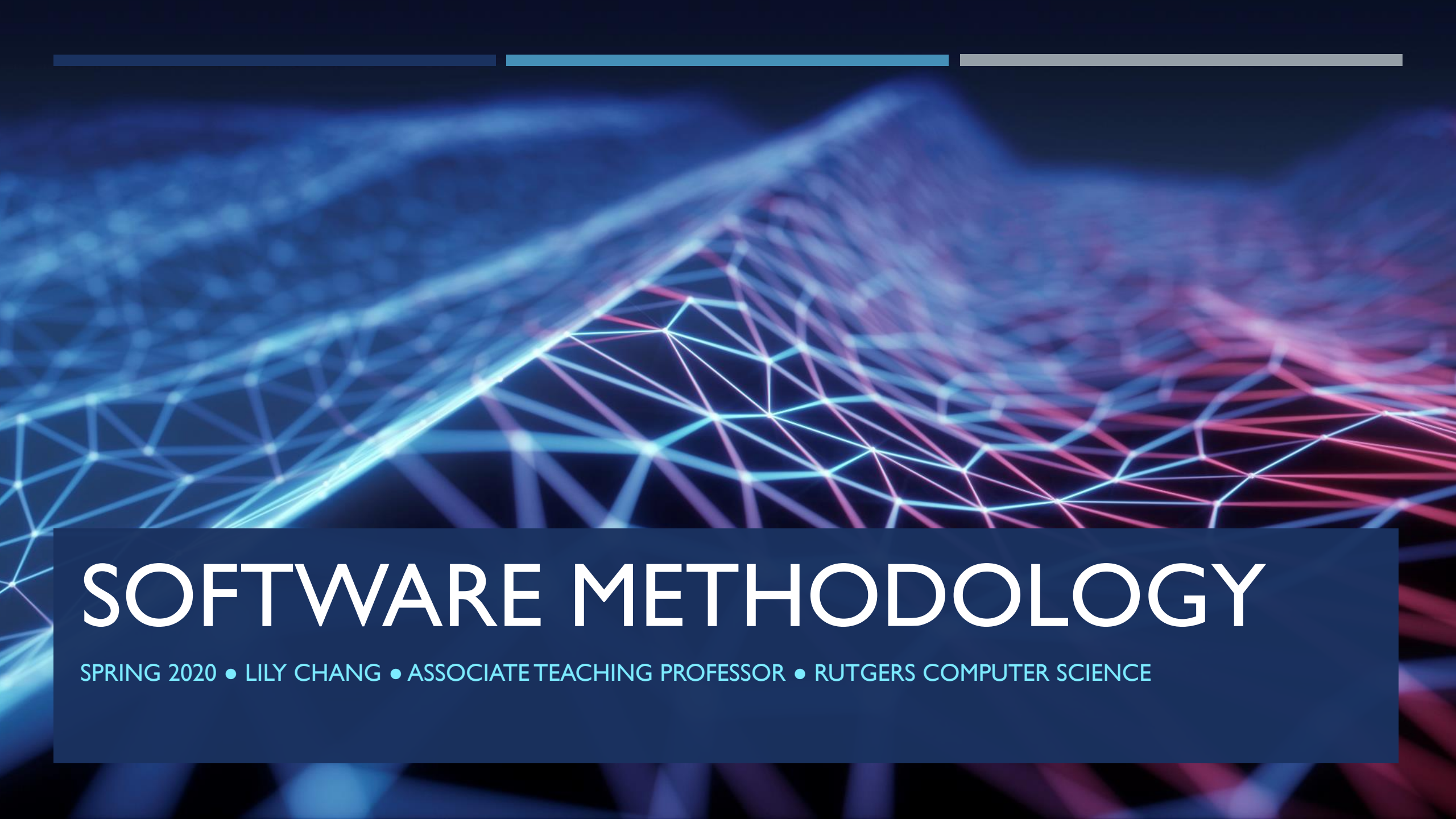


SOFTWARE METHODOLOGY

SPRING 2020 • LILY CHANG • ASSOCIATE TEACHING PROFESSOR • RUTGERS COMPUTER SCIENCE

VERSION CONTROL SYSTEMS

MANAGING CHANGES IN SOFTWARE DEVELOPMENT PROCESS

- Delivery of software is not the end of the story!!
  - Requirements change!!
  - Environment where software run changes!!
  - New technologies in software and hardware
- Software must evolve to remain useful
  - You invest a good amount of money to develop the software product
  - Bug fixes, add or update functionalities, new regulations, adapt to new environment, refactoring, …
- Configuration management and Version control
  - Different builds, software packages, patches, …

VERSION CONTROL (SOURCE MANAGEMENT)

- Version Control is about the management of multiple versions of a project. To manage a version, each change (addition, edition, or removal) to the files in a project is tracked.
- You probably have the experience of tracking different versions of your own programming assignments, you might name the assignments, project1_1, project1_2, …etc., with a ”version number”, or program1_old, progam1_new, program1_fixed… etc., with a suffix.
  - It is very easy to forget which file is which and what has changed between them.

WHY VERSION CONTROL SYSTEMS (VCS)

- Software product development is a collective effort
  - Teamwork
  - Collaboration
  - When several team members are contributing to a project, tracking changes becomes a nightmare
- Software vendors provide different options (features) to a software product
  - Different builds to meet the the needs of clients
- Ensure the quality of software products
- …

WHAT ARE THE CHOICES?

- There are many flavors of Version Control Systems, each with their own advantages and shortcomings.
- A VCS can be
  - Local,
  - Centralized, or
  - Distributed

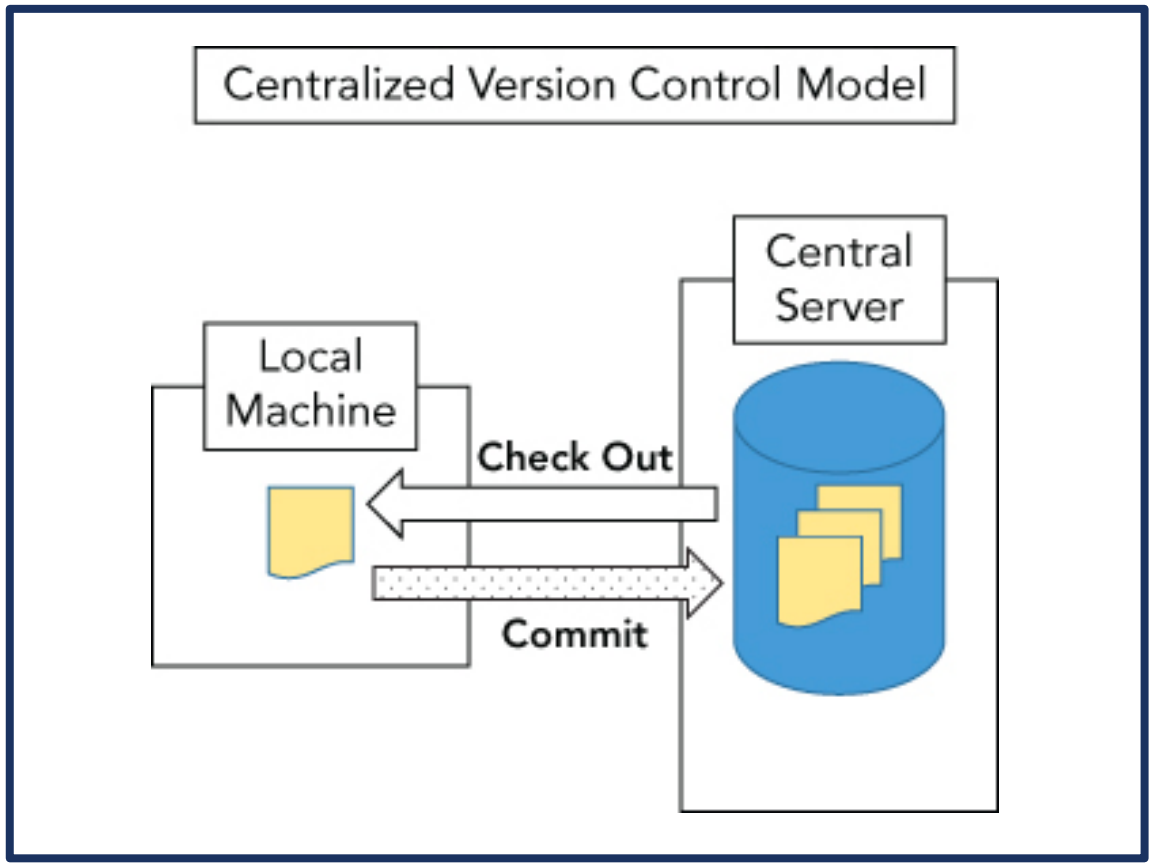
IN THE EARLY DAYS – LOCAL VERSION CONTROL SYSTEMS

- First VCSs created to manage source code. They worked by tracking the changes made to files in a single database that was stored locally.
- All the changes were kept in a single computer and if there were problems, all the work were lost .This also means that working with a team was out of the question.
- One of the most popular local VCSs was Source Code Control System or SCCS, which was free but closed source. Developed by AT&T, it was wildly used in the 1970s
- Revision Control System or RCS was released in 1982 and became more popular than SCCS because it was Open Source, cross-platform, and much more effective. It is currently maintained by the GNU Project.
- Local VCSs only worked on a file at a time; there was no way to track an entire project

NOWADAYS – CENTRALIZED OR DISTRIBUTED VCS

- VCSs currently in use can be broadly classified as either centralized or distributed.
- Centralized VCSs
  - Concurrent Versions System (CVS)
  - **Subversion**
- Distributed VCSs
  - Git
  - Mercurial
  - Bazaar

CENTRALIZE VERSION CONTROL SYSTEM



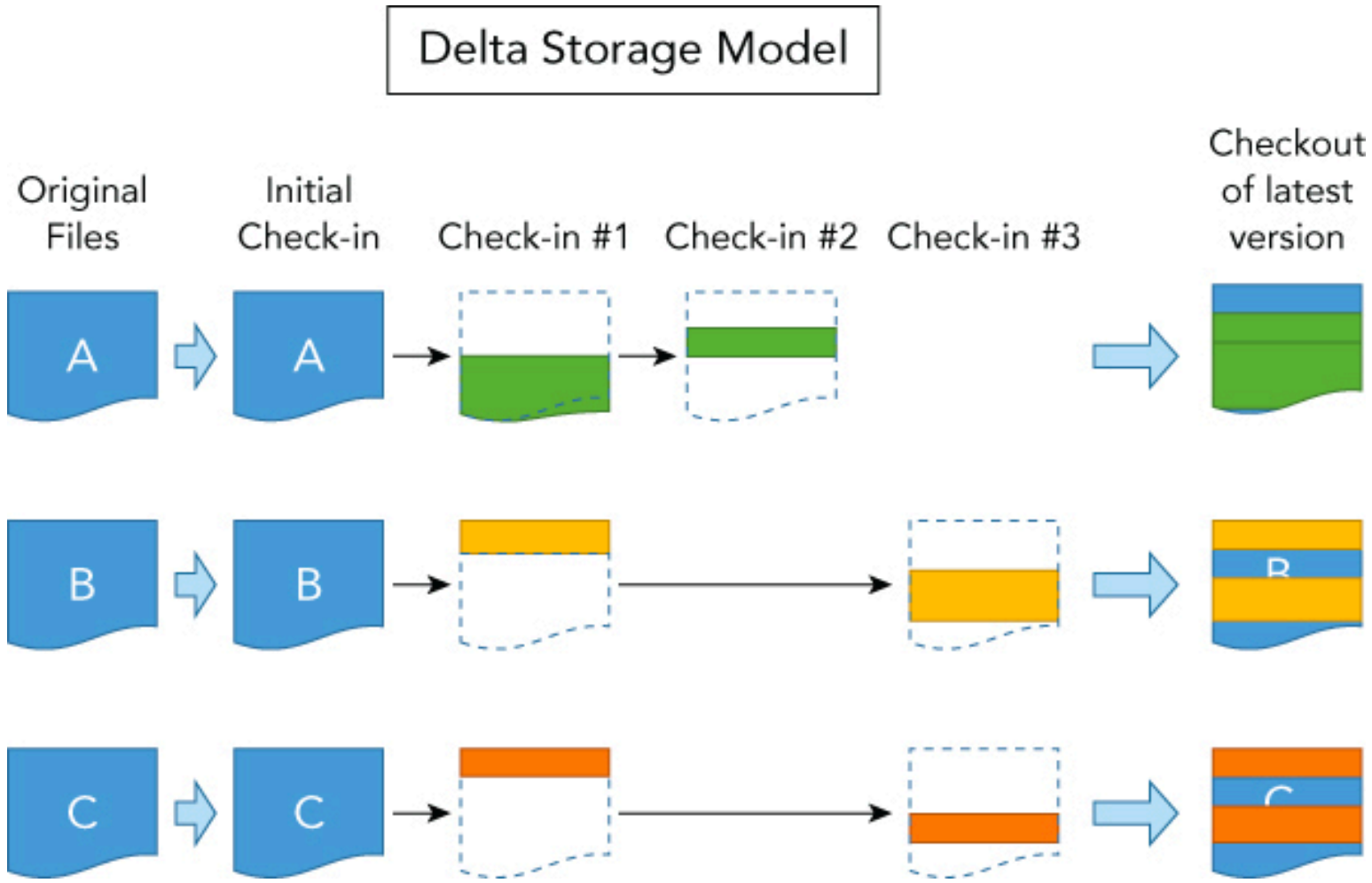
- When users want to work with a file in one of these repositories, they connect to the server via a client, and retrieve the files and the versions they want to work with.
- Users make changes locally and connect to the server again, and send the update back to the server
- The differences from the previous version are determined and stored in the repository as updates.
- In this type of model, users are dependent on the central server. If, for some reason, users cannot connect to the server, they cannot do any source management operations.

CENTRALIZE VERSION CONTROL SYSTEMS

- Content is managed on a **file-by-file basis**. That is, each file is managed as an independent entity in the repository.
- When a set of files is added to a repository for the first time, each file is stored as a separate object in the repository, with its complete contents.
- The next time any changes to any of these files are checked in, the system **computes the differences** between the new version and the previous version for each file.
  - It constructs a delta, or patch set, **for each file** from the differences. It then stores that **delta** as the file's next revision.

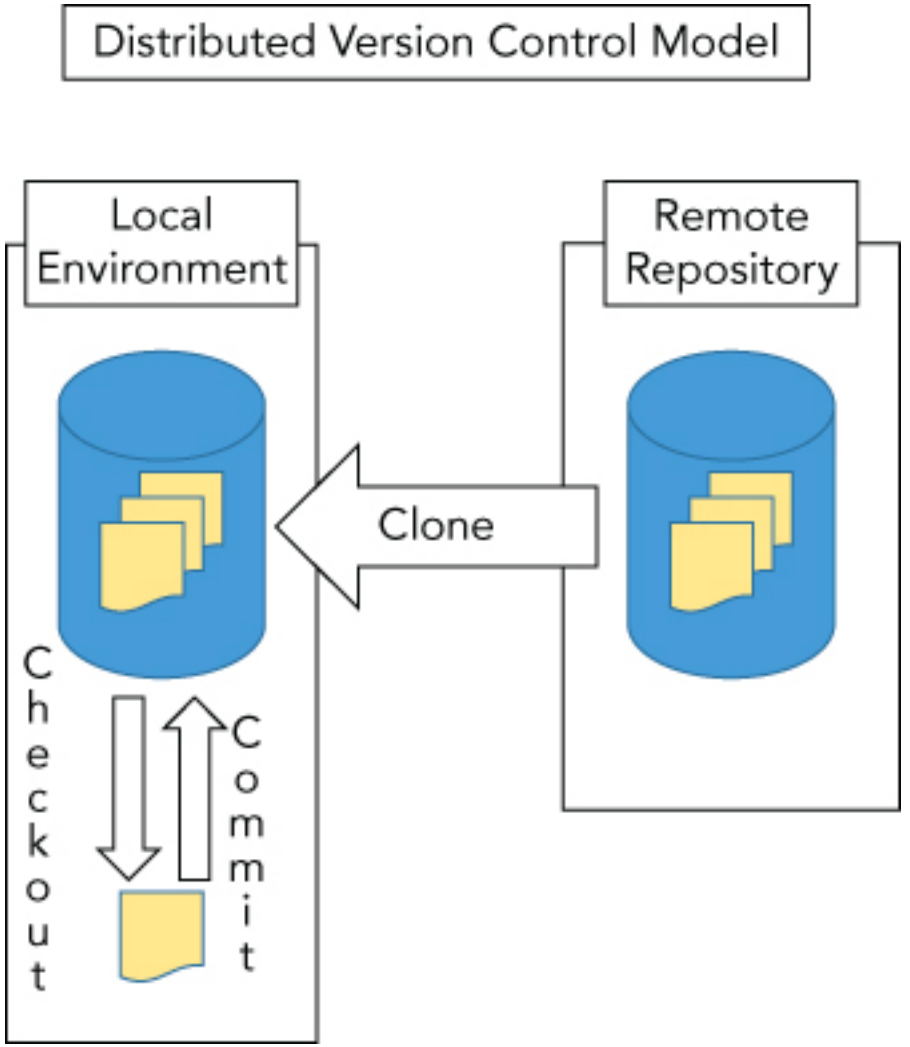
CENTRALIZE VERSION CONTROL SYSTEM

- In order to get the most current version of a file from the system when the client requests it, the system starts with the original version of the file and then applies each delta in turn to arrive at the desired version.

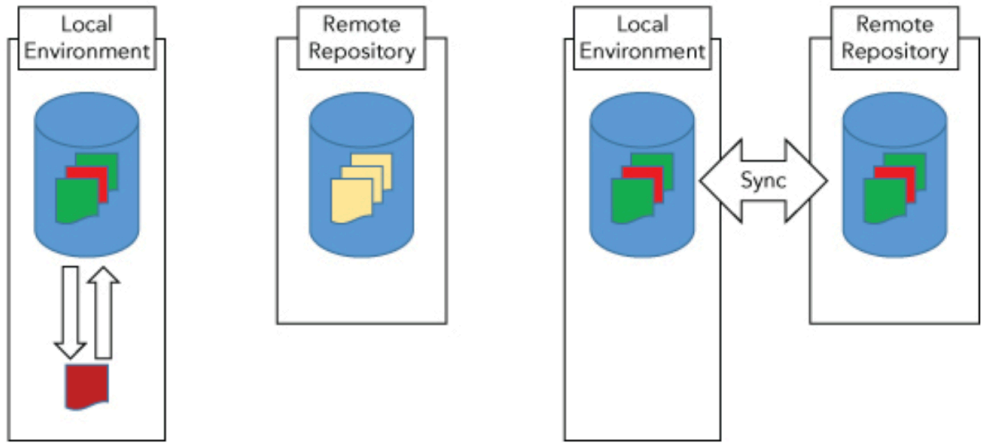


DISTRIBUTED VERSION CONTROL SYSTEMS

- A server holds the shared repositories (Remote Repository)
- Users get a copy of the entire repository (Local Repository). The copy comes from the server side and has all content (including history) up to the point in time when the copy is created.

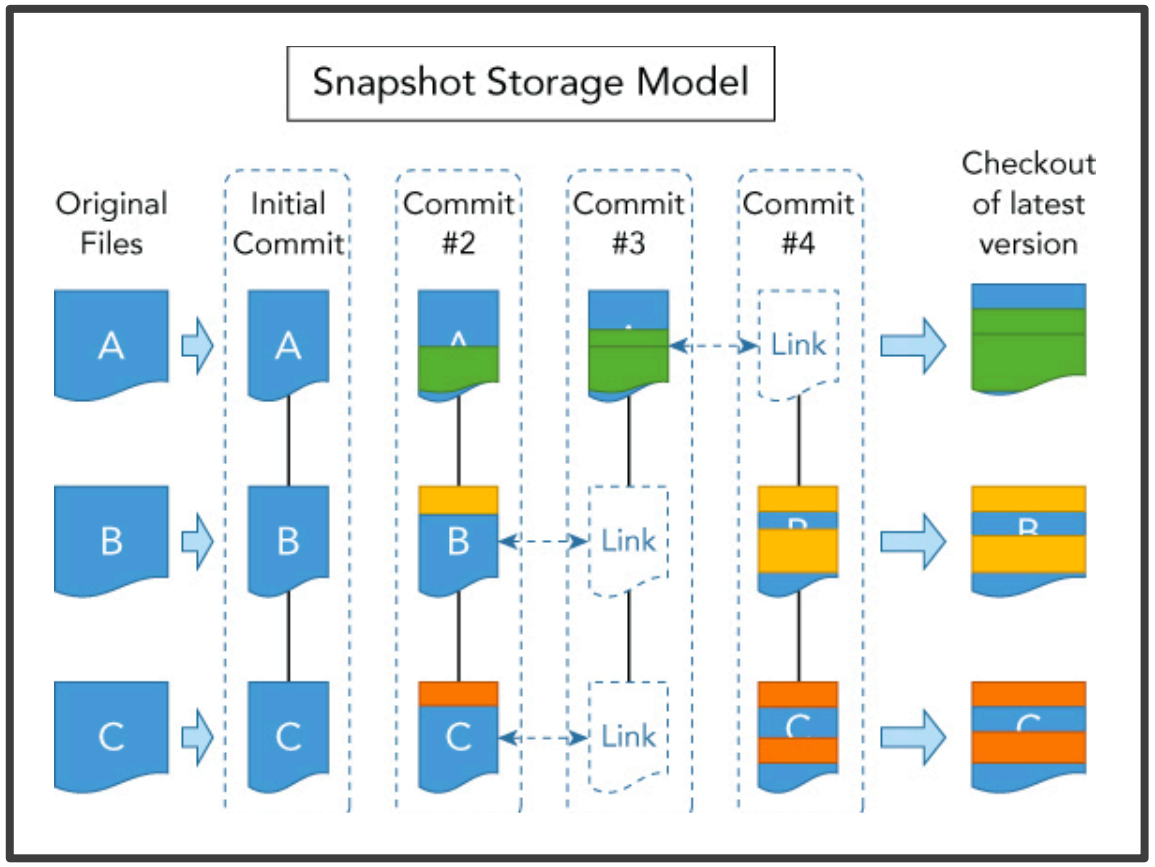


DISTRIBUTED VERSION CONTROL SYSTEMS



- Users are performing the source management operations against a local copy of the server-side (remote) repository instead of making them against the actual server-side repository.
- Until users need to push the changes back to the remote, the connection between the local and the remote side is activated and the repositories are synchronized
- Because users do not have to be connected to the remote to do their source management operations, they **can work disconnected from the remote.**

DISTRIBUTED VCS – SNAPSHOT STORAGE MODEL IN GIT



- Tracks revisions at the level of a directory tree.
- A commit represents a snapshot of part or all of the directory tree in the workspace, at that point in time.
  - In each of these snapshots, Git is capturing the contents of all of the involved files and directories as they are in your workspace at that point in time. It's recording the full content, not computing deltas.

GIT – AN EXAMPLE ON DISTRIBUTED VCS

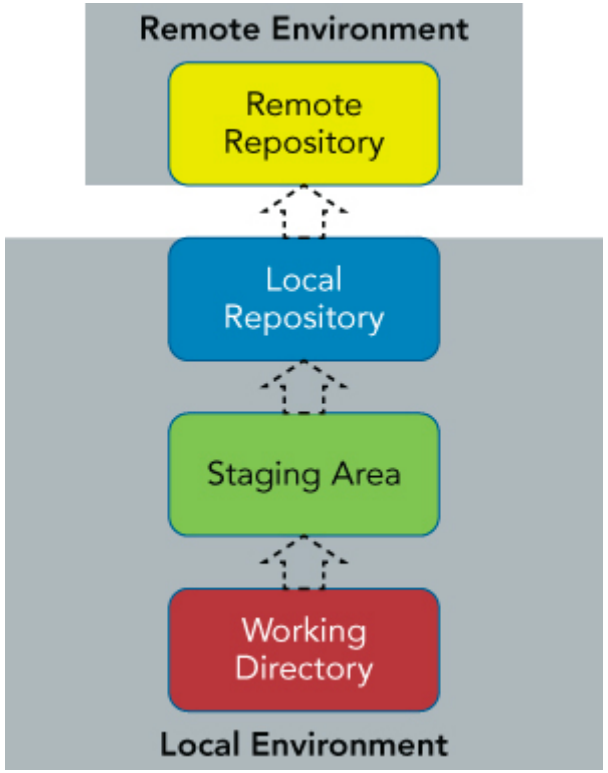
- In the early 2000s, a proprietary distributed source control system called BitKeeper
- July 2005, Git was released as a toolkit that could have other systems implemented on top of it
- Git was given its name by its creator, Linus Torvalds, who jokingly stated that he named all his projects after himself
- Git has grown to become an industry-standard tool. It is a key component of many **continuous integration/continuous delivery** pipelines

CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY

Software development and release process

Levels in Git

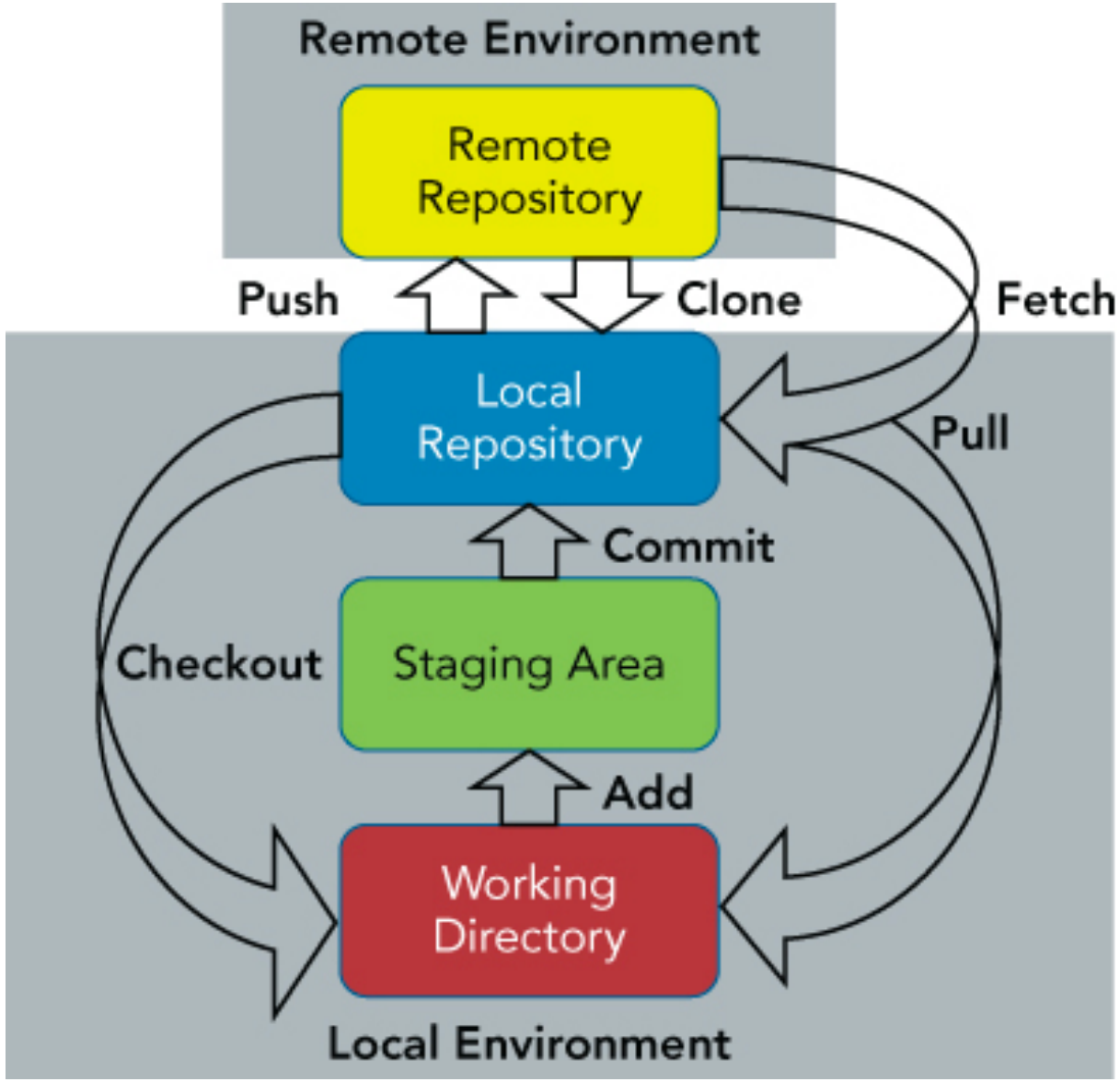
THE LEVELS IN GIT



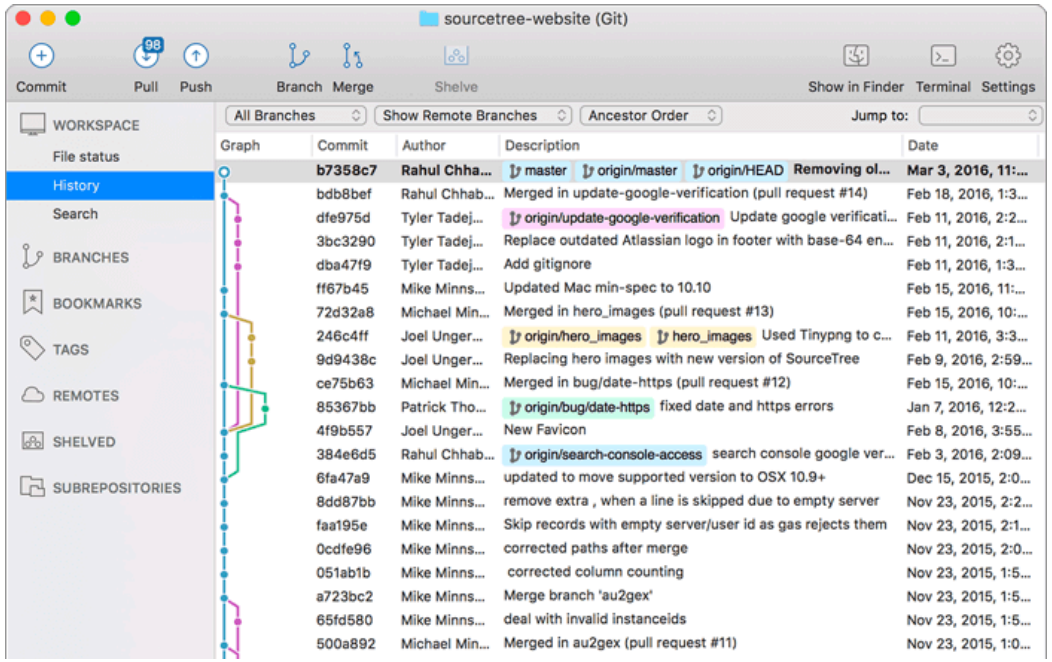
- Starting at the bottom is the working directory where content is created, edited, deleted
- The staging area serves as a holding area to accumulate and stage changes from the working directory before they are committed into the next level—the local repository
- The local repository is the actual source repository where content that Git manages is stored. Once content is committed to the local repository, it becomes a version in the repository and can be retrieved later.
- The combination of the working directory, staging area, and local repository make up your local environment, where users create and update content and get it in the form they want before making it available or visible to others, in the remote repository
- The remote repository is a separate Git repository intended to collect and host content pushed to it from one or more local repositories. its main purpose is to be a place to share and access content from multiple users.

GIT CORE COMMANDS

From	To	Command	Notes
Working Directory	Staging Area	Add	Stages local changes
Staging Area	Local Repository	Commit	Commits only content in staging area
Local Repository	Remote Repository	Push	Syncs content at time of push
Local Repository	Working Directory	Checkout	Switches current branch
Remote Repository	Local Environment	Clone	Creates local repository and working directory
Remote Repository	Local Repository	Fetch	Updates references for remote branches
Remote Repository	Local Repository and Working Directory	Pull	Fetches and merges to local branch and working directory


Remote Environment
Remote Repository
Push
Clone
Fetch
Local Repository
Pull
Commit
Checkout
Staging Area
Add
Working Directory
Local Environment

GIT GUI CLIENTS



Sourcetree

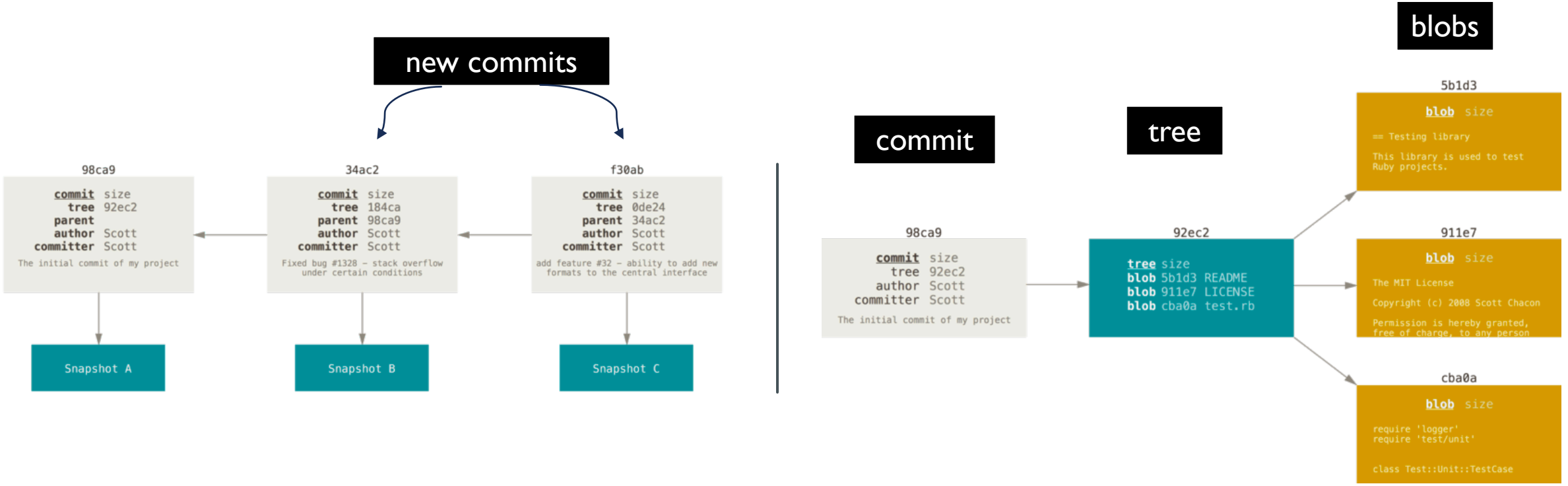
- Provide GUI interfaces for working with repositories and may support GUI-based conventions such as drag-and-drop to move content between levels.
- They often provide graphical tools for labor-intensive operations such as merging.
- Examples include SourceTree, SmartGit, TortoiseGit, and Git Extensions.

GIT – PROS AND CONS

- Disconnected Development
- Strong support for Branching
  - It is rare these days for source management users to only be concerned with one release of content.
  - When software products are managed via a continuous delivery process, in a user's local environment, there are typically multiple changes underway, for new features, bug fixes, and so on.
- Staging area
- Limited support for binary files (the same with centralized VCSs)
- Learning curve

GIT – BRANCHING

- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory
- The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast.
- A branch in Git is actually a simple file that contains the 40 character **SHA-1 checksum** of the commit it points to
  - SHA (Secure Hash Algorithm) → generate a fix-length message digest from the input (plaintext)
  - Branches are cheap to create and destroy.
  - Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline)

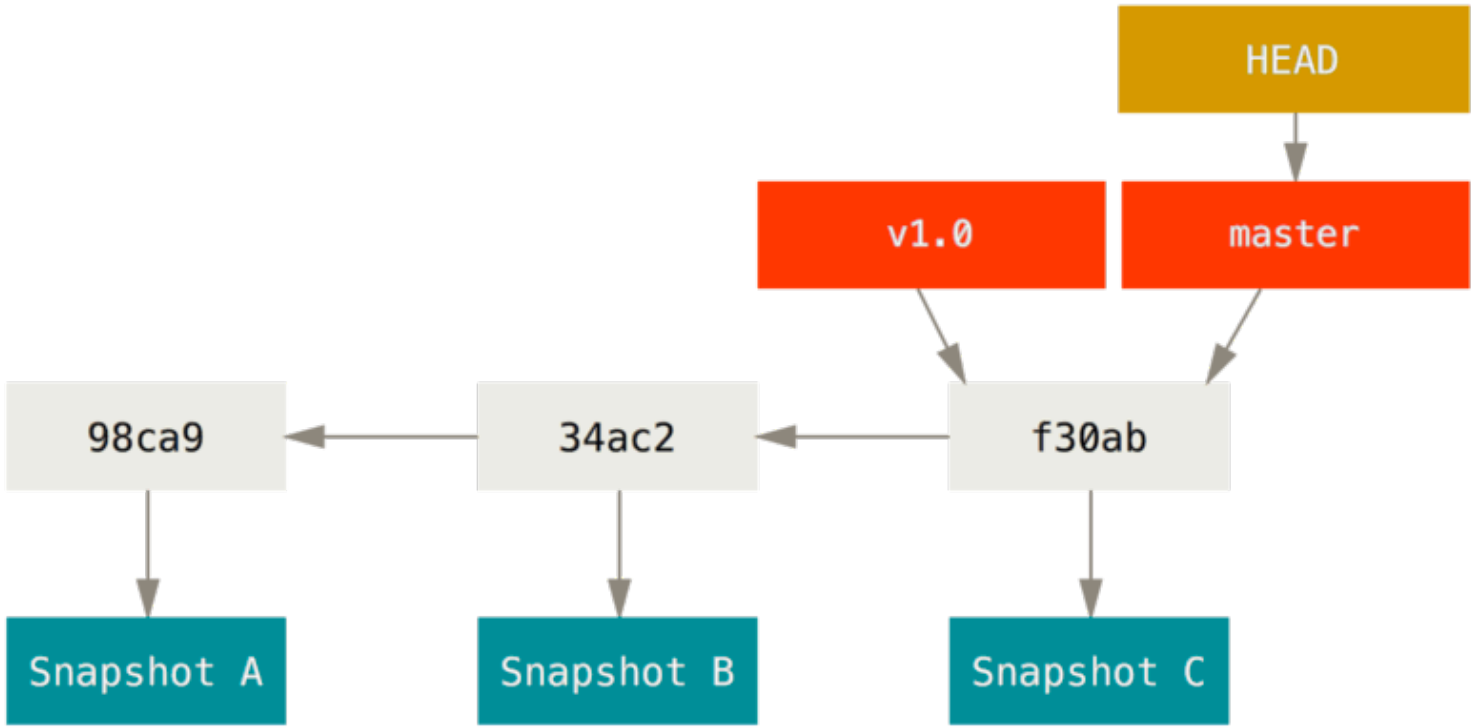


GIT – BRANCHING

- A commit and its tree (right)
  - Git creates 5 objects for a commit with 3 files
- Commits and their parent (left) – subsequent commits point to their direct parent commit

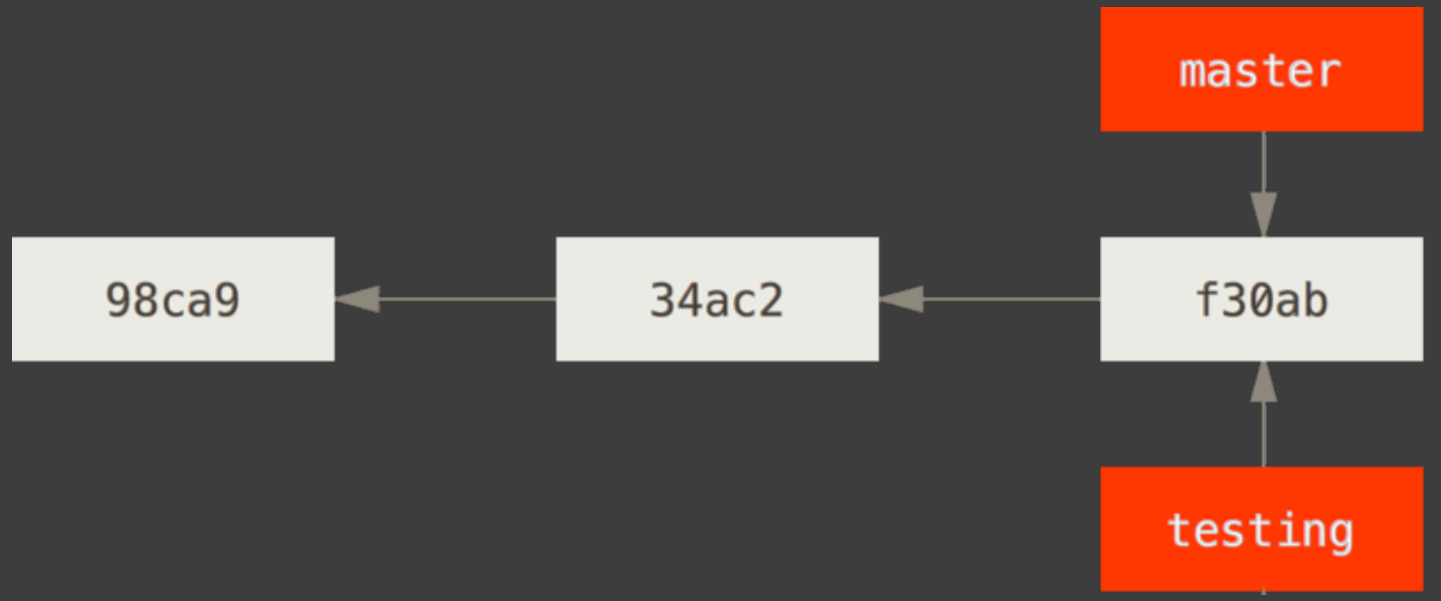
GIT – BRANCHING

- A branch in Git is simply a lightweight movable pointer to one of these commits.
- A branch and its commit history (figure above)
- The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made.
- The "master" branch in Git is not a special branch. It is exactly like any other branch.



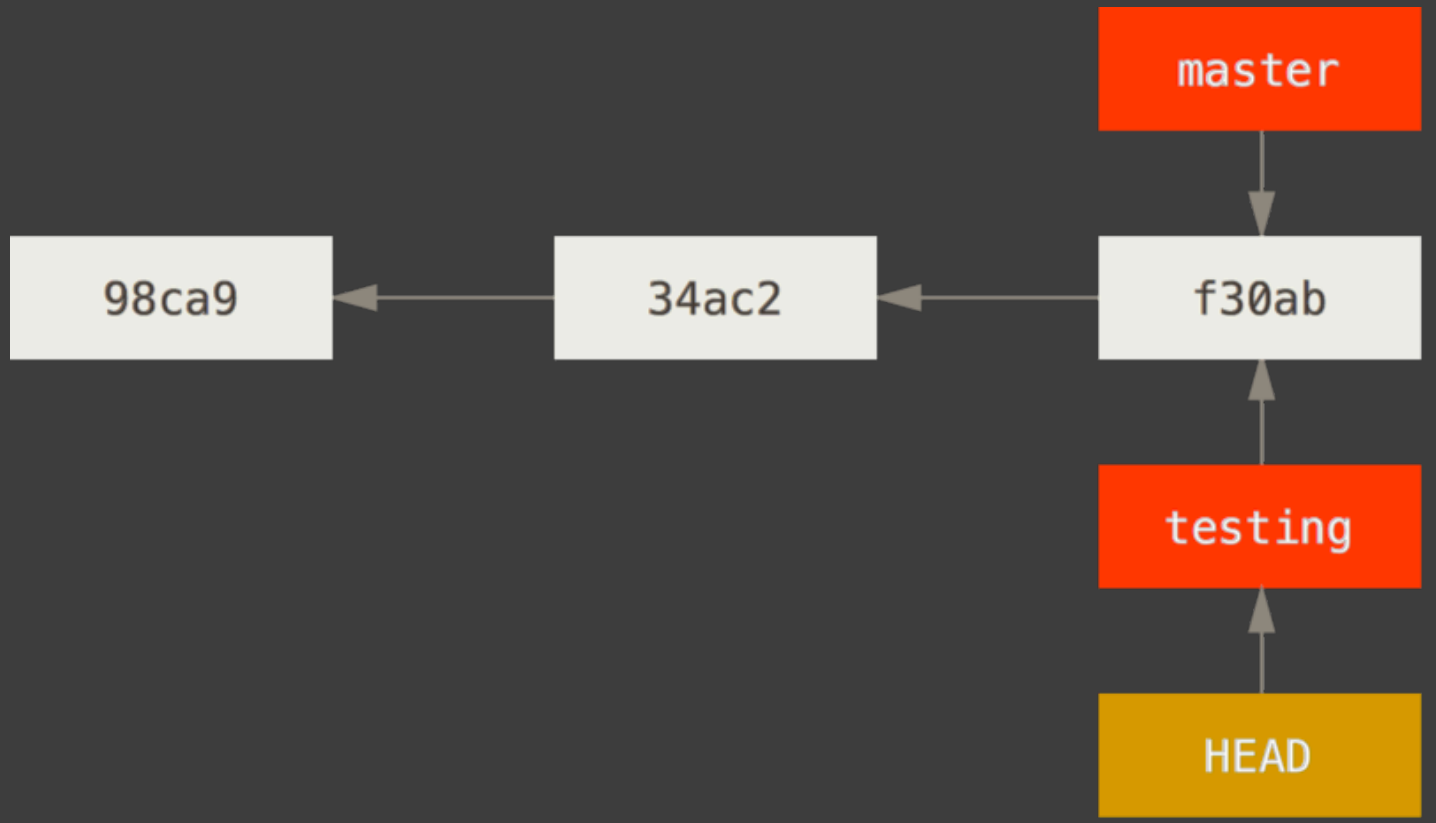
GIT – BRANCHING

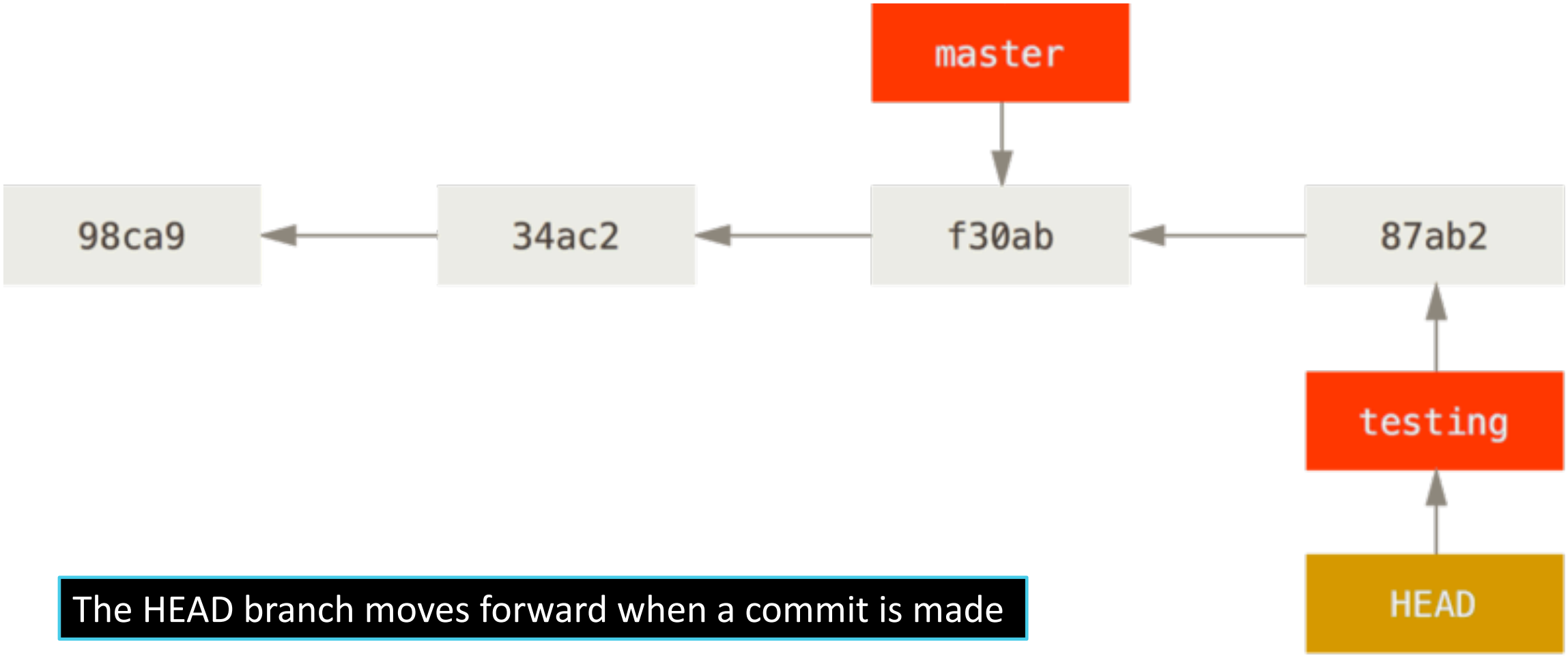
- Creating a branch is to create a new pointer to the same commit you're currently on
- Two branches pointing into the same series of commits



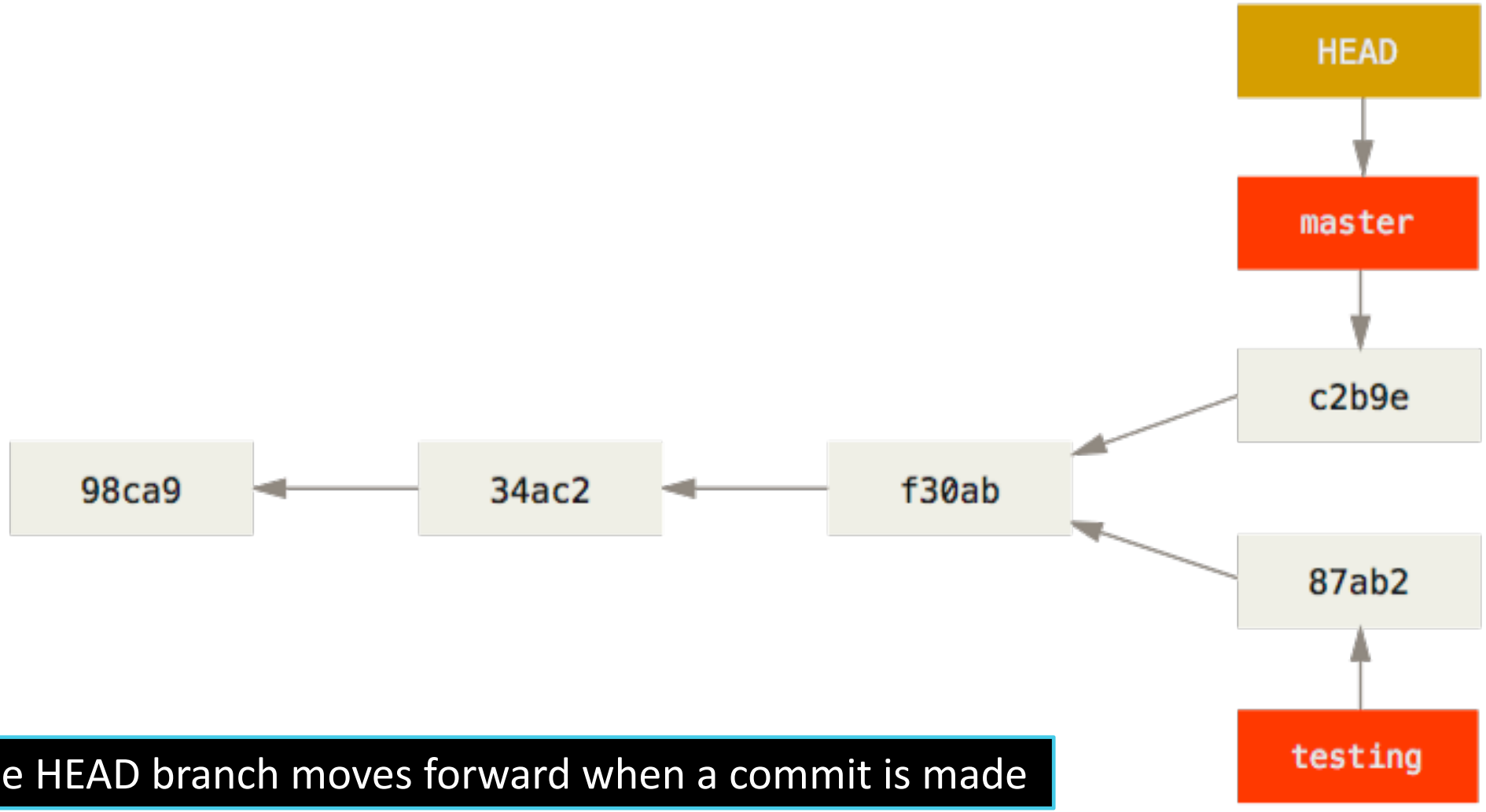
GIT – BRANCHING

- In Git, HEAD is a pointer to the local branch you're currently on
- For example, when you checkout "testing", this moves HEAD to point to the testing branch.

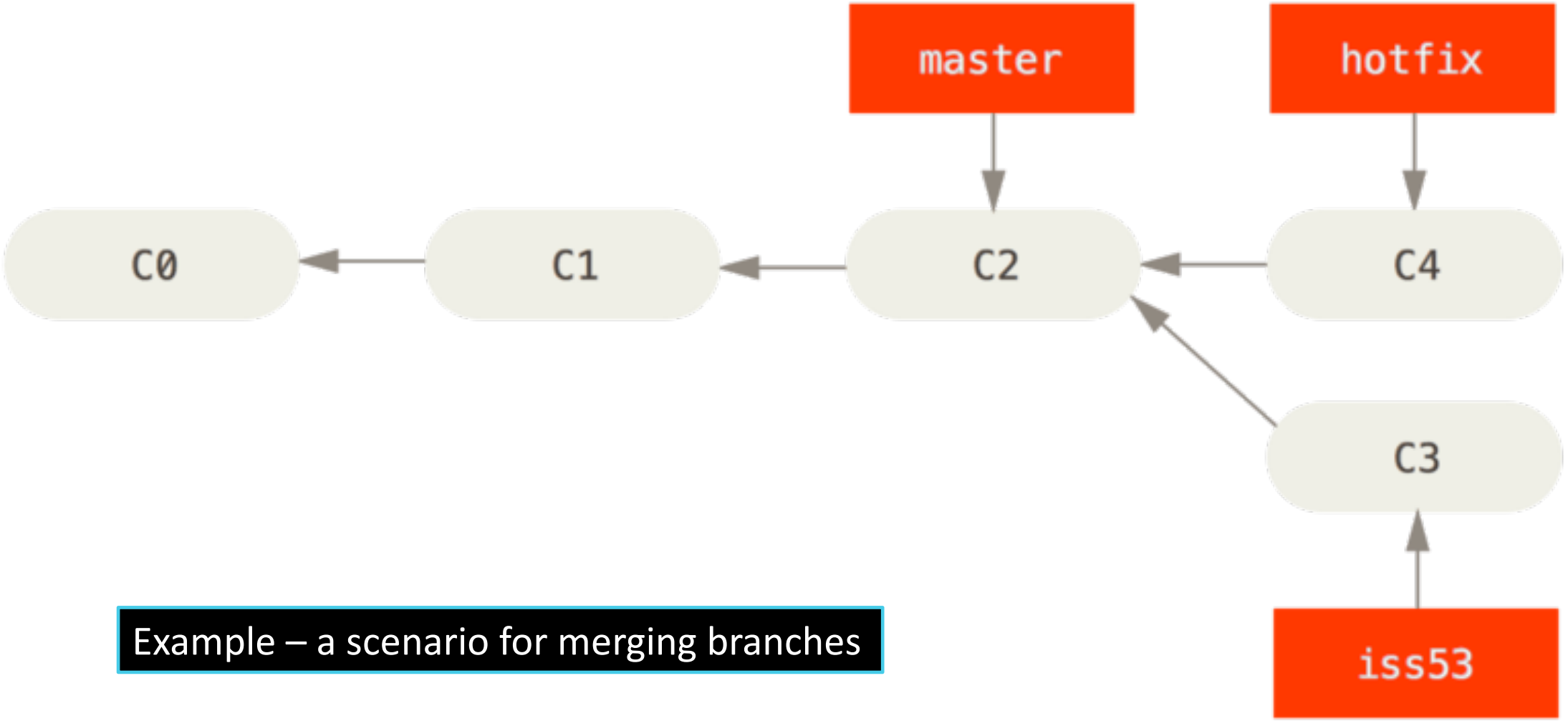




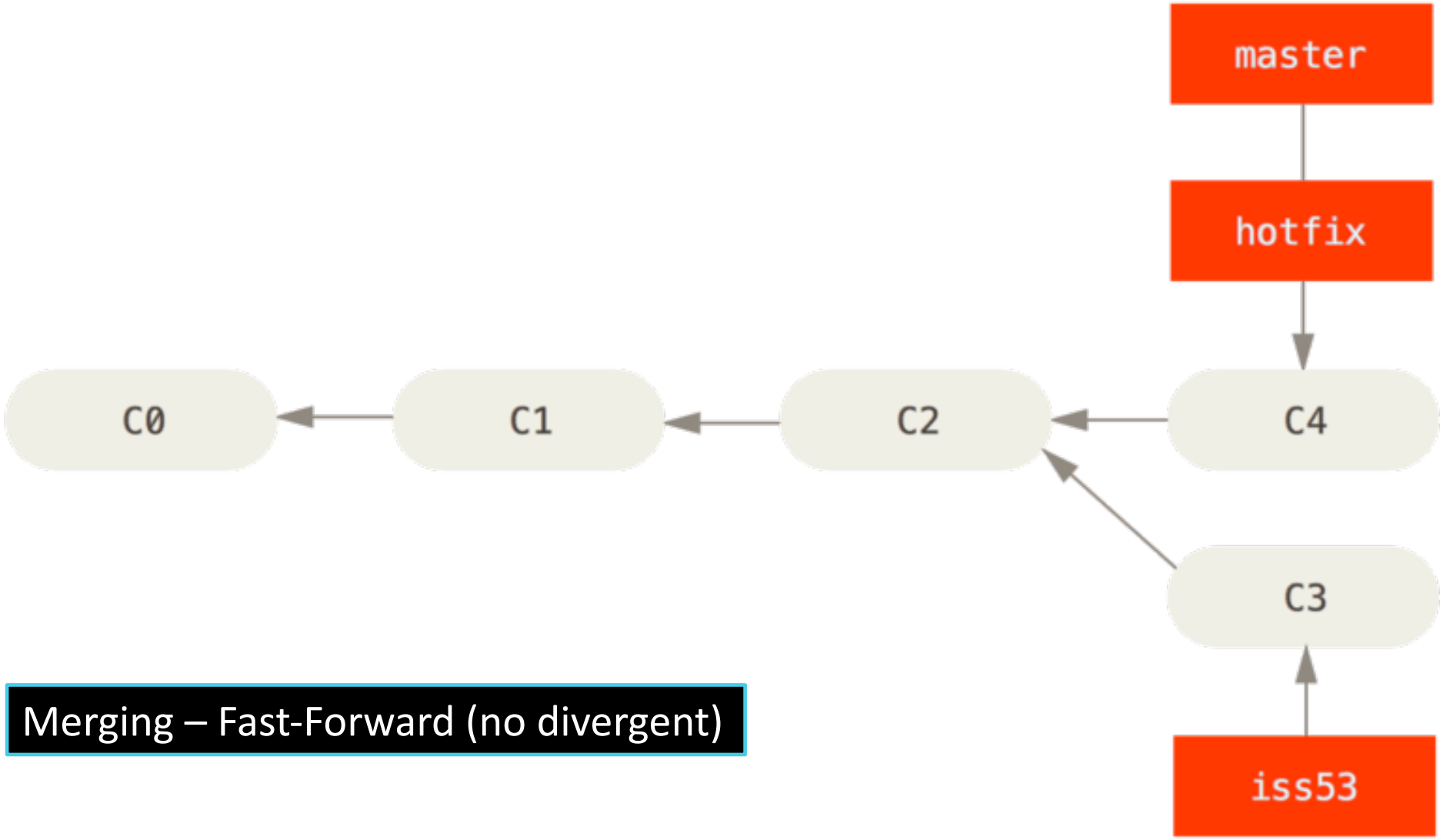
The HEAD branch moves forward when a commit is made



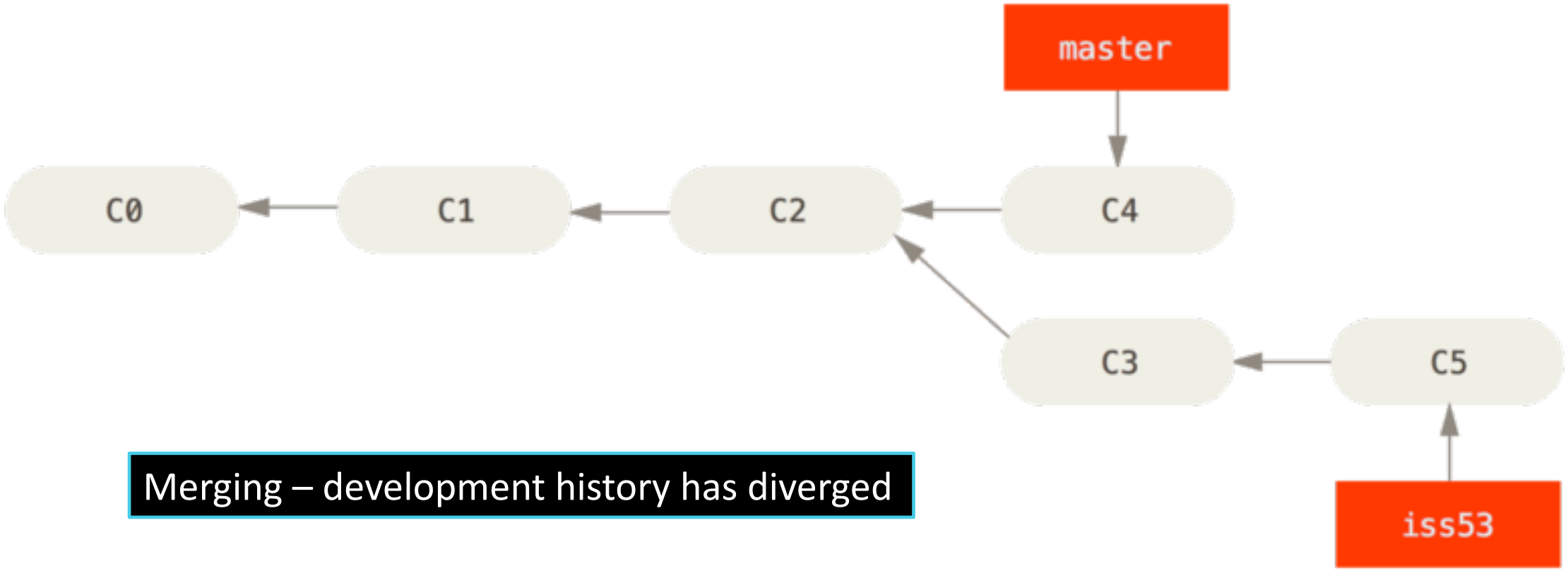
The HEAD branch moves forward when a commit is made



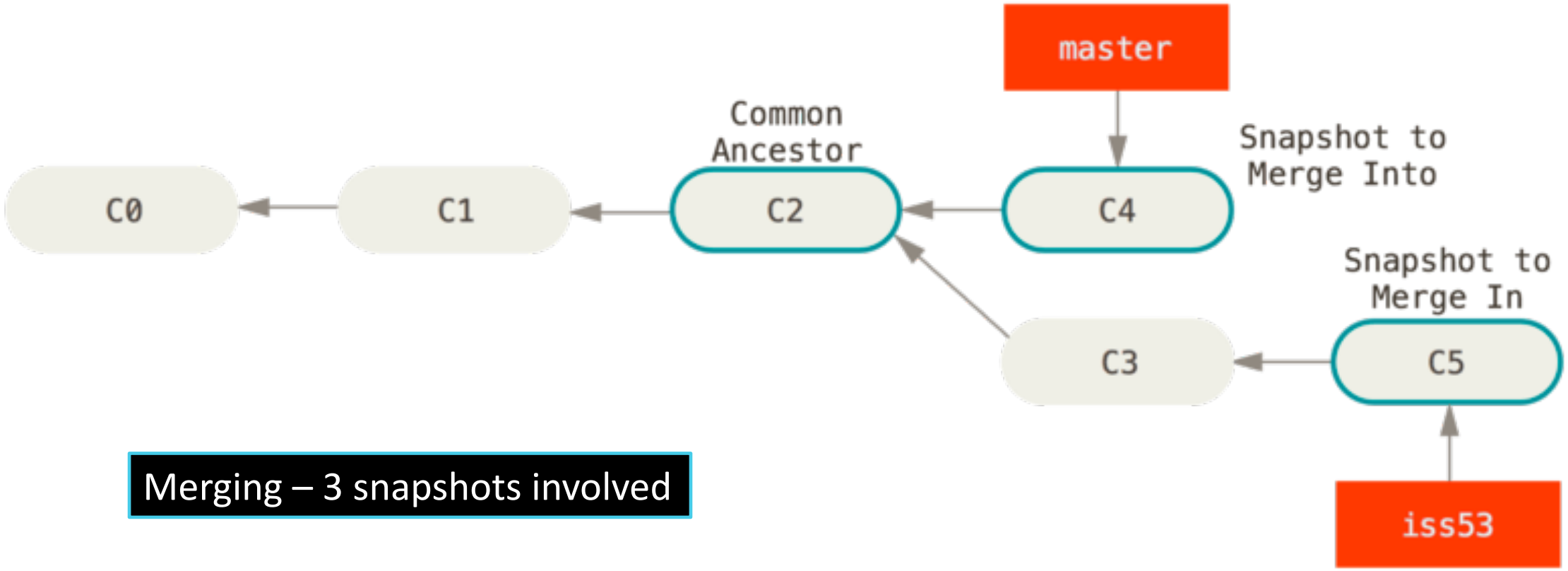
Example – a scenario for merging branches



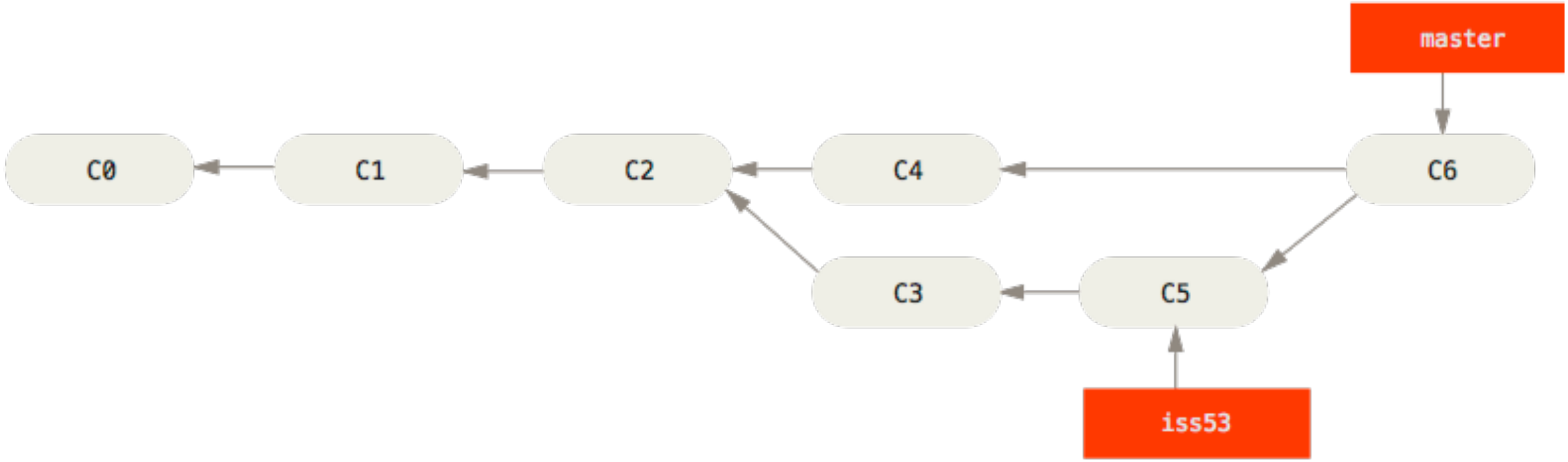
Merging – Fast-Forward (no divergent)



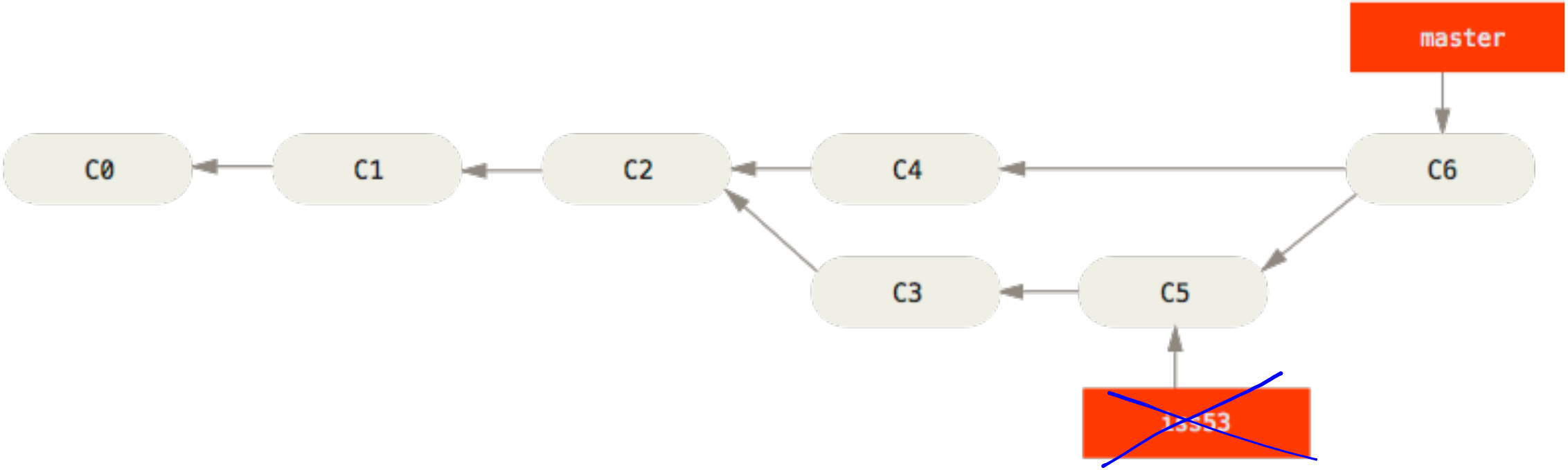
Merging – development history has diverged



Merging – 3 snapshots involved



Merge Commit – a new snapshot is created (C6)



You can remove the branch after the merge commit

RESOURCES

- Git hosting sites
  - GitHub – https://github.com/
  - Bitbucket – https://www.atlassian.com/git
- Git core – https://git-scm.com/downloads
- Git GUI clients – https://git-scm.com/downloads/guis/
- Git command cheat sheet: https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
- Free ebook: https://git-scm.com/book/en/v2

THANK YOU