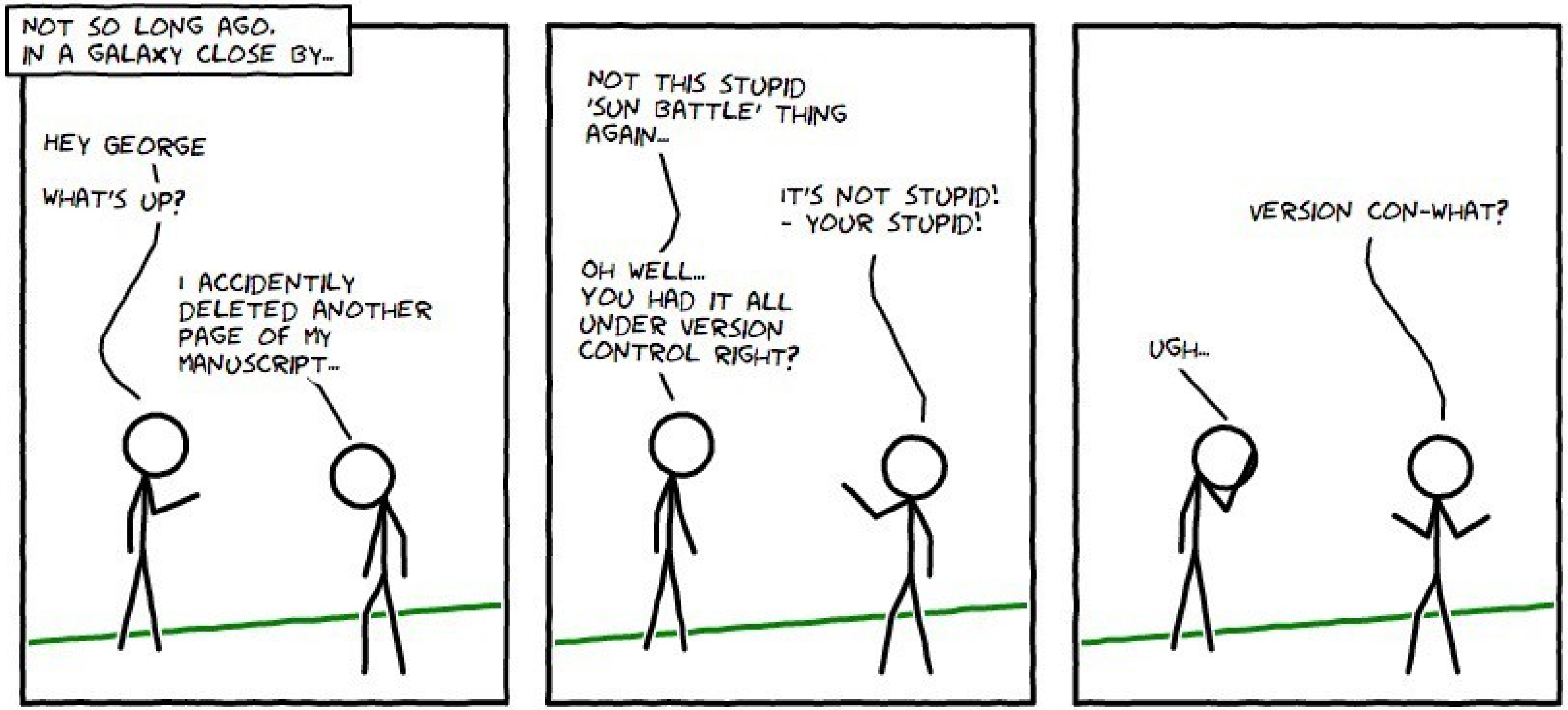
Git Basics

Introduction

- This module has two major sections
  - The first part will cover the basics of using the git version control system
  - The second part will focus on using git to support the collaboration of team members during development

Version Control



Version Control

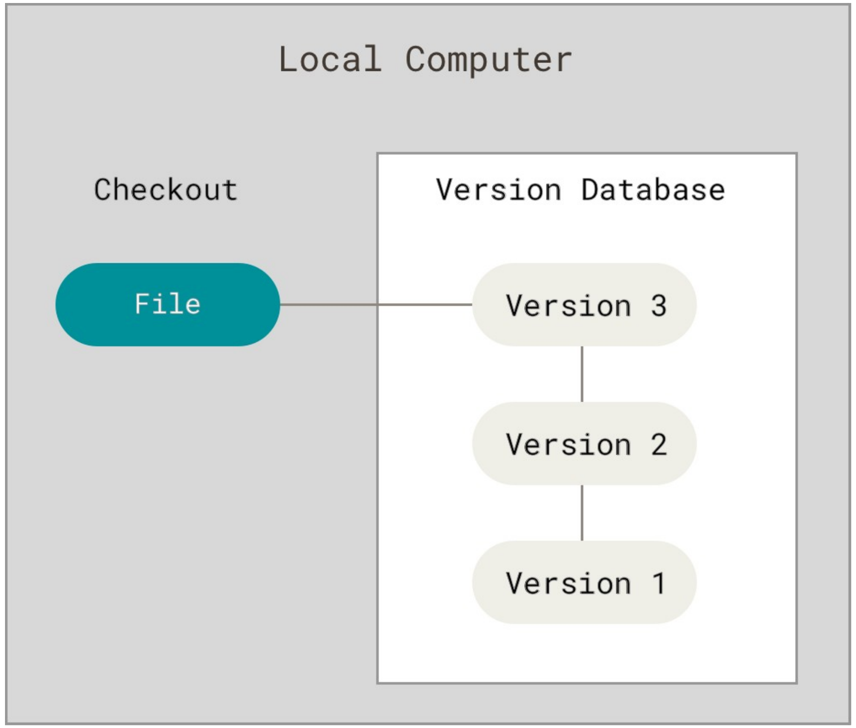
- A version control system (VCS)
  - Keeps a running history of your working files
    - Every change to the content is recorded
    - As well as "meta-data," who made the changes and when for example
- Allows you to revisit or recall earlier versions " like a “time machine” for your files.
- Allows multiple people to work on copies of the same file without overwriting each other's changes
  - And then to combine or merges their changes.
- It supports “branches,” where you experiment with changes without changing the main document

Version Control

- The software engineering reasons for using VCS
  - Recover & undo:Roll back bad changes or restore deleted work
  - Collaborate safely: Multiple people can code in parallel without overwriting each other's work
  - Traceability: Provides an audit record of what changed, who changed it, and why
    - Important for security and regulatory compliance
  - Quality control: Allows multiple states of development for controlled releases
    - We can support a development version and a production version for example
  - Confidence: Allows for modularity where we can refactor or experiment in a "sandbox" without affecting the production code

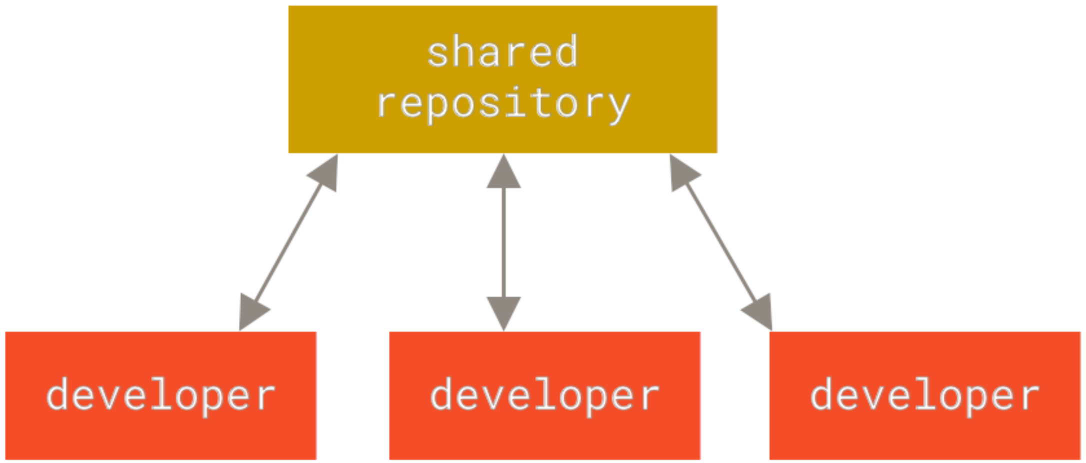
Local Version Control

- Represents the earliest type of automated version control
  - The repository was a locally managed directory on the local file system
  - Did not allow for collaboration in any effective manner between developers
  - Were popular before there was widespread networking (1980s)



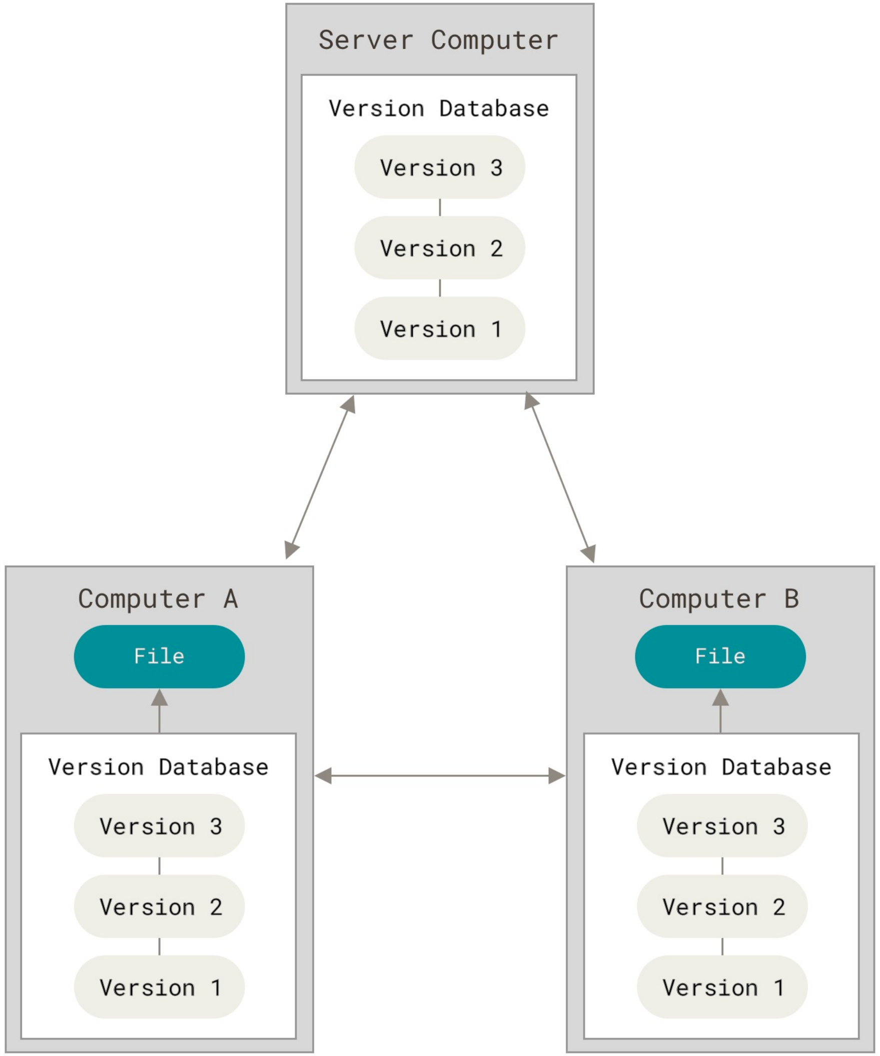
Centralized Version Control

- Designed for team collaboration
  - The repository is kept on a server
  - Working copies of files are checked out of the repository on the server
  - Changes are checked back in to update the repository
  - There is no local repository
- ClearCase is an example of a server VCS
  - Released in the early 1990s
  - Still has a massive installed base across large companies and government agencies



Distributed Version Control

- Like the local version control
  - Each user has their own local copy of a given repository
- Like a centralized system
  - There is what is called a remote repository where all the users merge and synchronize the changes made to their local repositories
- Weren't technically feasible in the past
  - Because it requires a significant amount of compute power to manage the contents of the repositories
  - For example, computing the hashes for the repo contents to ensure that synchronizations work without loss
  - Prior systems, like ClearCase were developed in eras where there were server limitations on disk space and processing capabilities
  - Distributed systems like git take advantage of increases in compute power, network capabilities and storage capacity

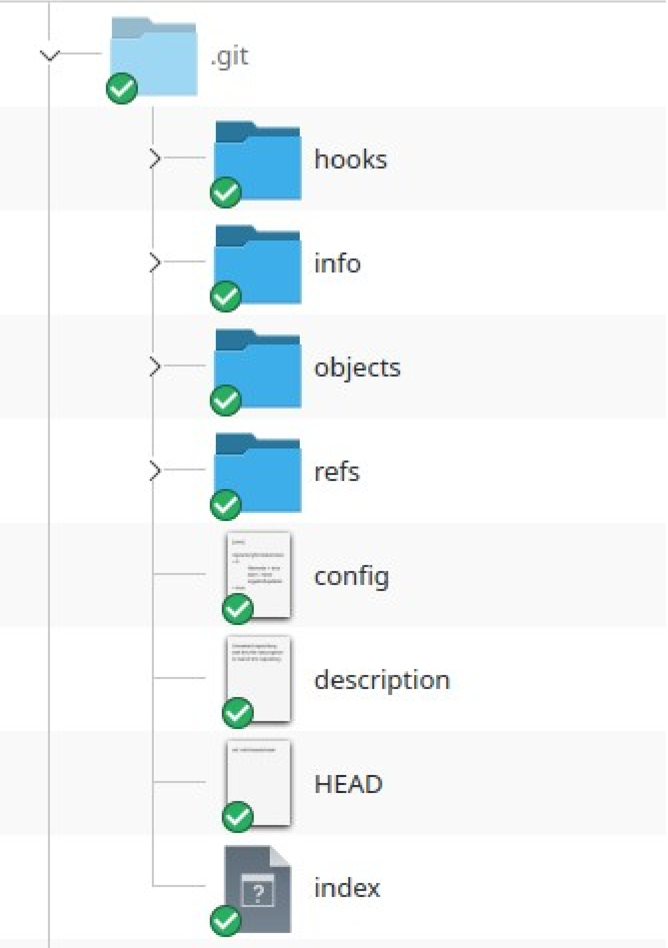


Origins of Git

- In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper
- In 2005, tool’s free-of-charge status was revoked
- In response, the Linux development community develop their own tool based on what they learned while using BitKeeper.
- Design goals:
  - Speed
  - Simple design
  - Strong support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like the Linux kernel efficiently (speed and data size)

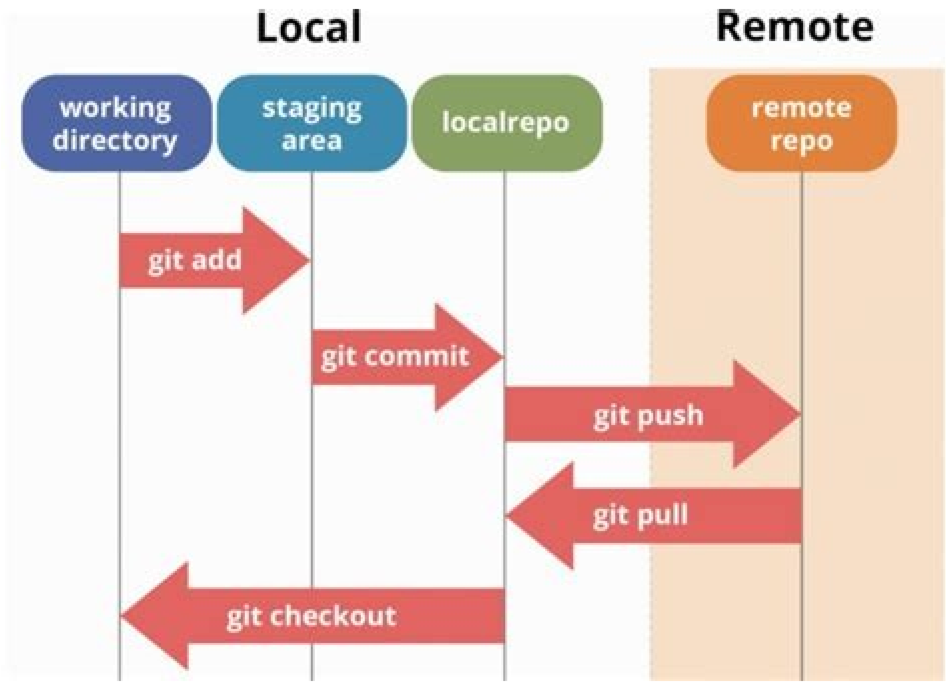
The .git repository

- Any directory can be used as a git repo
  - The command `git init` creates the repository in a subdirectory called `.git`
- The main components of the repo are:
  - Objects: the content store (Git's "vault").
    - Holds compressed, hashed objects: blobs, trees, commits, tags
  - Refs: human-friendly names that point to commits
    - For example refs/heads/main is your branch named Main
    - refs/tags/v1.0_ is a tag.
    - HEAD: a text file recording what the current branch is
  - Index: the staging area (a binary file) which lists exactly what will go into the next commit



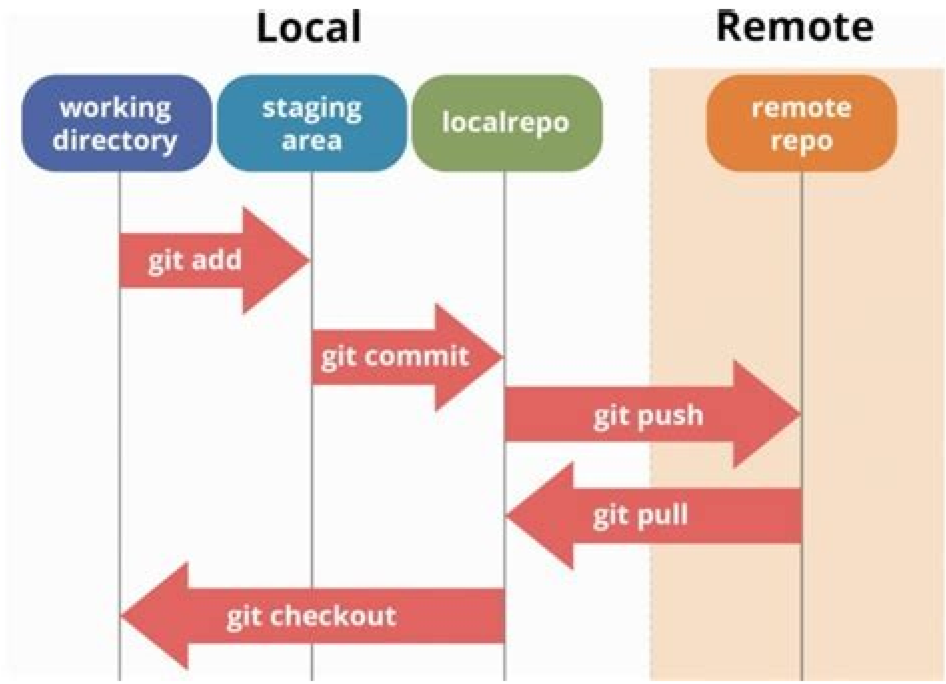
The git workflow

- Working directory
  - The set of files and directories in the same directory as the `.git` directory
  - These may or may not be in the repository
  - git ignores what you do in the work directory until you start the git flow
- Staging area (index)
  - List of files that you want to commit to the repository
- Repository (.git)
  - The database of past commits/objects (history).



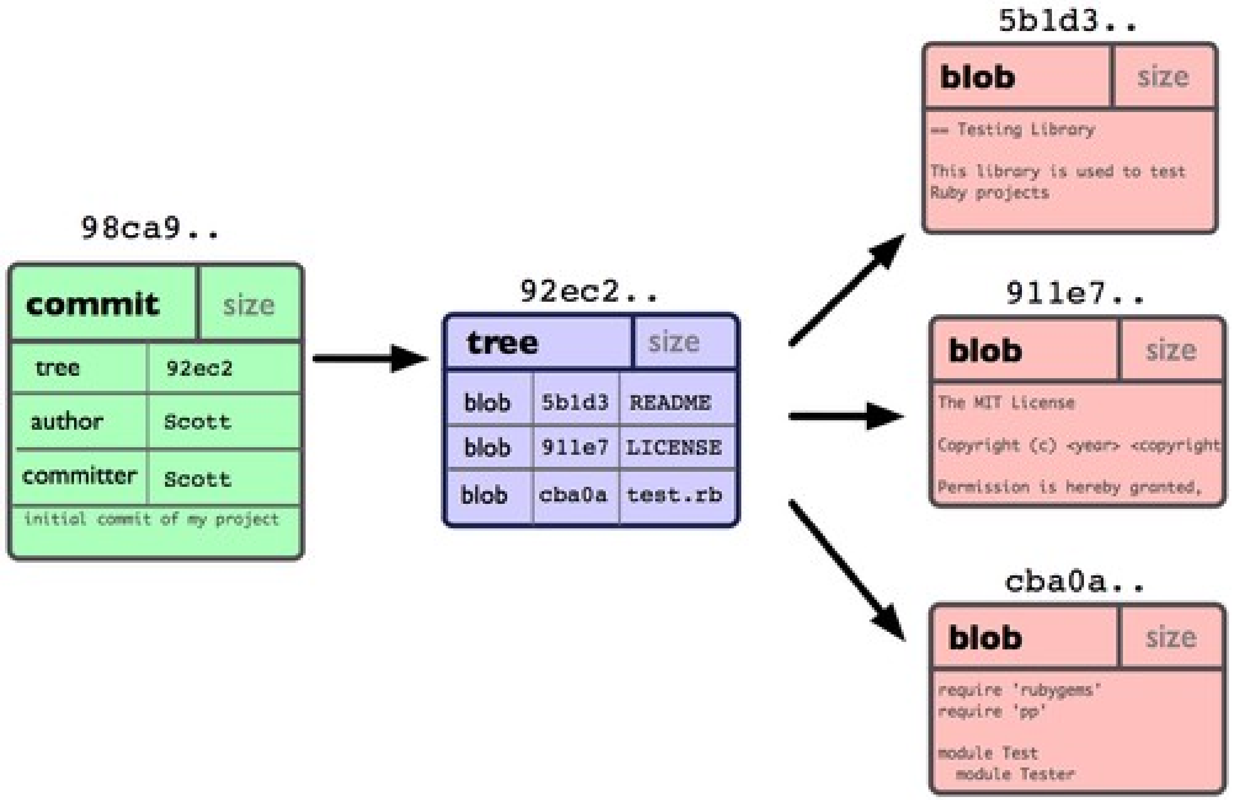
The git workflow

- The *git status* command tells you:
  - Which files in the working directory are not "tracked" or files that are in your working directory that are not in the repository
  - Which files are "staged" or will be added to the repository when you do a “git commit”
- The *git add* takes the current content of your files specified in the working directory
  - Creates/records blob objects
  - Updates the index to say “next commit should include these blobs at these paths.”
  - The index records the path, file mode, and the blob’s object ID (hash).
  - That means the stage index file holds a snapshot of content, not a pointer to the file in the work areas.
- The *git commit* command
  - Reads the index, writes tree objects to represent the directory structure into the repository and resolves what goes into the new commit
  - Creates a commit object that points to the new top tree and also points to the previous commit object
- This is covered in lab 5-1



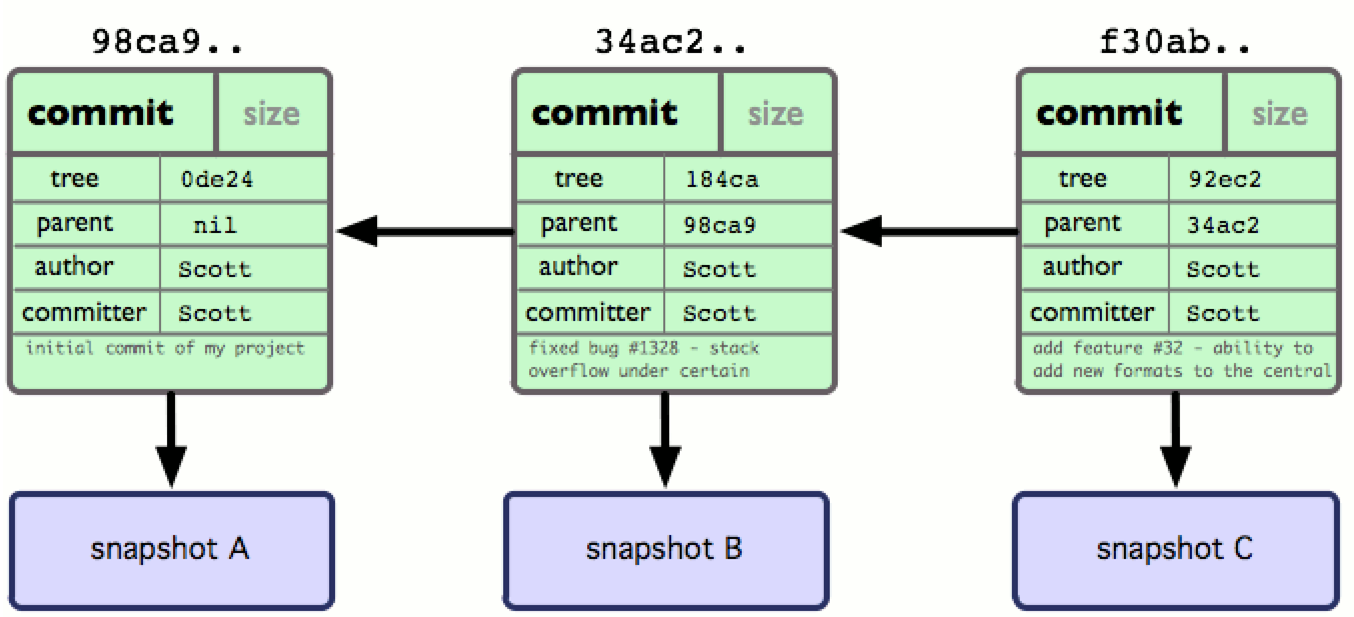
git History and Commits

- Each commit includes
  - Id - a hash of the contents
  - time stamp
  - author name and email
  - message describing the commit
  - link to previous commit
  - link to files in this commit



git History and Commits

- git history is a linked list of commits
  - Each commit links back to previous commits
- Branches are just pointers to commits
  - The `HEAD` points to the commit that is currently in the working directory
- In the diagram, `master` and `feature/2` are branch labels

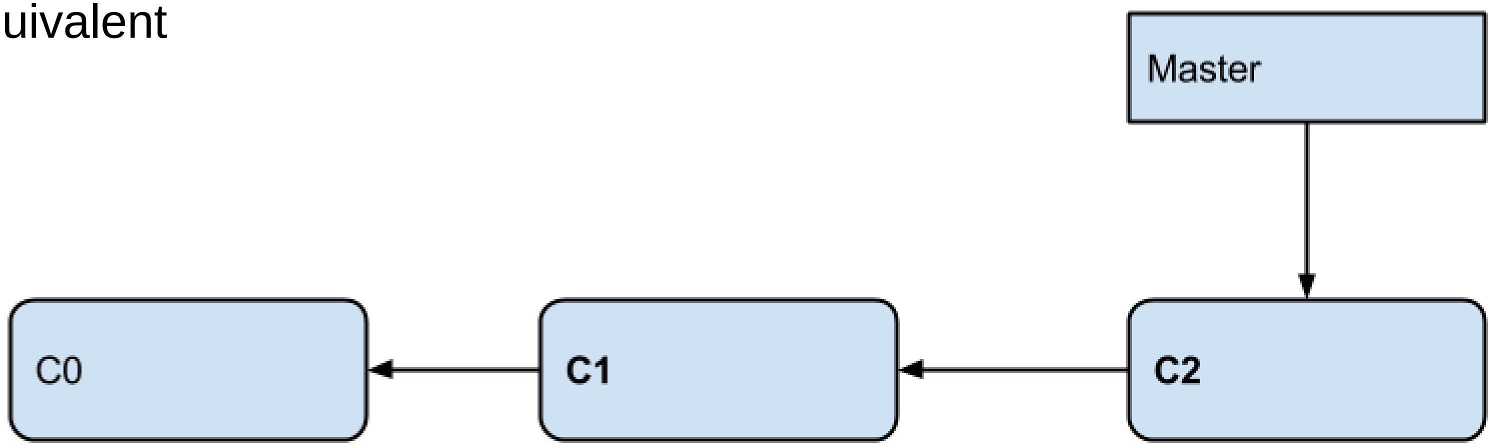


Checking files out

- There are several commands to restore file
  - git checkout
  - git restore
  - The restore command is a new version and checkout is the legacy form
- If we want to restore a file from any of the previous commits
  - We just have to specify the file and the commit using either
    - The hash of the commit (only the first five or so characters suffice)
    - How many commits counting backward from the HEAD commit
  - You will do this in lab 2
- You can also checkout or restore a whole commit.
  - This has the effect of overwriting your current working directory with a given commit
  - This can be dangerous since it might overwrite any unsaved work you have

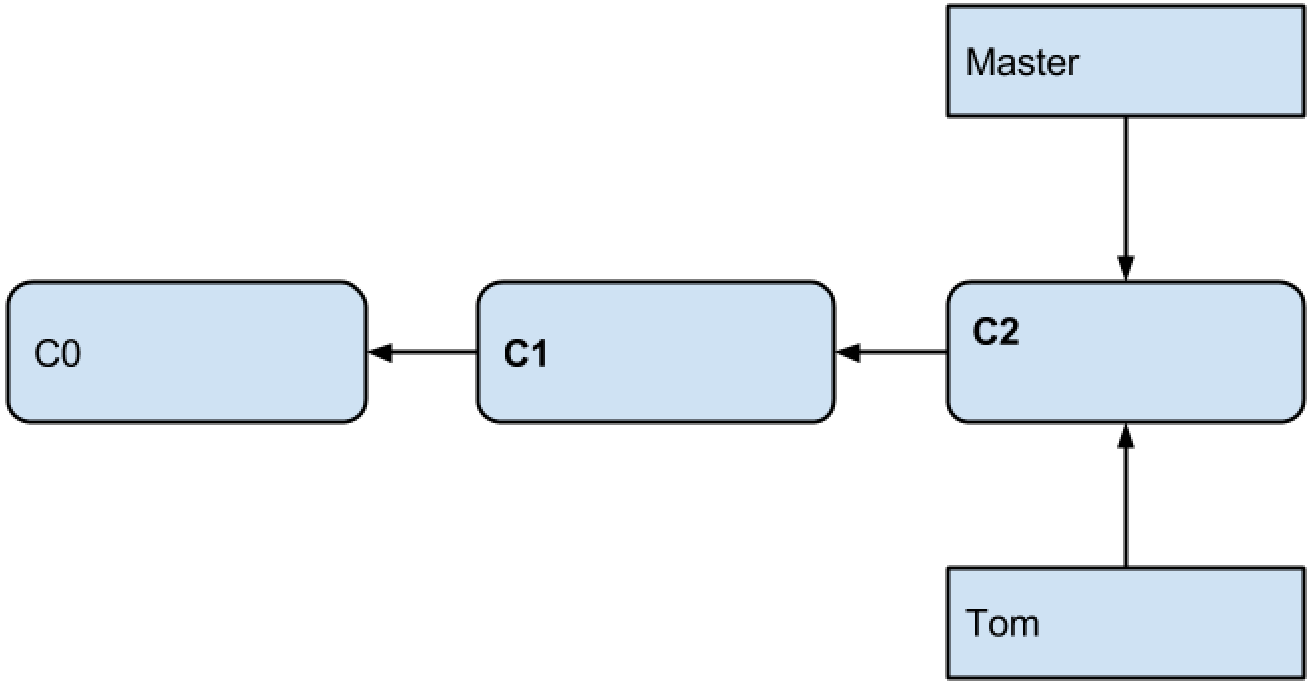
Branching and Merging

- A branch is just a pointer to a specific commit
  - Nothing is created when you create a branch other than the name
  - The default branch is called “main” but you can name it whatever you want
  - In older version of git, the default was named “master”
  - “main” and “master” are equivalent



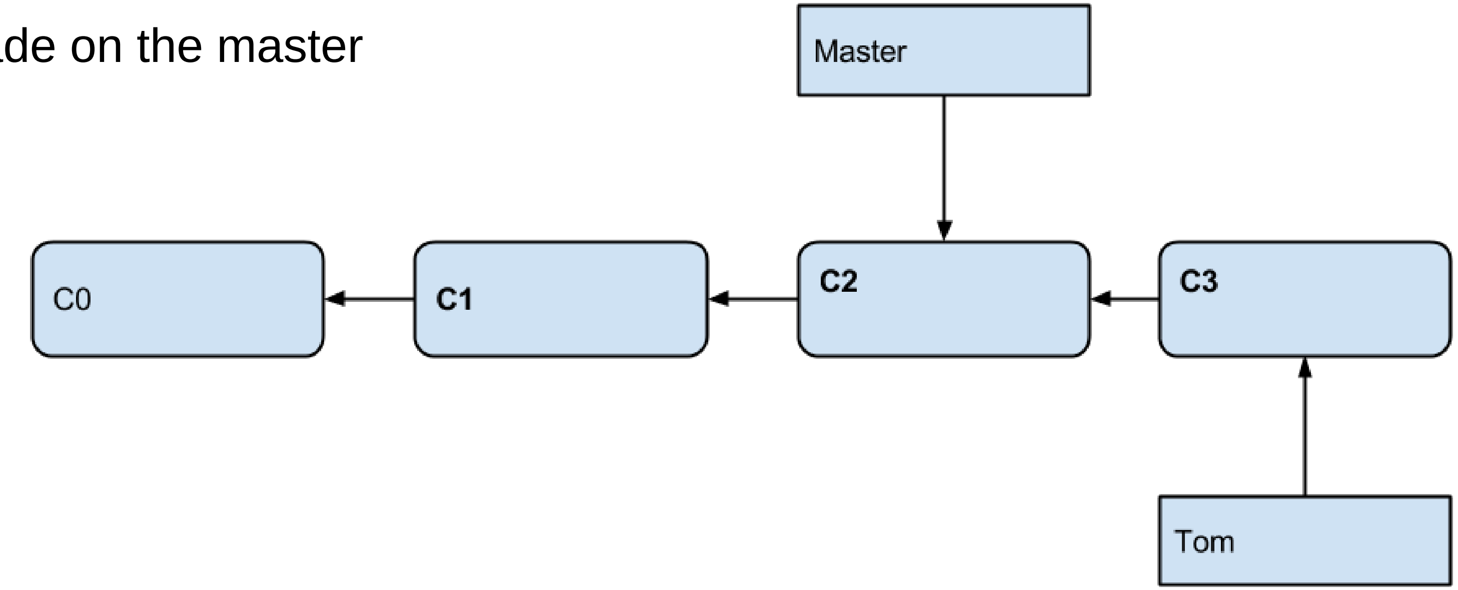
Creating a new branch

- Git checkout tom -b
  - Creates a new branch pointer which points to the current “HEAD” commit



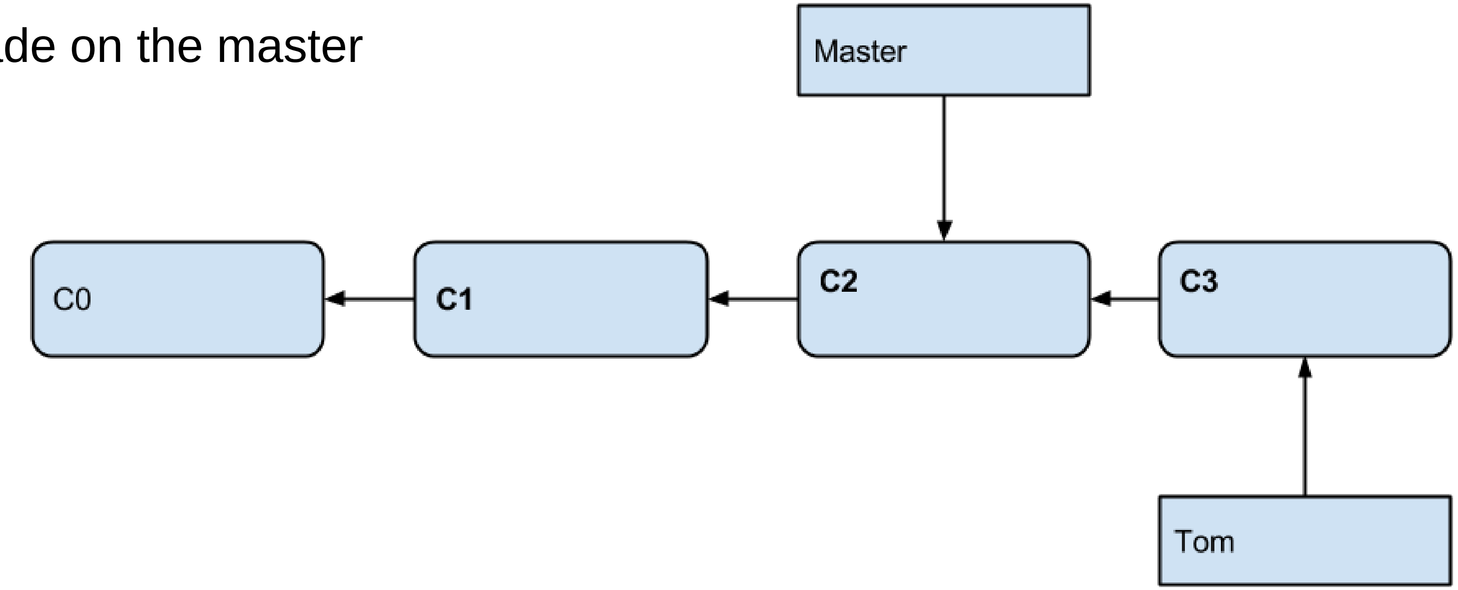
Commit on the branch

- Git commit -m “msg”
  - Creates a new commit
  - The branch that created it now points to the new commit
  - The master branch still points to the previous commit
  - Because the commit was not made on the master branch



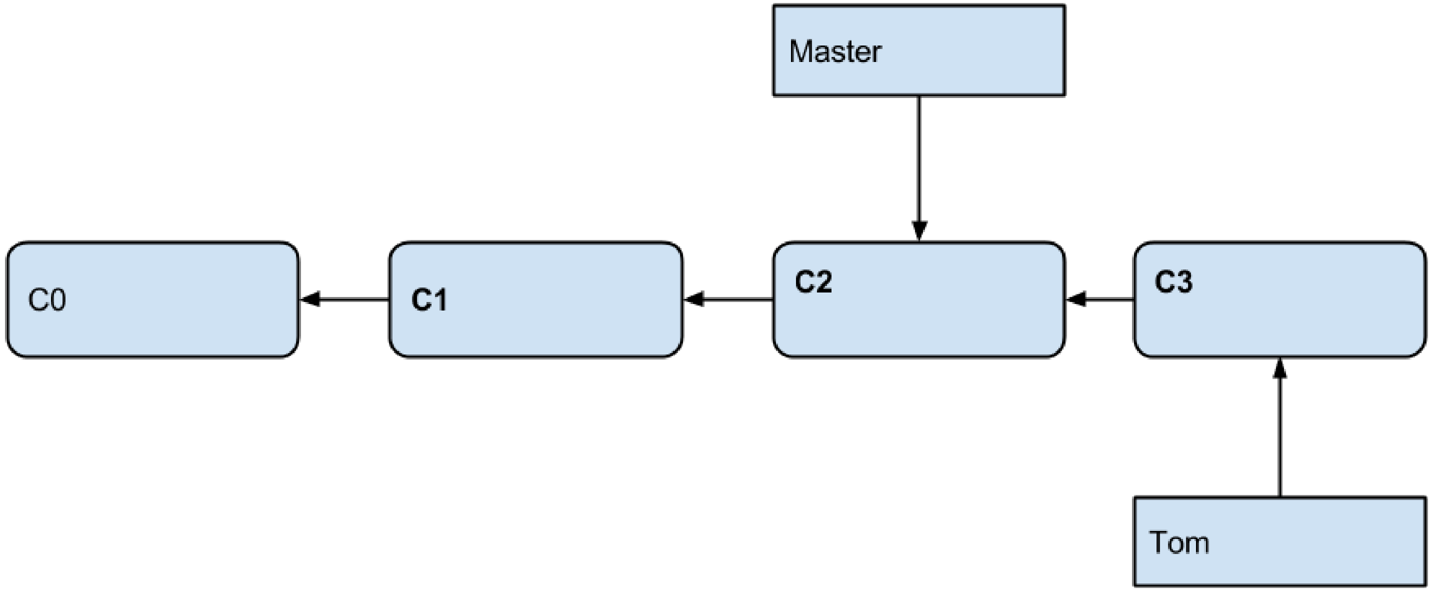
Merging the branch

- Git commit -m “msg”
  - Creates a new commit
  - The branch that created it now points to the new commit
  - The master branch still points to the previous commit
  - Because the commit was not made on the master branch



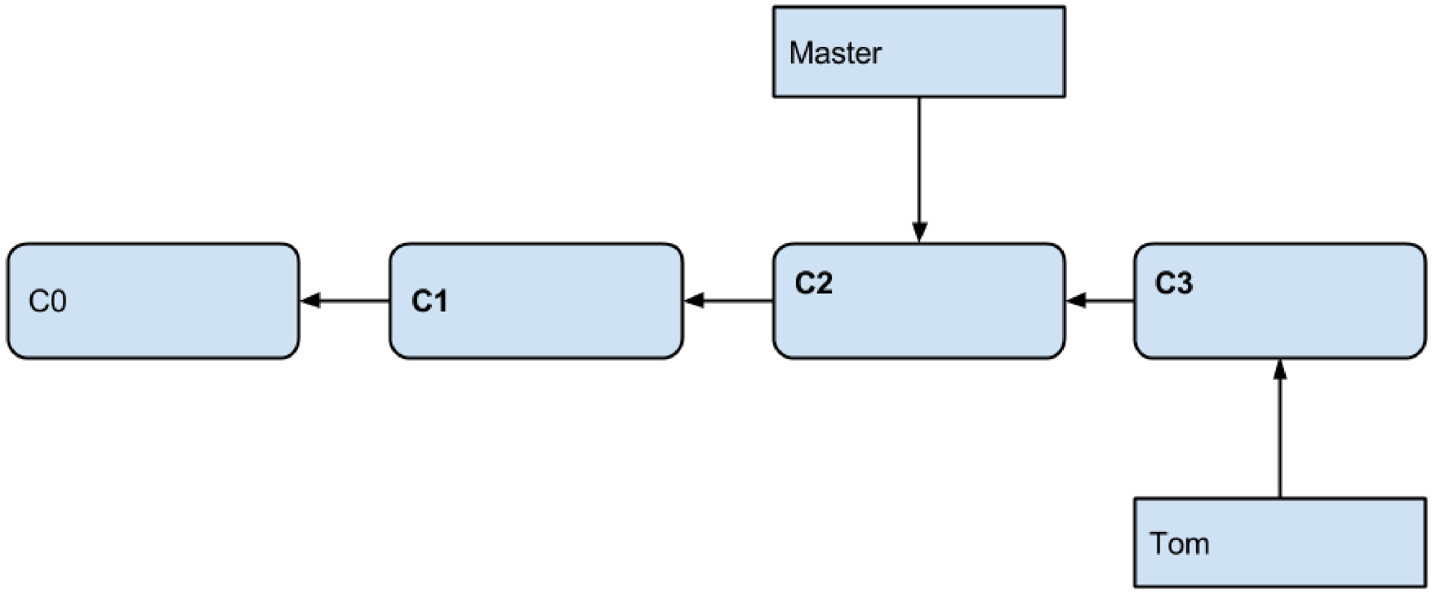
Switching the branch

- Checking out the different branches moves the contents of the commit into the working directory
  - Checking out Master loads C2 into the working directory
  - Checking out Tom loads C3 into the working directory



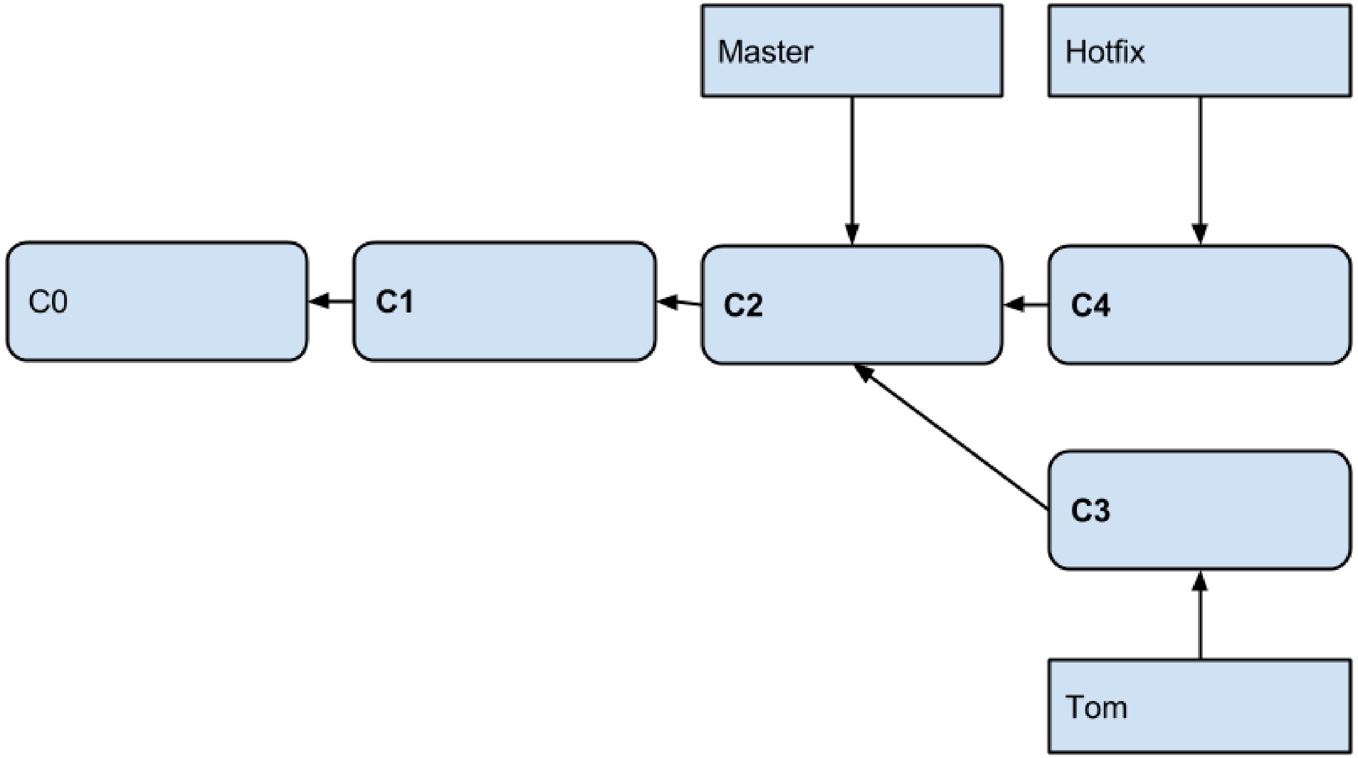
Merging

- In this case, we have a fast-forward merge
  - To merge the contents of the Tom branch into the Master branch, we just move the Master branch to point to C3



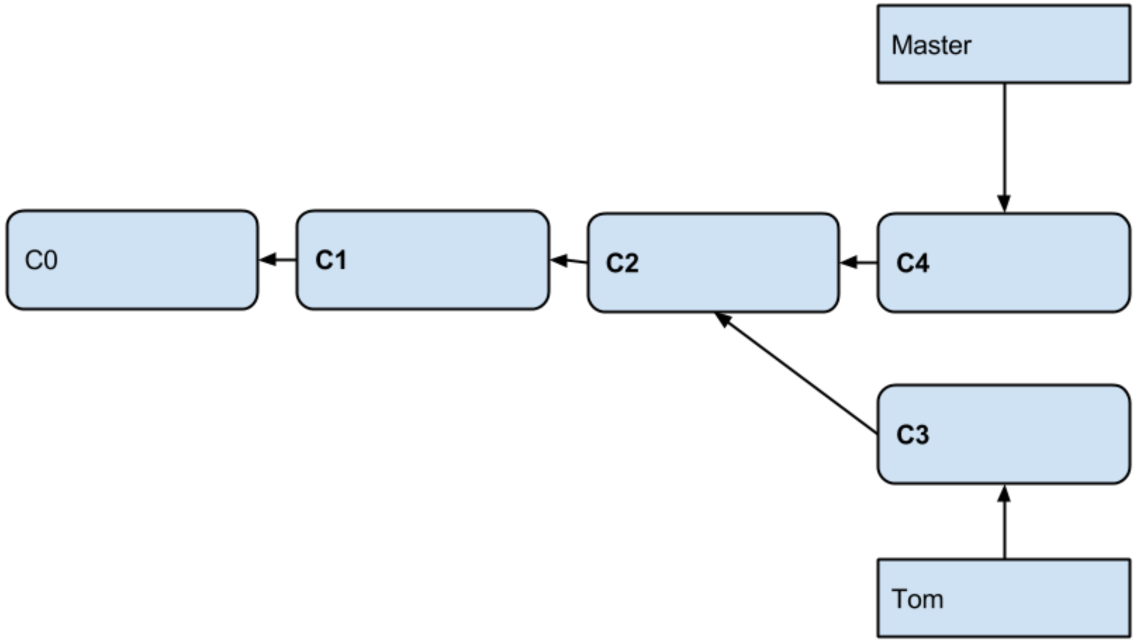
Merging

- Now we have a more complicated merge
  - We have two branches each with a commit
  - We can’t just move pointers
  - We have to do this in two merges



Merging

- First we can do a fast forward merge to merge Hotfix into Master
  - We just move Master to C4
  - Then delete the Hotfix branch name
  - But now we have C3 and C4
  - Each has changes the other doesn’t
  - The second merge is to merge C3 into C4
  - We may need to resolve conflicts
  - The result is a new commit C5 (not shown)



Git for Collaboration

- Branches
  - A movable pointer to a commit.
  - By default, Git starts with a branch called main (or master in older repos)
- Why use branches?
  - Isolation: Work on features or fixes without disturbing the stable code
  - Parallelism: Multiple team members can develop in separate branches simultaneously
  - Experimentation: Try ideas without affecting production
- Common branch types:
  - Feature branches for new functionality (e.g., feature/login-form)
  - Bugfix branches to address issues (bugfix/null-pointer)
  - Release branches to prep for stable versions (release/v1.2)
  - Hotfix branches for critical patches applied to production (hotfix/security-patch)

Command Summary

```
git branch                  # list branches
git branch new-feature      # create a new branch
git switch new-feature      # switch to the branch
git checkout -b hotfix      # create and switch in one command
git merge branch-name       # merge another branch into current one
git branch -d old-feature # delete a branch (after merge)
```

Git for Collaboration

- Merging
  - Combine changes from one branch into another (usually from feature into main)
- Types of merges:
  - Fast-forward merge
    - *Happens when the target branch is directly ahead of the source branch with no divergent commits*
    - *Simply moves the pointer forward*
  - Three-way merge
    - *Occurs when branches have diverged*
    - *Git uses the common ancestor + changes from both branches to create a new “merge commit”*
    - *Produces a new commit with two parents*

Git for Collaboration

- Merge conflicts:
  - If two branches modify the same lines in a file (or one deletes a file that another changes)
  - How to resolve:
    - *Git marks conflict sections with <<<<<<<, =======, >>>>>>>.*
    - *Developer edits file to decide what stays*
    - *Stage the resolved file*
    - *Commit the resolved file*
- Best practice
    - *Keep commits small and frequent → easier conflict resolution*

Git for Collaboration

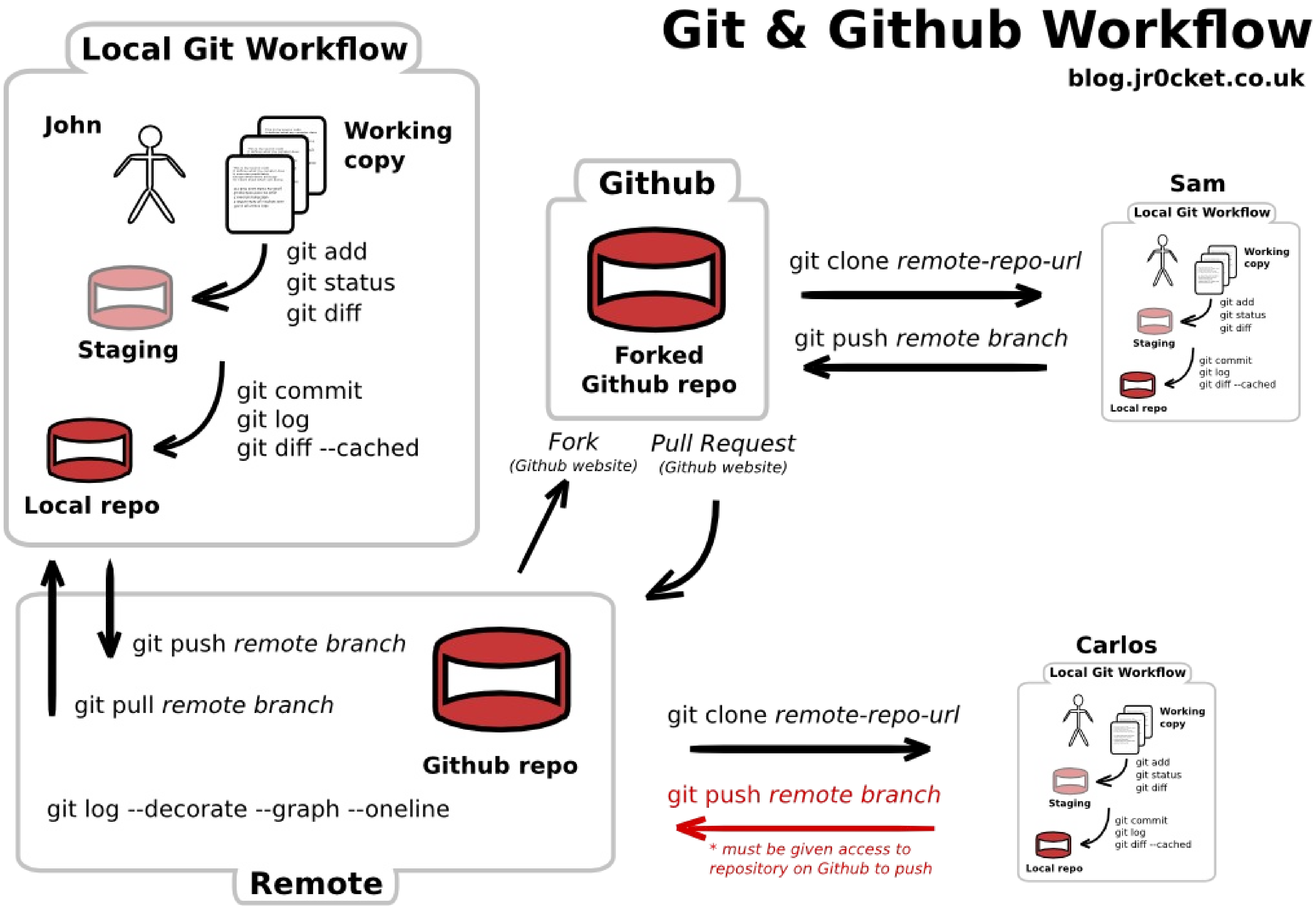
- Tags & Releases
  - Tags are immutable pointers to specific commits
    - *Usually marking important milestones (like version numbers)*
- Types of tags:
  - Lightweight tag: simple pointer to a commit
    - *command: git tag v1.0*
  - Annotated tag: includes metadata (tagger name, date, message, optional GPG signature)
    - *Command: git tag -a v1.0 -m "Release version 1.0"*

Git for Collaboration

- Releases on GitHub/GitLab:
  - Releases build on tags:
  - Each release corresponds to a tag
  - Adds release notes, changelogs, binaries, or compiled artifacts
- Benefits of releases
  - Users can download stable versions
  - Developers can see clear history of stable milestones
  - CI/CD pipelines often trigger builds based on tags/releases

Remote Repositories

- A remote repository is a version of your Git repository hosted on another server, typically online
- Benefits:
  - Collaboration: Provides a shared location where multiple developers can contribute code
  - Backups: Protects project history by storing it outside of your local machine
  - CI/CD Integration: Enables continuous integration (tests, builds) and continuous deployment pipelines
  - Transparency: Makes project history visible to teams or the public (open source)
  - Access Control: Permissions and roles determine who can read, clone, or write to the repository



Git & Github Workflow
blog.jr0cket.co.uk
Local Git Workflow
John
Working copy
git add
git status
git diff
Staging
git commit
git log
git diff --cached
Local repo
Github
Forked Github repo
git clone *remote-repo-url*
git push *remote branch*
Sam
Local Git Workflow
Working copy
git add
git status
git diff
Staging
git commit
git log
git diff --cached
Local repo
Fork
(Github website)
Pull Request
(Github website)
git push *remote branch*
git pull *remote branch*
Github repo
git log --decorate --graph --oneline
Remote
git clone *remote-repo-url*
git push *remote branch*
** must be given access to repository on Github to push*
Carlos
Local Git Workflow
Working copy
git add
git status
git diff
Staging
git commit
git log
git diff --cached
Local repo

Common Remote Operations

- Adding a remote
  - *git remote add origin <url>*
  - Links local repository to a remote server (origin is the default nickname)
- Pushing (uploading) commits:
  - *git push origin main*
  - Sends local changes to the remote branch
- Fetching (downloading) commits:
  - *git fetch origin*
  - Updates your local repo with remote changes (but does not merge)

Common Remote Operations

- Pulling (fetch + merge):
  - *git pull origin main*
  - Updates local branch with remote branch changes
- Viewing configured remotes:
  - *git remote -v*
  - Shows remote names and URLs
- Removing or renaming remotes:
  - *git remote remove origin*
  - *git remote rename origin upstream*

Popular Platforms

- GitHub
  - World’s largest platform for open-source projects.
  - Massive developer community
  - GitHub Actions for CI/CD automation
  - Pull Requests (PRs) for collaborative reviews
- Best suited for:
  - Open-source collaboration
  - Small to large projects needing global visibility

Popular Platforms

- GitLab
  - All-in-one DevOps platform (repository hosting + CI/CD + issue tracking)
  - Built-in pipelines (GitLab CI/CD)
  - Fine-grained permission and role management
- Best suited for:
  - Enterprises that need a self-hosted or cloud-based DevOps solution
  - Private repositories with strong process integration

Popular Platforms

- Bitbucket
  - Tight integration with Atlassian tools (Jira, Confluence)
  - Supports Git and Mercurial (legacy)
  - Bitbucket Pipelines for CI/CD
- Best suited for:
  - Teams already using Jira/Confluence
  - Enterprises needing Atlassian ecosystem integration
- Cloud Platforms
  - Most cloud vendors have a remote repository facility
  - Designed to integrate with their in-Scloud workflow tools

Workflow Models

- Workflows define how teams organize development.
- A consistent branching model:
  - Prevents chaos when multiple people code at the same time
  - Reduces merge conflicts
  - Supports testing, staging, and production pipelines
  - Different workflows fit different team sizes and project types
- Branching strategies automate your existing workflow
  - Implementing a branching model without a well defined workflow never ends well

Common Workflows

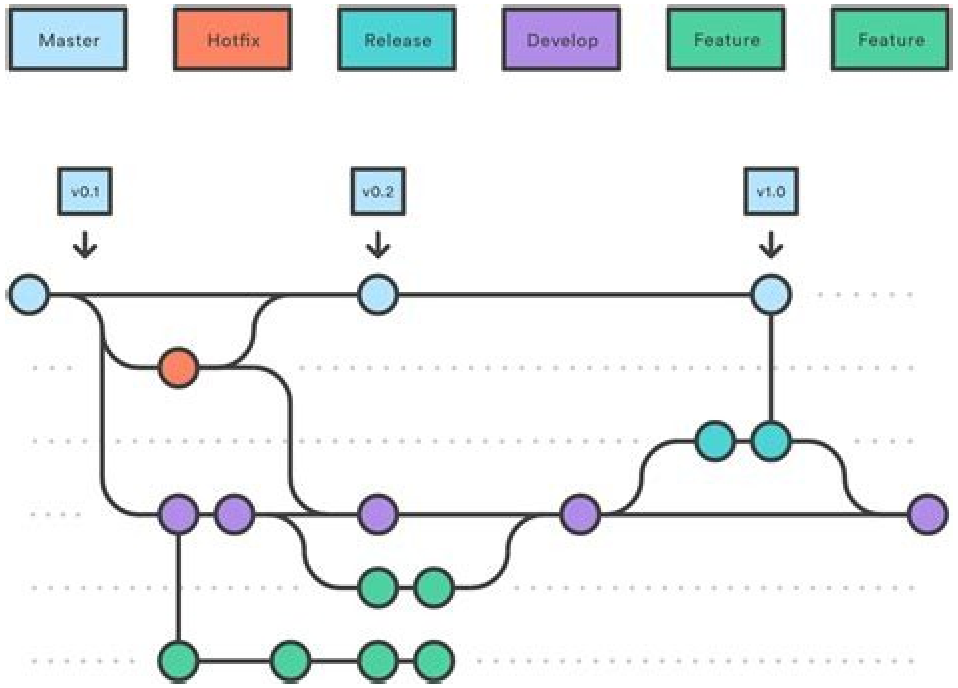
- Centralized Workflow
  - Everyone commits directly to the main branch (like traditional version control)
- Pros
  - Simple, minimal Git knowledge needed
  - Works well for small teams or solo projects
- Cons:
  - High risk of conflicts
  - Harder to isolate unfinished work

Common Workflows

- Feature Branch Workflow
  - Each feature or bugfix is developed in its own branch
  - Once complete, the branch is merged into main (often via Pull Request/Merge Request)
- Pros:
  - Keeps main stable
  - Easier to review changes (via pull requests)
  - Encourages collaboration through code review
- Cons:
  - Many branches may pile up if not managed

Common Workflows

- Gitflow Workflow
- Defines a structured branching model:
  - main: production-ready code
  - develop: integration branch for upcoming releases
  - feature/*: new features branched from develop
  - release/*: prepare a version, stabilize code
  - hotfix/*: urgent fixes branched from main
- Pros:
  - Clear separation of work
  - Fits enterprise release cycles
- Cons:
  - More complex for beginners
  - Slows down fast-moving teams



Common Workflows

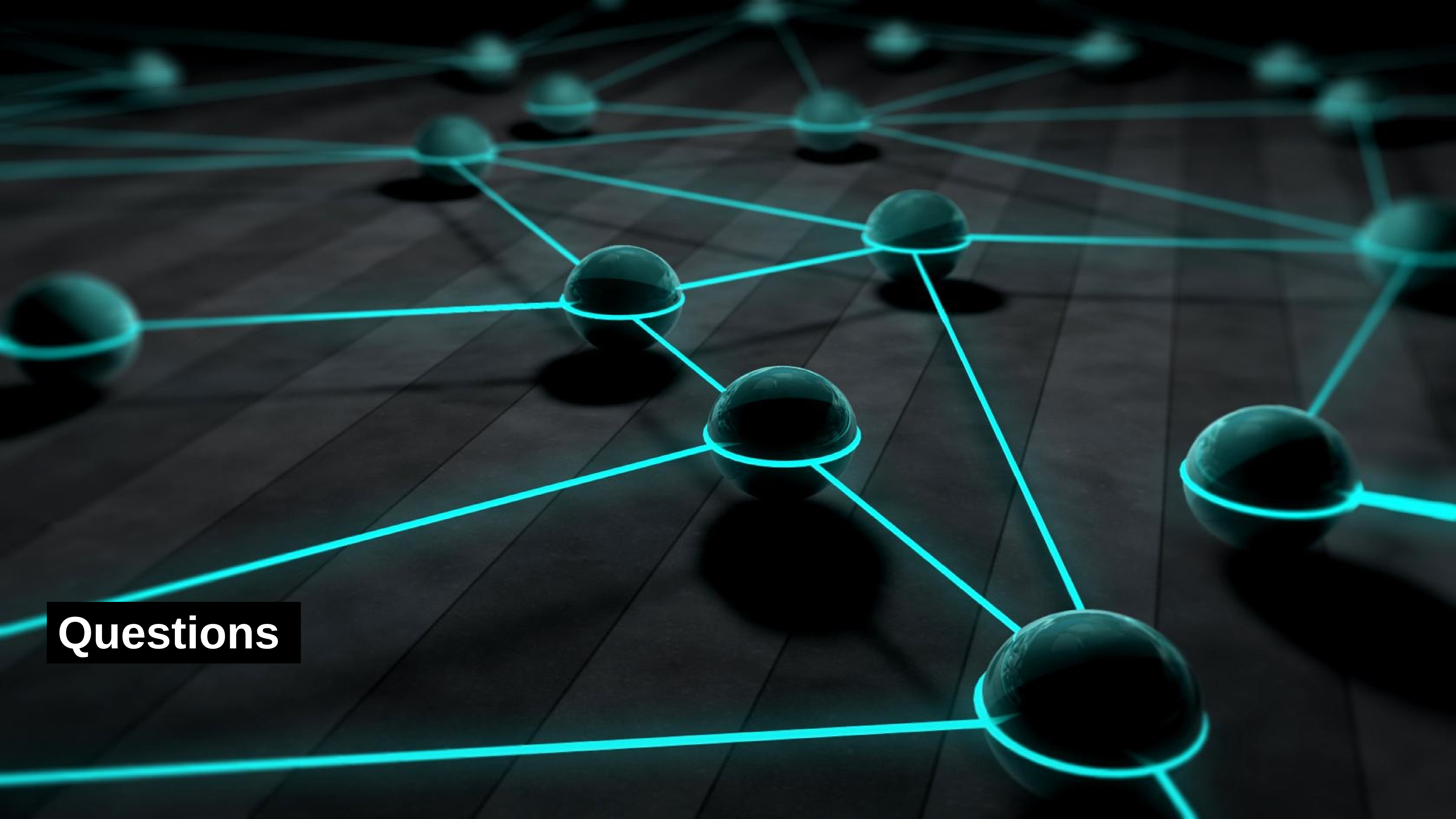
- Forking Workflow
  - Each contributor forks the repository into their own account
  - They push changes to their fork, then open a Pull Request against the main repo
- Pros:
  - Strong isolation → no one touches the official repo directly
  - Standard in open source projects
- Cons:
  - More overhead for setup
  - Slower for tightly-knit teams

Developing a Workflow Strategy

- Define a Branching Strategy
  - Every team member must follow the same rules
  - Example: all new work happens on feature/* branches, main is always deployable
- Use Pull/Merge Requests
  - Open a PR/MR before merging into main
  - Benefits
    - Code review
    - Automated checks (linting, tests)
    - Clear discussion history

Developing a Workflow Strategy

- Protect Important Branches
  - Protect main (and sometimes develop) with rules:
  - Require at least 1 or 2 approvals before merging
  - Require successful CI checks before merge
  - Disallow force-pushes
- Integrate CI/CD Pipelines
  - Every commit should:
    - Be tested automatically
    - Possibly trigger builds, deployments, or staging environment updates
  - Ensures stable code reaches production


Questions