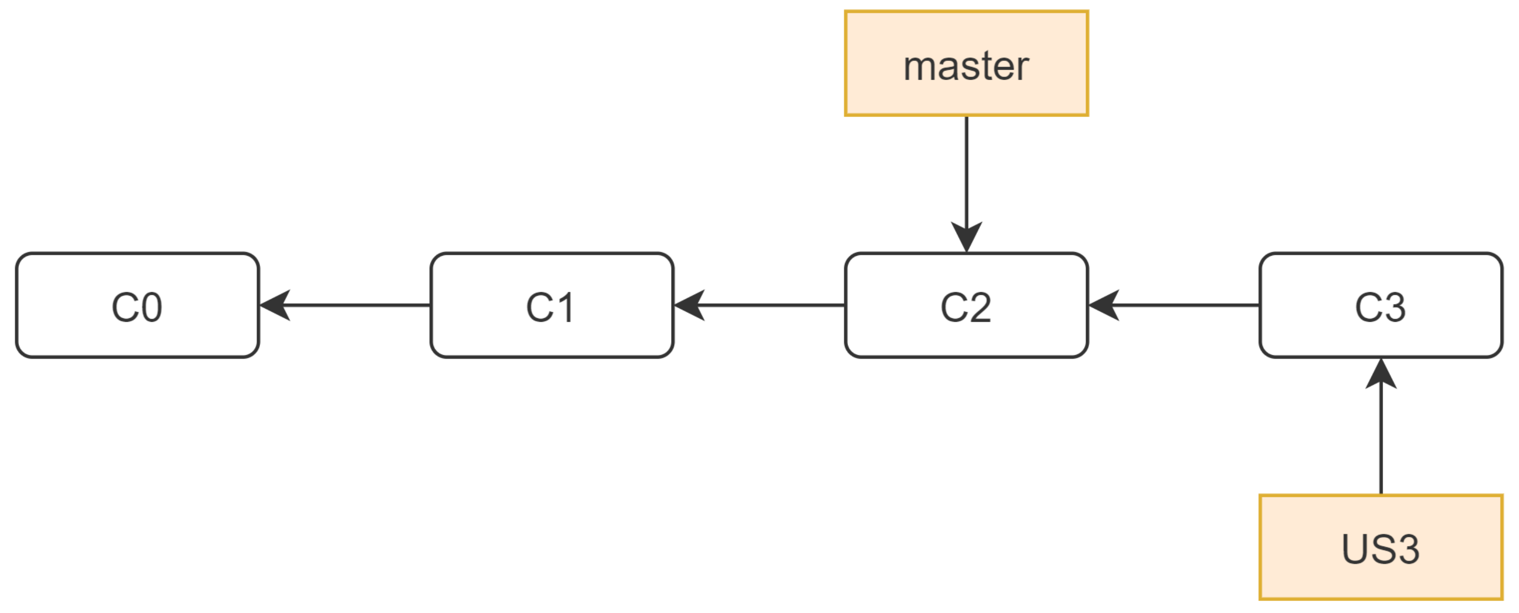


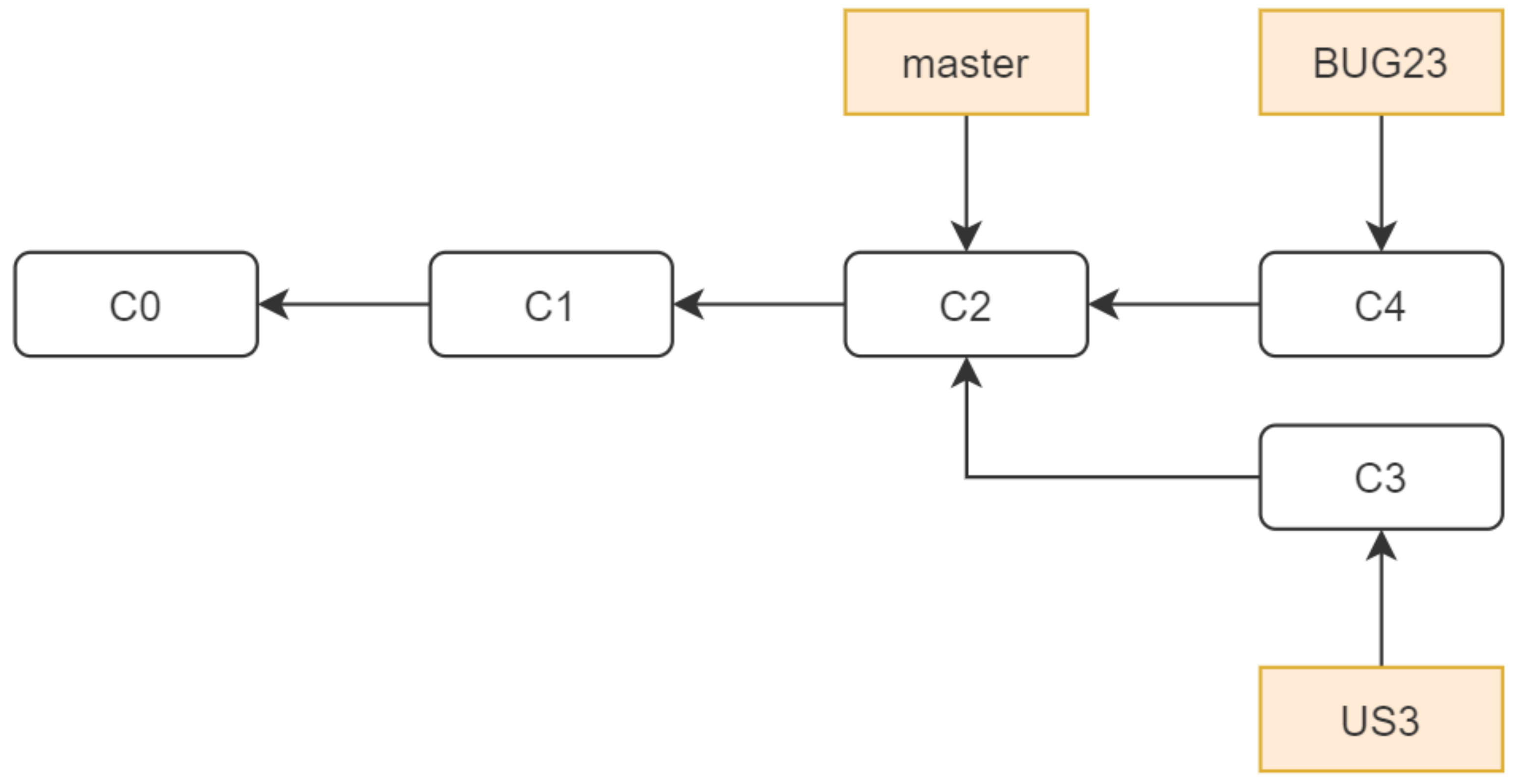
- After doing some work you do a commit on US3:



GIT BRANCHES SCENARIO

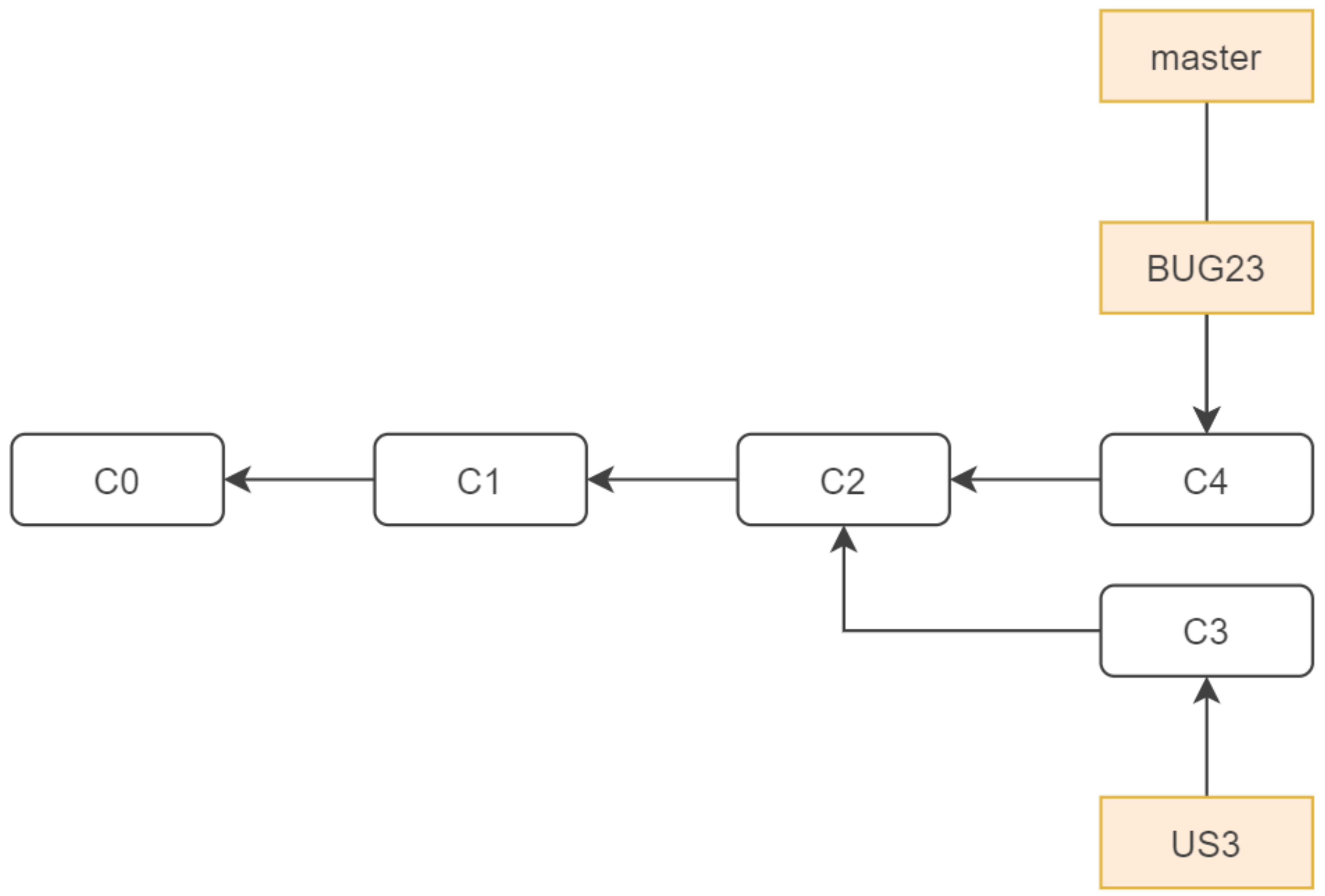
- Now you get a call that there is an urgent issue you need to fix immediately, it has been assigned the id of BUG23.
- You have two choices:
  - Do the repair on the current branch (US3).
  - Create a new branch based on master and apply the fix there.
- Option 1 is a bad choice in most circumstances because:
  - You may not be done with implementing US3.
  - You do not want one commit to have both US3 and BUG23 in case you ever need to rollback you will be undoing both.
- Option 2 is a better choice, rollback to master and create a new branch to add the code for repairing BUG23.





GIT BRANCHES SCENARIO

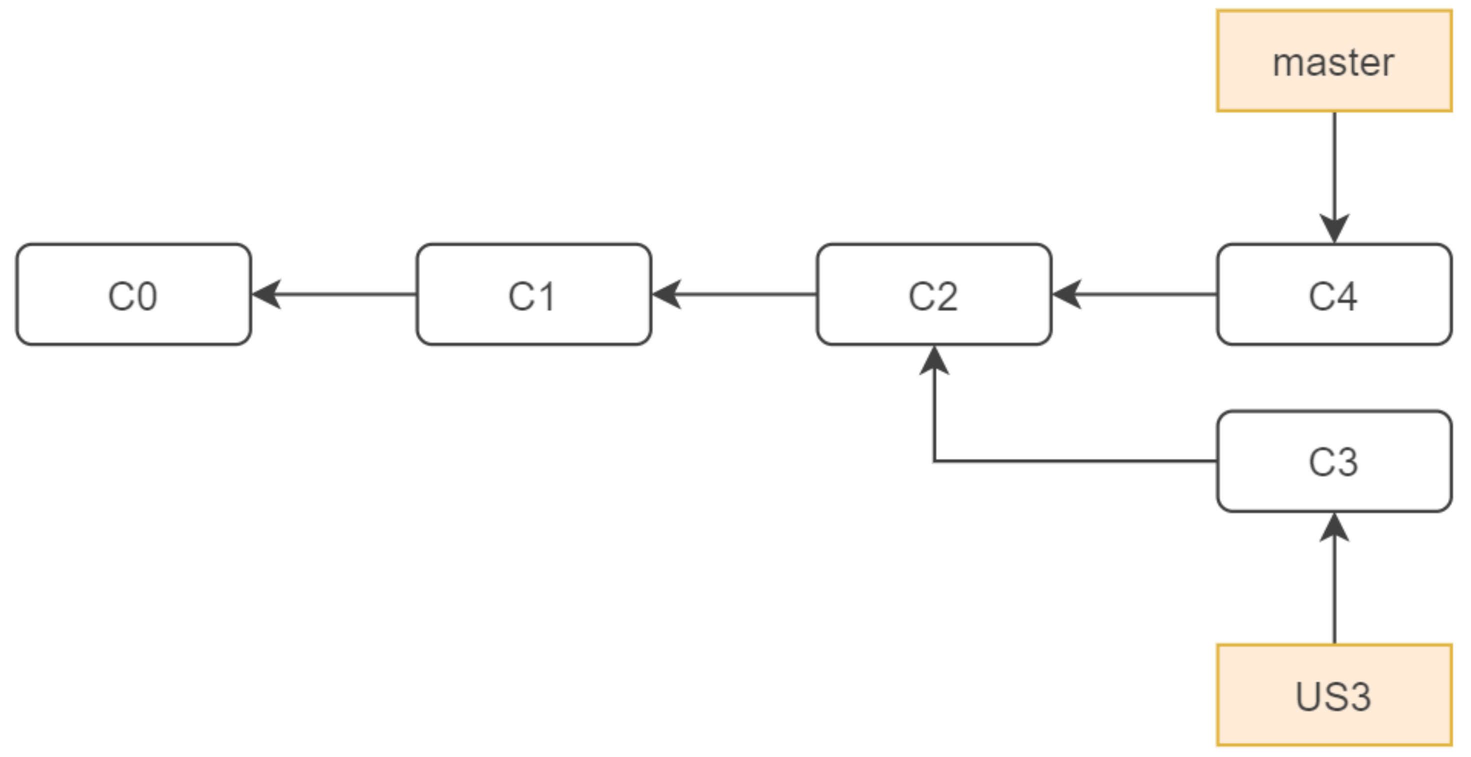
- When you are done applying the fix, and after testing it, you merge the BUG23 branch back into your master branch to deploy to production.



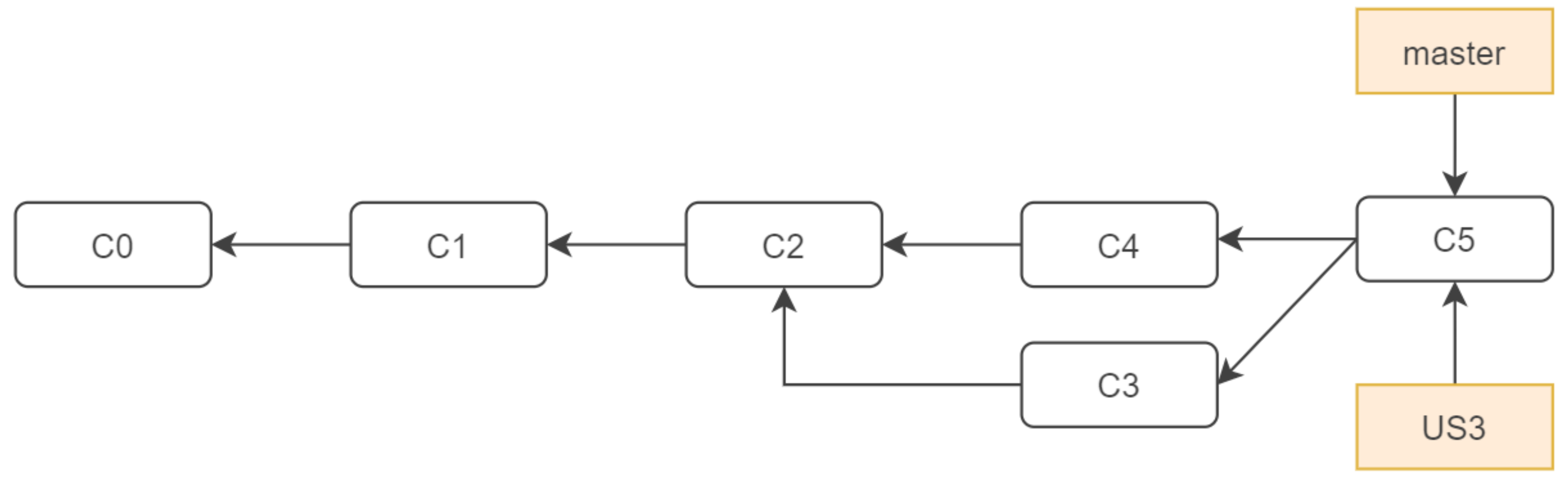
- Can we delete BUG23 branch?
- Now we can switch back to US3 and continue working on it

GIT BRANCHES SCENARIO

- After deleting BUG23:

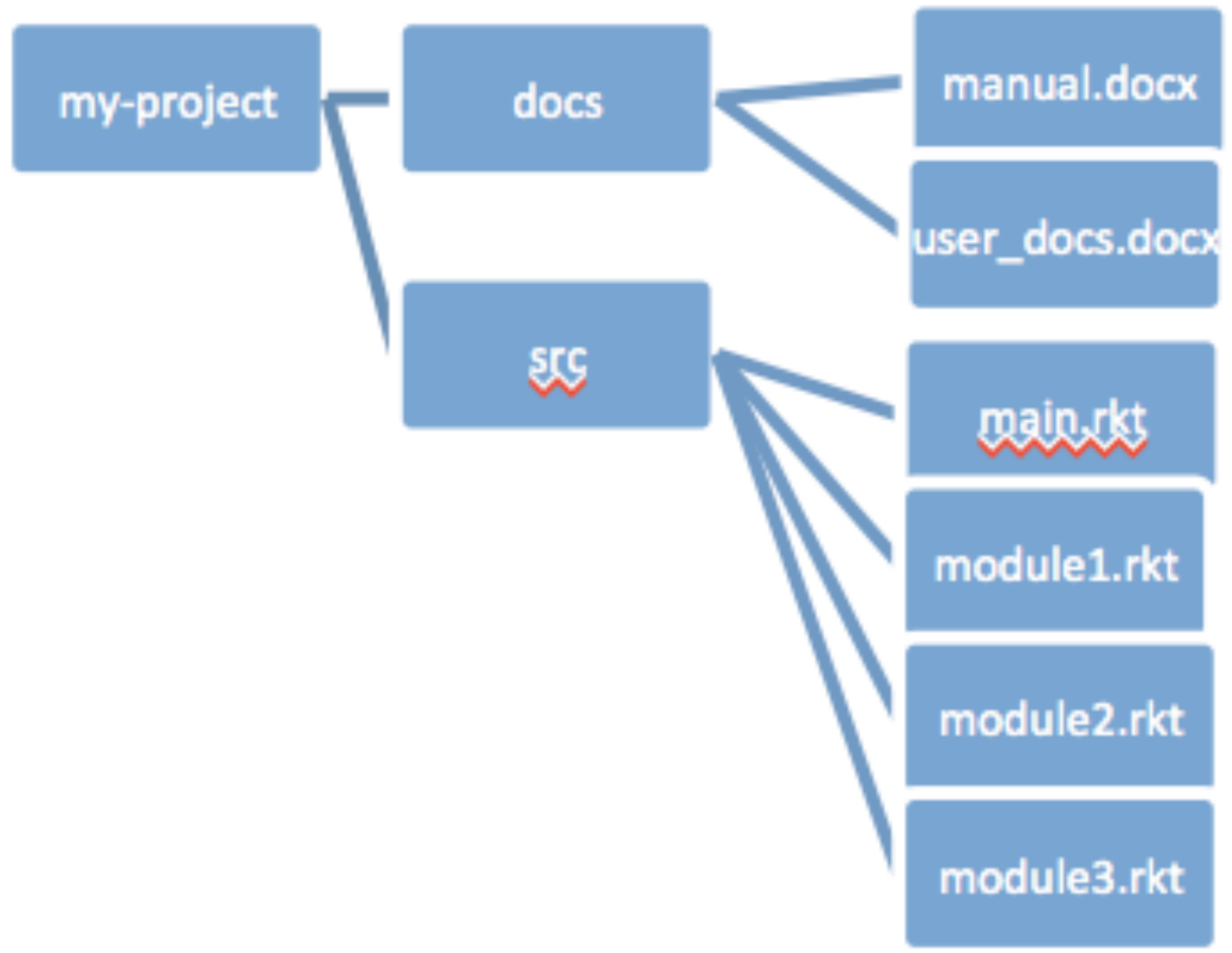


- After merging with US3 with master:



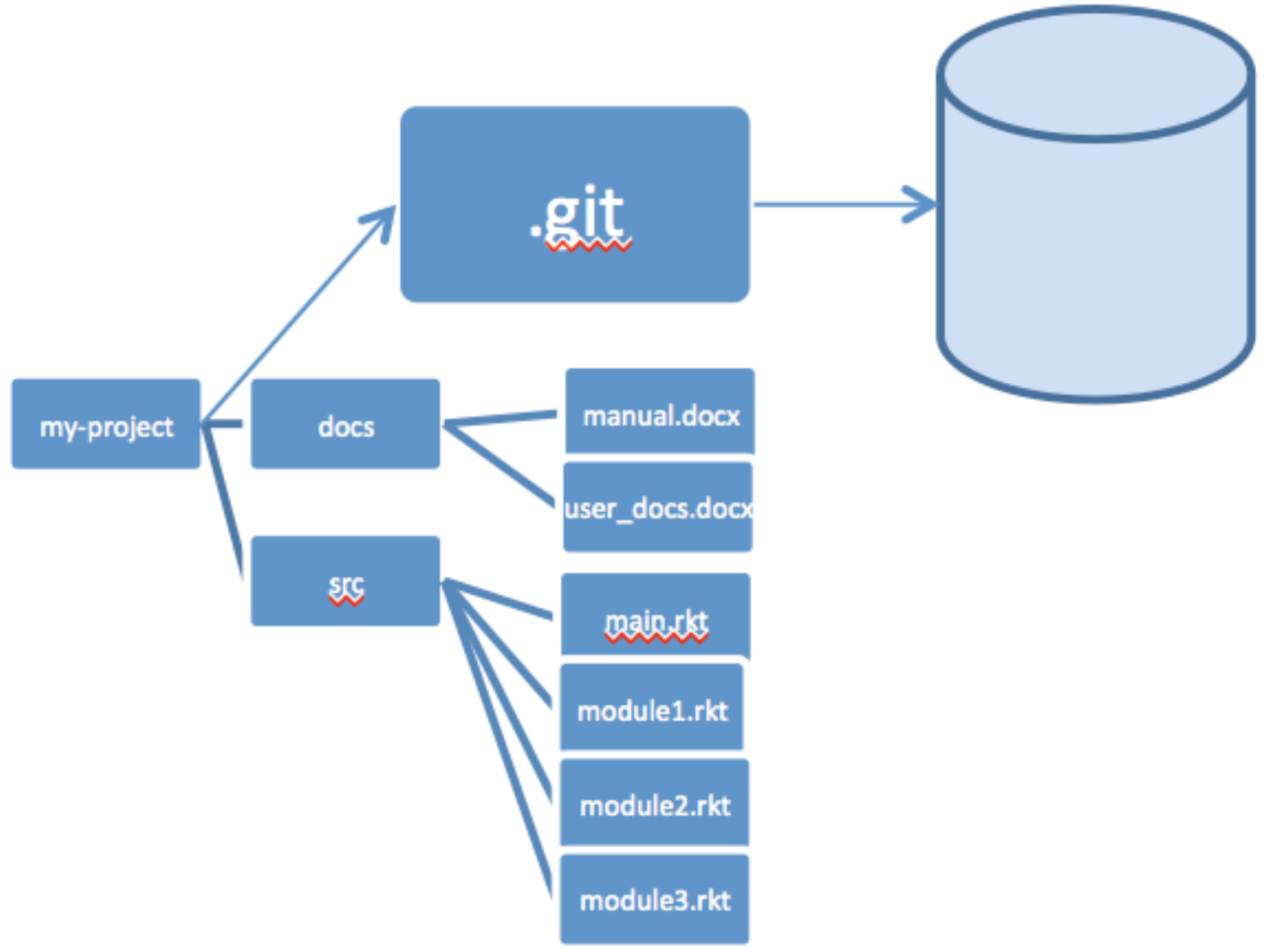
YOUR FILES

Here are your files, sitting in a directory called my-project:



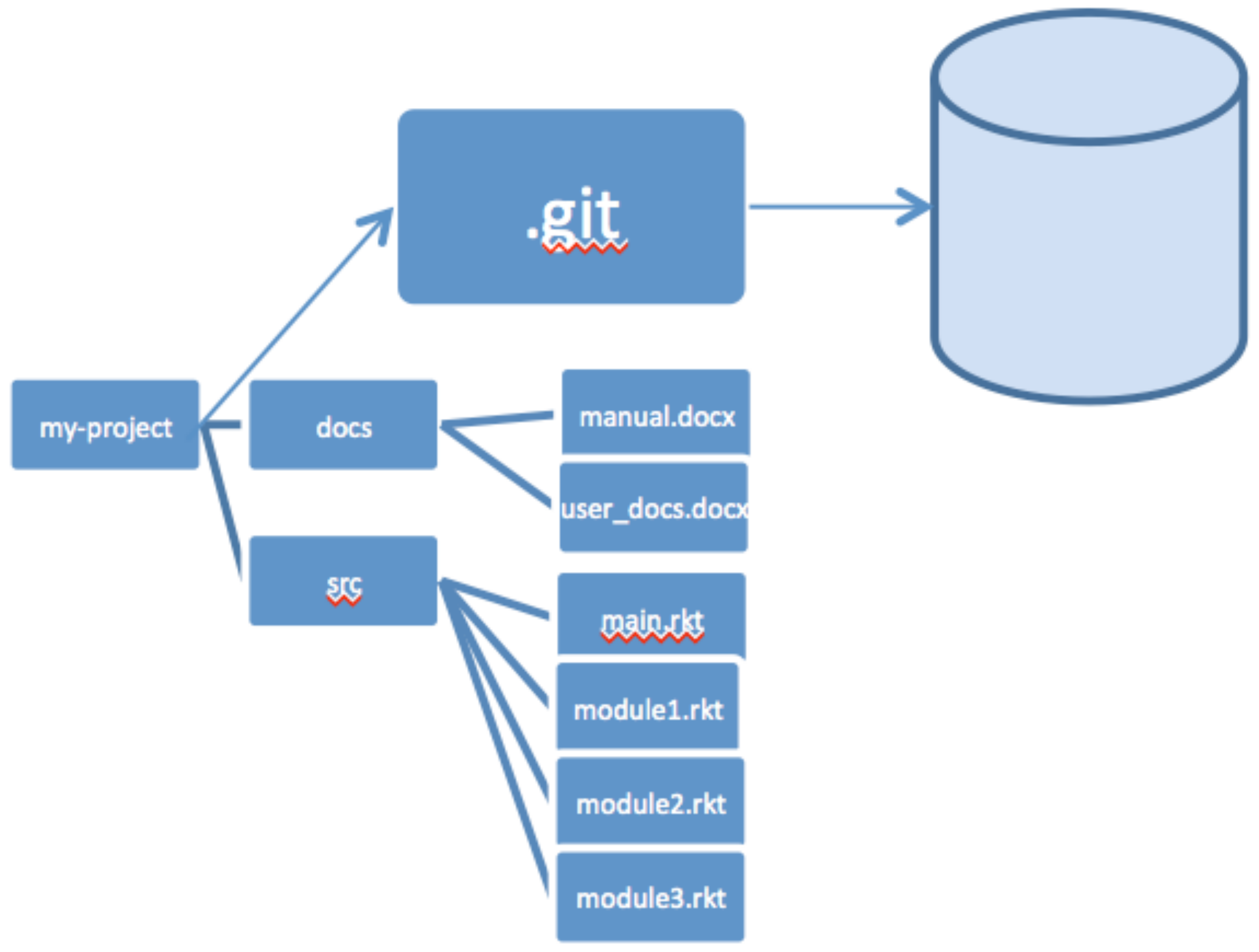
A GIT CLIENT

- When you have a git repository, you have an additional directory called **.git,** which points at a mini-filesystem.
- This file system keeps all your data, plus the bells and whistles that git needs to do its job.
- All this sits on your local machine.



A GIT CLIENT

- This mini-filesystem is highly optimized and very complicated. Don't try to read it directly.
- The job of the git client (either Github for Windows, Github for Mac, or a suite of command-line utilities) is to manage this for you.

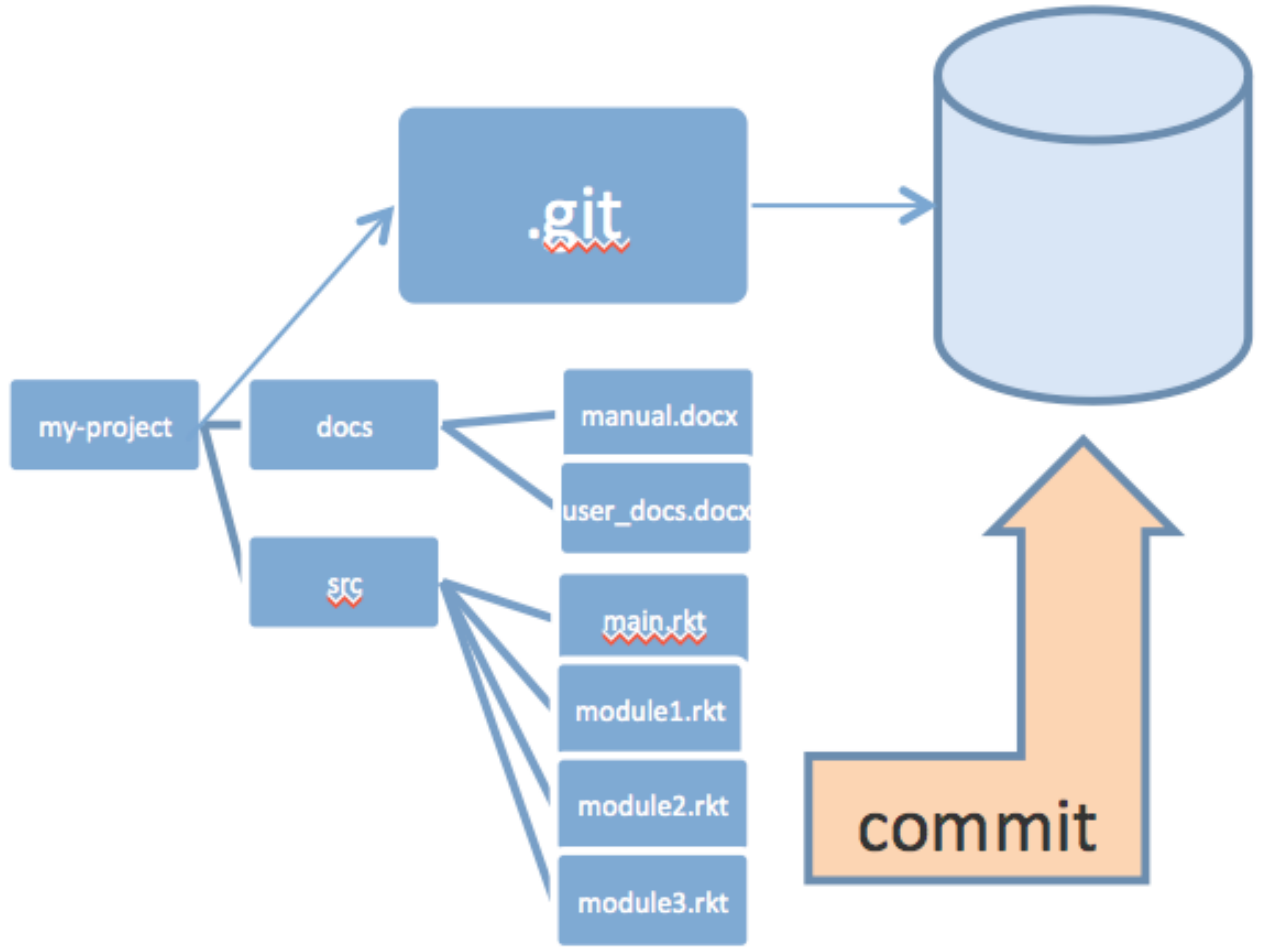


YOUR WORKFLOW (PART 1)

- You edit your **local files** directly.
- You can **edit**, **add** files, **delete** files, etc., using whatever tools you like.
- This **doesn’t change** the mini-filesystem, so now your mini-fs is behind.

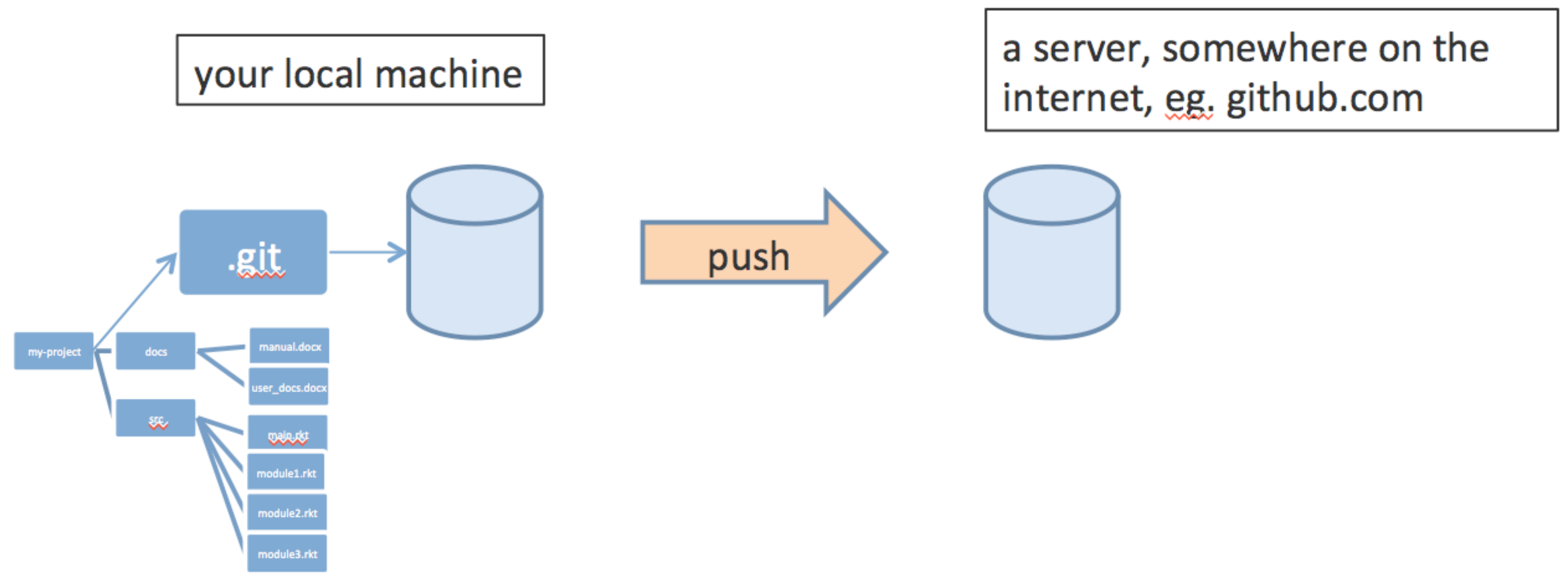
A COMMIT

- When you do a "**commit**", you record all your local changes into the mini-fs.
- The mini-fs is "append-only". Nothing is ever over-written there, so everything you ever commit can be recovered.



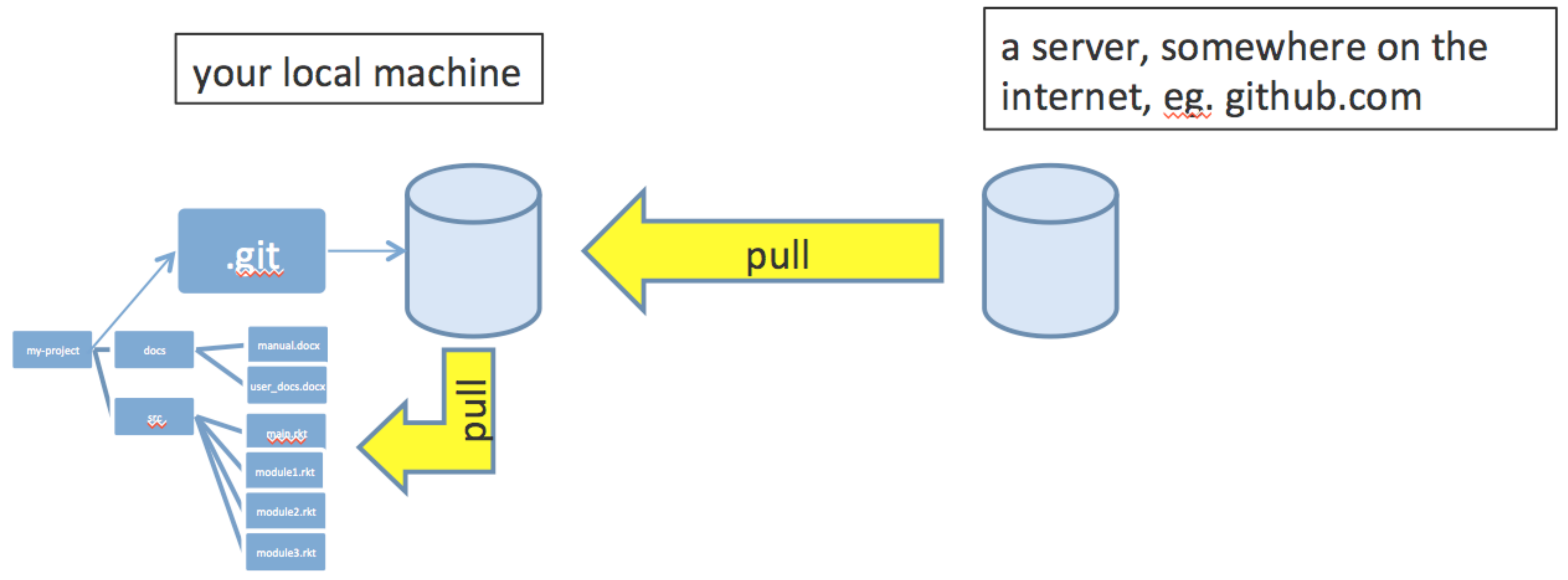
SYNCHRONIZING WITH THE SERVER (1)

- At the end of each work session, you need to **save** your changes on the server. This is called a "**push**".
- Now all your data is backed up.
- You can retrieve it, on your machine or some other machine.

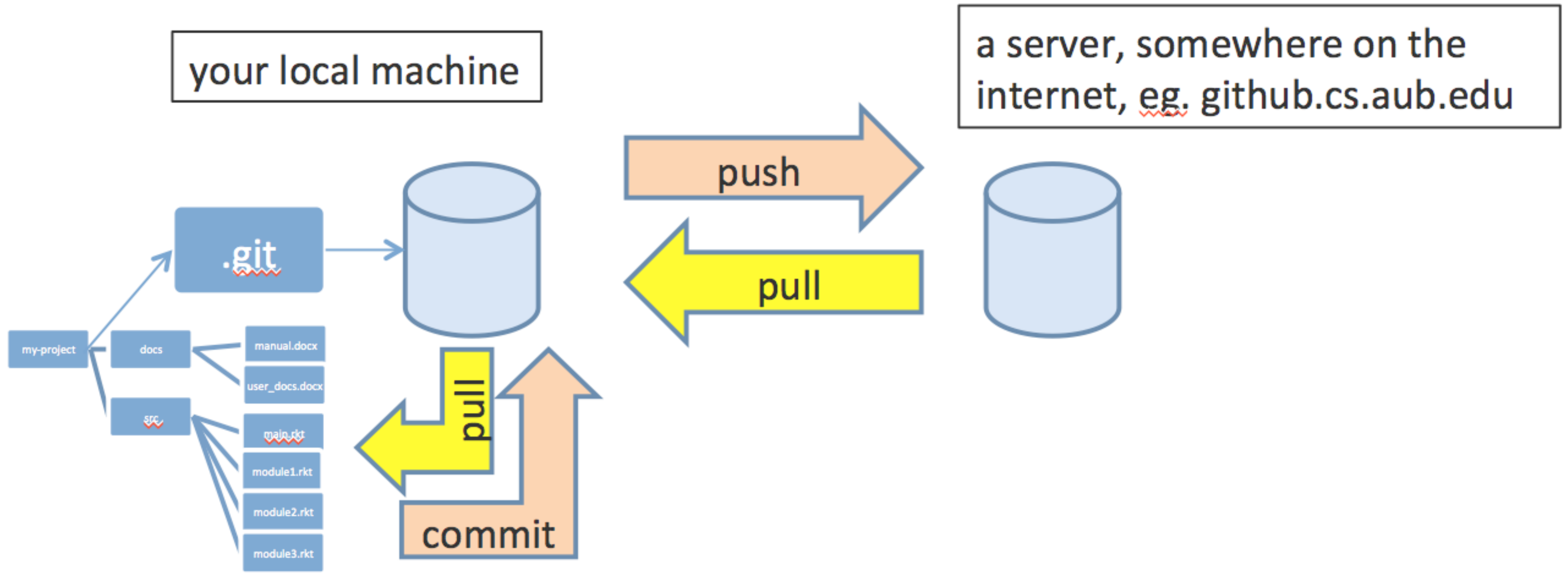


SYNCHRONIZING WITH THE SERVER (2)

- To **retrieve** your data from the server, you do a "**pull**". A "pull" takes the data from the server and puts it both in your local mini-fs and in your ordinary files.
- If your local file has changed, git will merge the changes if possible. If it can't figure out how to the merge, you will get an error message.

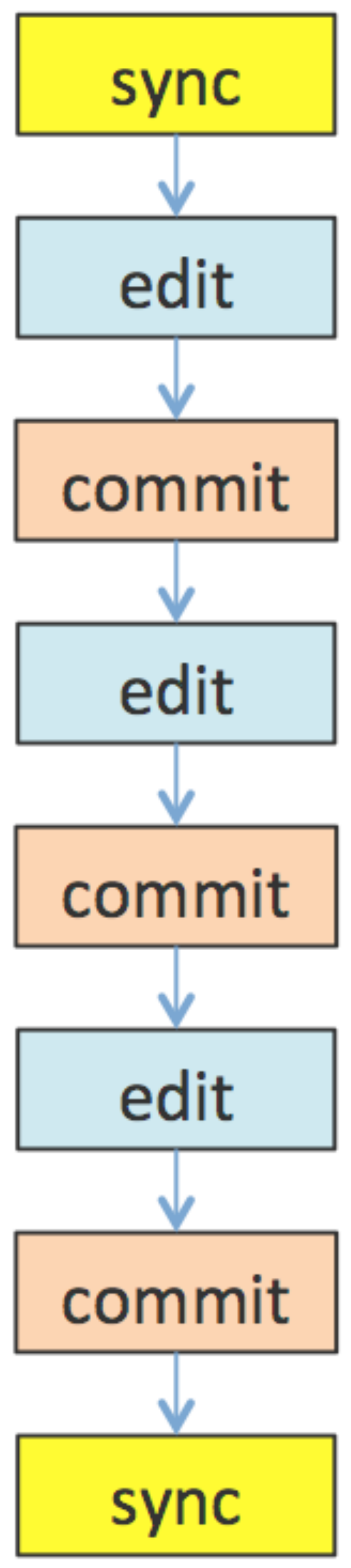


THE WHOLE PICTURE

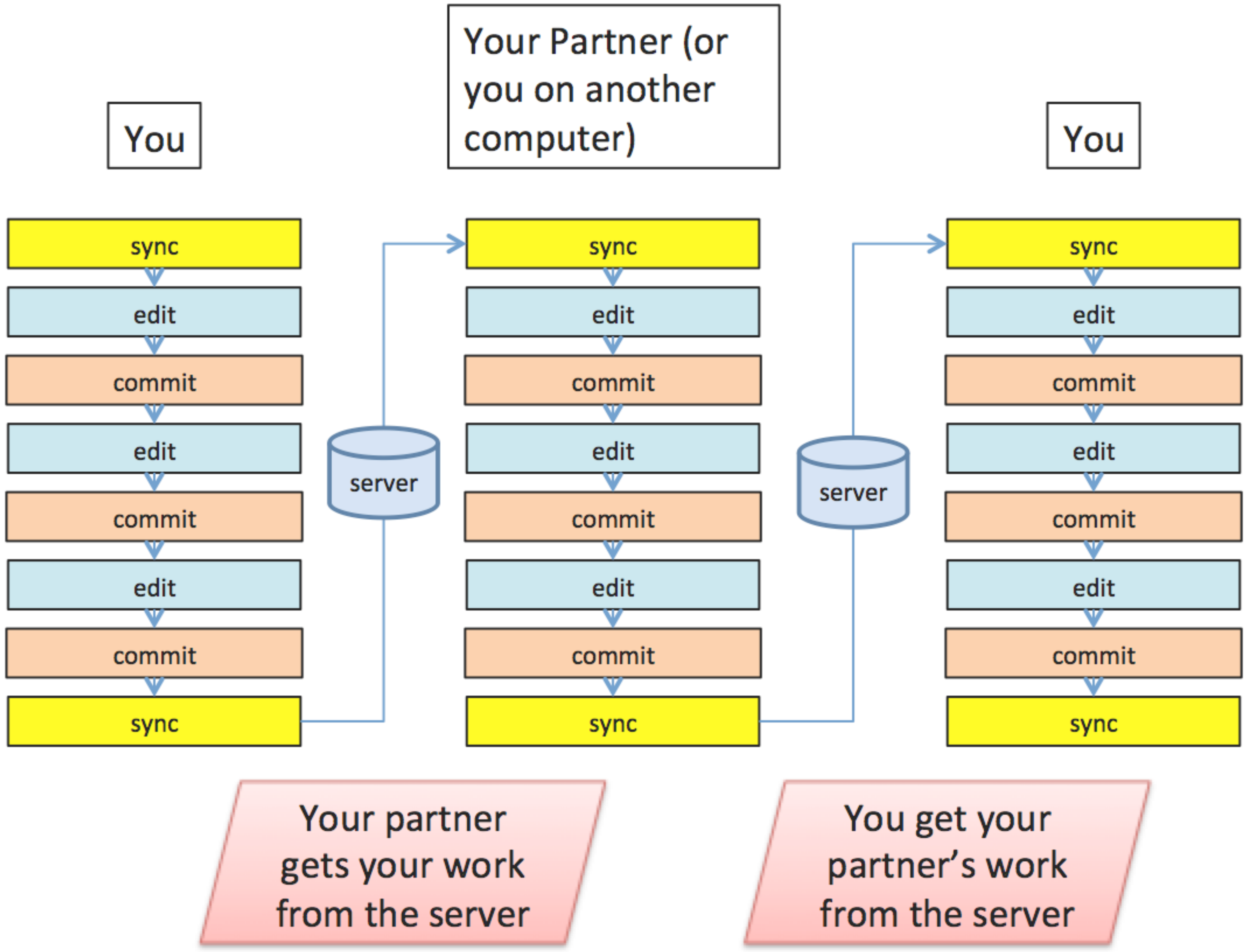


YOUR WORKFLOW (PART 2)

- Best practice: **commit** your work whenever you've gotten one part of your problem working, or before trying something that might fail.
- If your new stuff is messed up, you can always "**revert**" to your last good commit.



YOUR WORKFLOW WITH A PARTNER



RESOURCES

- Additional resources used in this slide deck:
- Mahmoud Bdeir, Advanced Programming Skills Course.
- Slides from http://www.cs.washington.edu/403/ Adapted from slides created by Ruth Anderson for CSE 390a, with images from http://git-scm.com/book/en/
- Mitchell Wand, 2012-2014, CS 5010 Program Design Paradigms "Bootcamp" Lecture 0.5