Power Your Workflow With Git

Patrick Hogan
@pbhogan

Credit: **GitHub Octocat**, http://octodex.github.com/

Credits

Scott Chacon

Vincent Driessen

Benjamin Sandofsky

Before I begin, I want to give special credit to these guys:
Scott Chacon (Cha-kone) for much of the Git Internals content. He even sent me his slide deck.
Vincent and **Benjamin** for their ideas on branching and workflow.

Patrick Hogan

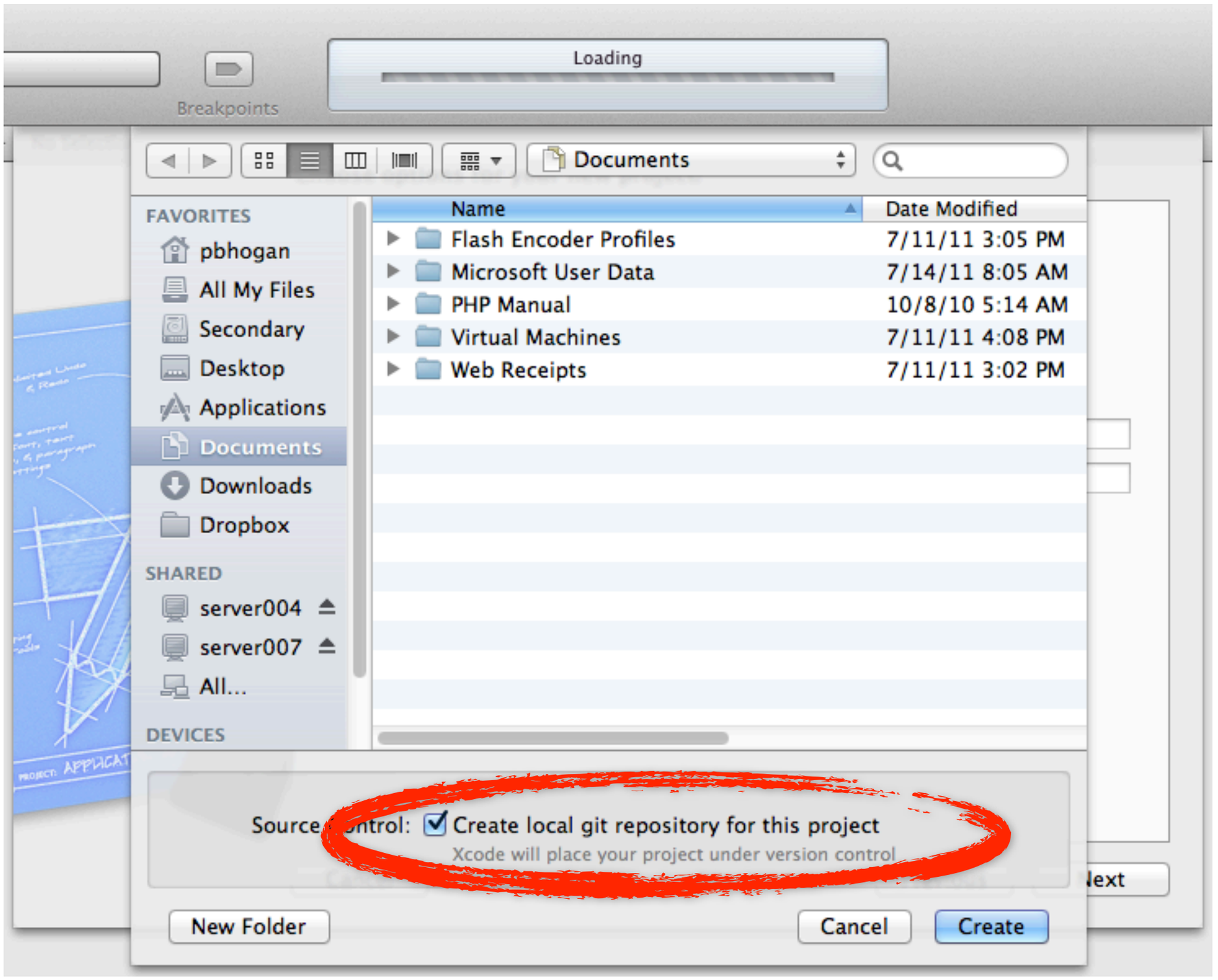
My name is Patrick Hogan

I'm the founder of Gallant Games

I have a game in the App Store called Swivel.

Why This Talk?

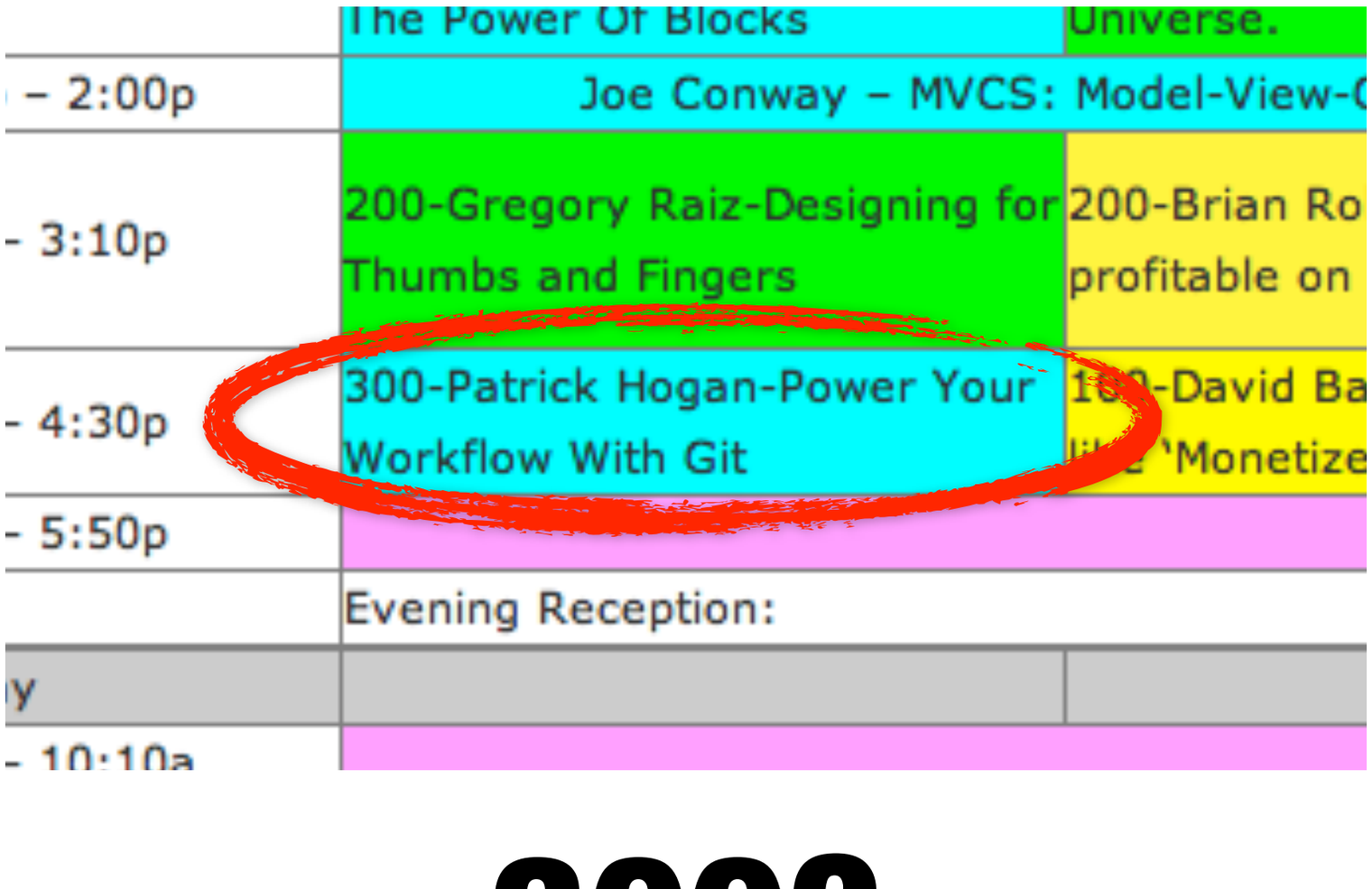
I believe we are all creatives.
Whether you're a developer, designer or artist, you're passionate about creating.
To create you need tools. If your tools are frustrating they get in the way of your passion.
I'm passionate about creating, so I'm fanatical about elegant tools.
Elegant tools either help you or get out of your way.
I believe Git is an elegant tool.



Plus... Xcode uses Git for new projects by default, so you need to know about it.

Approach

The first part of this talk will be a technical deep dive. I believe that to truly understand how to use Git, you have to know what Git is doing and how it thinks about your project. This demystifies a lot of the complexity and makes getting into Git a lot less scary.
Once we get through the internals, we'll examine workflows and how Git's model can help you individually and your team as a collective.



300?

So, this talk is classified 300 level.
I tried to figure out what that meant.
I asked Google, and it suggested...

...this.
So, strap in for a wild ride.

Version Control

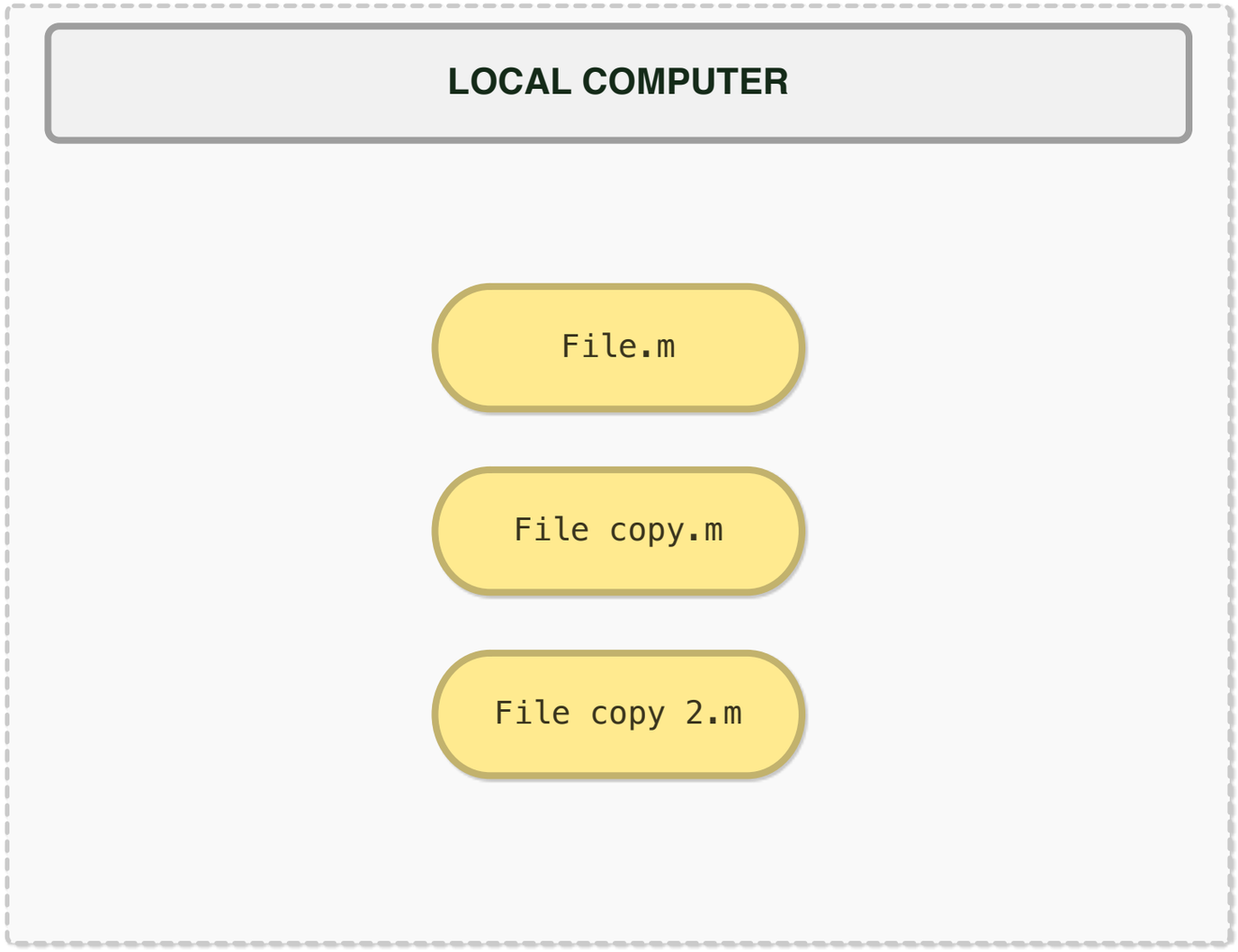
Workflow

For most people version control is a pain. At best it is a chore.
As a result they erect a wall between their workflow and version control.
Version control should inform your workflow, not hamper it.

History

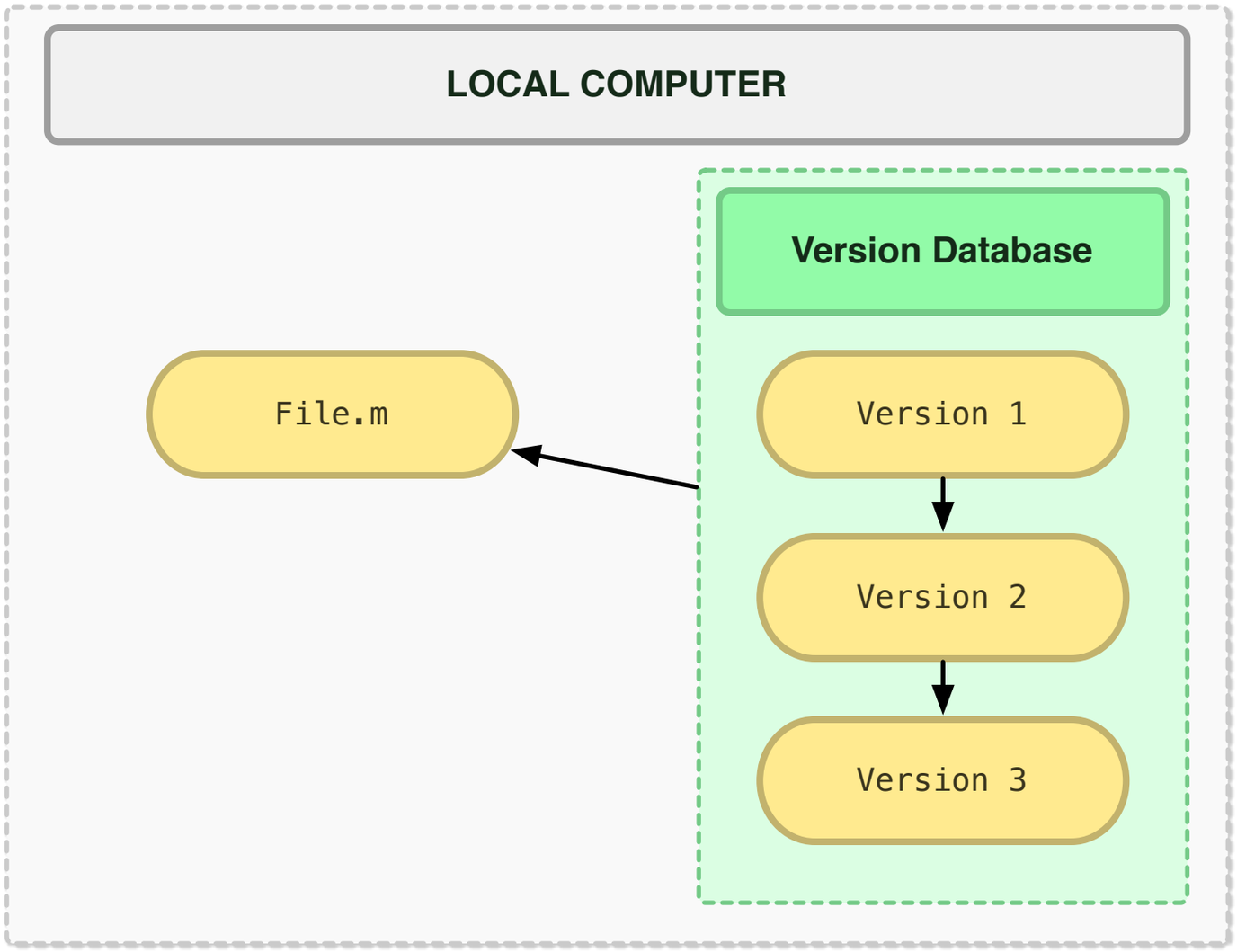
Let’s go back and see the progression.

Local Filesystem



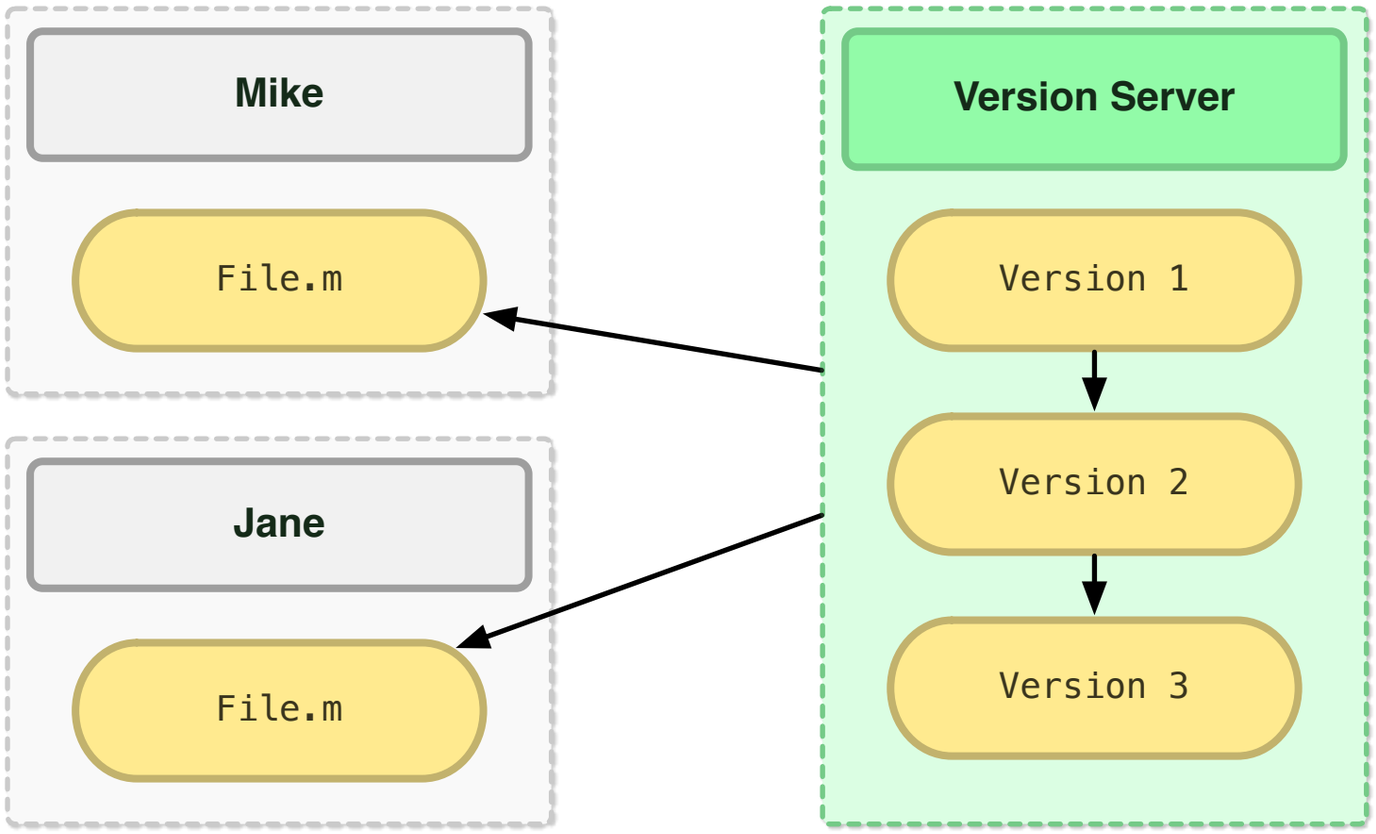
So, we've all done this: right-click, duplicate
It starts to get unmanageable really fast.
Whose ever gone back to their project and couldn't remember which file is the correct one, or maybe what bits you wanted to save for later, or even why?
Basically, we do this out of paranoia.

LOCAL
Version Control System



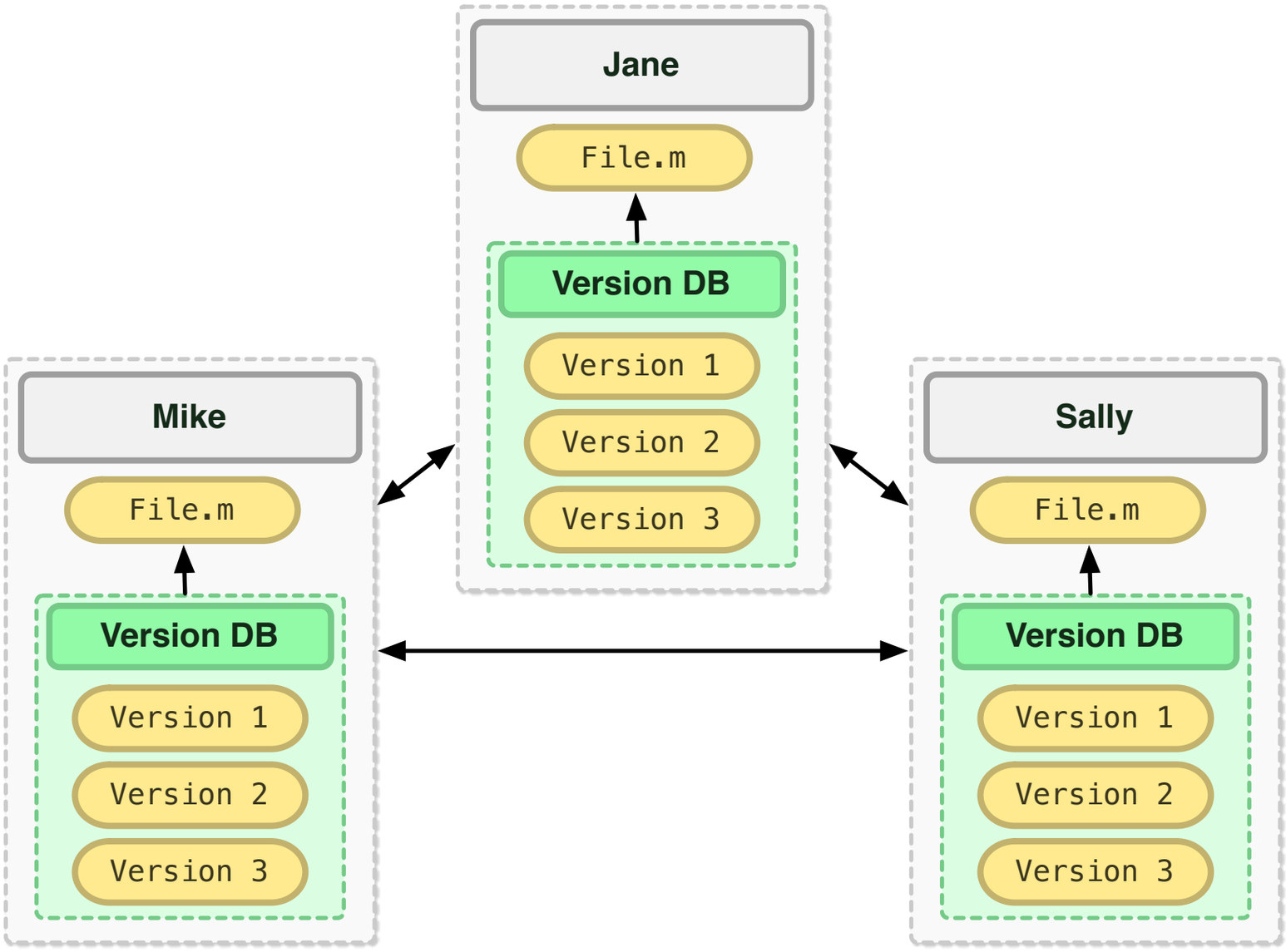
So, it didn’t take long for someone to come up with this.
RCS, one of the first LVCS, was released in 1982 (29 years ago!)
But if your computer crashes, it’s all gone.

CENTRALIZED
Version Control System



So, the natural progression was to store things on the server, and projects like CVS, SVN and others popped up. CVS in 1990, SVN in 2000
Problem: everything is on the server. You need access to the server and you'd better not lose the server.

DISTRIBUTED
Version Control System



DVCS came along to solve a lot of the problems with existing VCS.
Linux, for instance, switched to using BitKeeper to deal with their growth problems, and eventually switched to Git.
Both Git and Mercurial popped up in 2005.

DISTRIBUTED
Version Control System

In many ways DVCS solves a lot of problems. It retains the best aspects of both local and centralized VCS, and has several benefits on top of that.

Everything is Local

(Almost)

One huge benefit is that almost all version control operations happen locally -- aside from sync, and then only if sync is not between two local repositories.

No Network Required

Create Repo **Status**

Commit **Revisions**

Merge **Diff**

Branch **History**

Rebase **Bisect**

Tag **Local Sync**

None of these tasks require you to be connected to a server or any kind of network. You can be on a plane 30,000 ft up and do this stuff.

Advantages

Everything is Fast

Everything is Transparent

Every Clone is a Backup

You Can Work Offline

That means...
By transparent I mean you can literally inspect and see what Git is doing.

Storage

So there are two ways VCS store their data, and if you come from Subversion this might be hard for you to get used to at first.

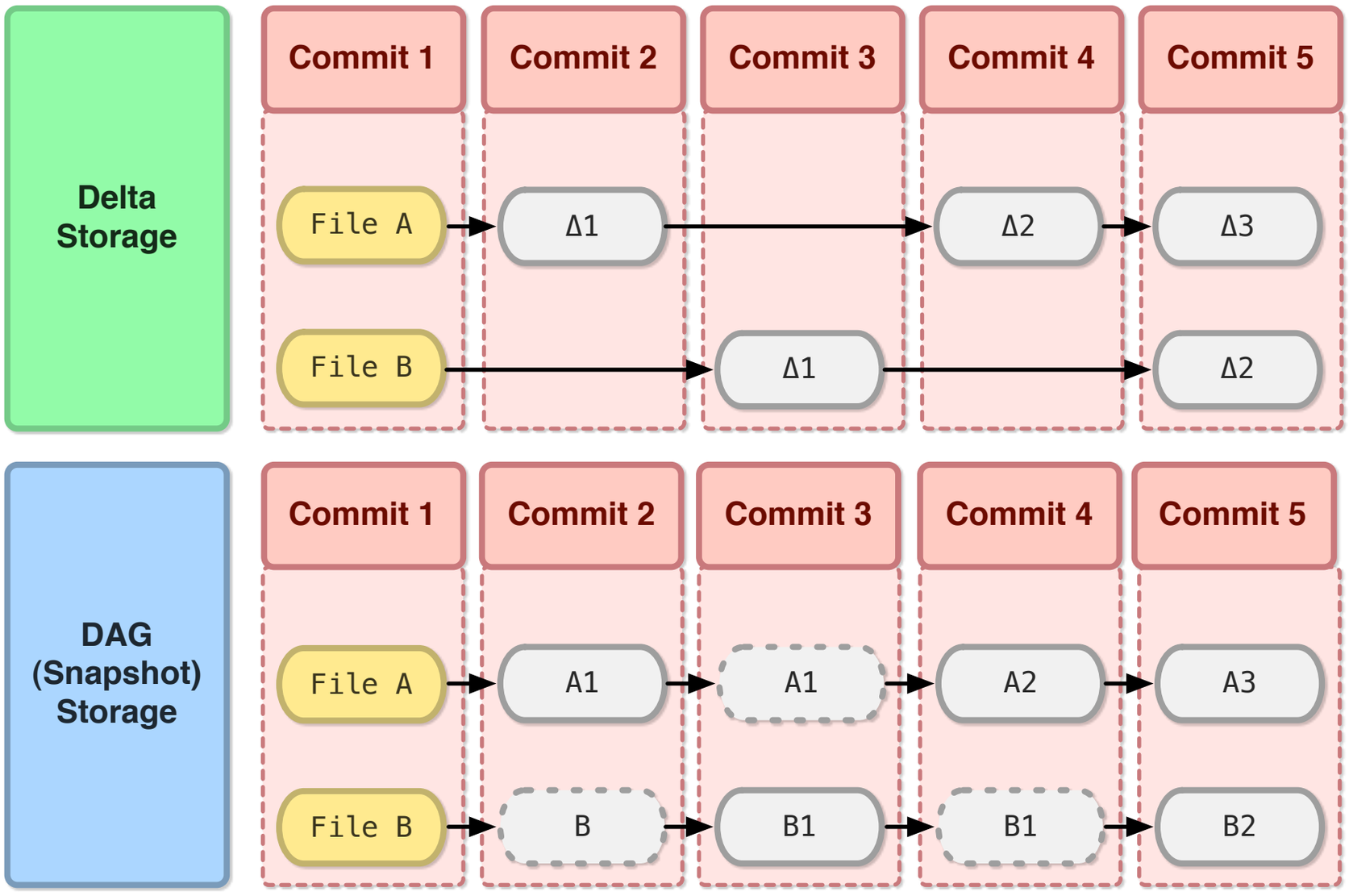
Delta Storage

Snapshot Storage

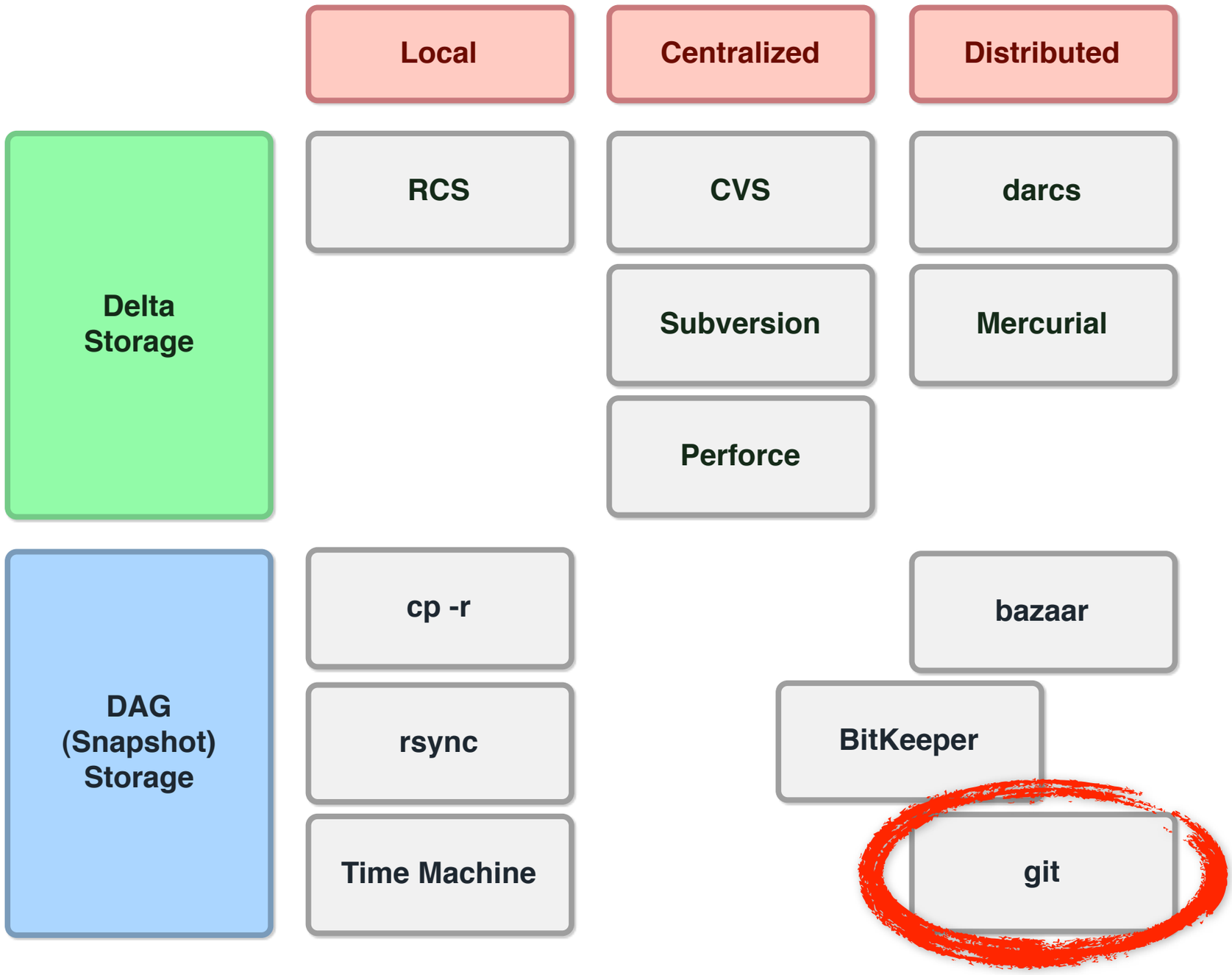
(a.k.a. Direct Acyclic Graph)

Direct Acyclic Graph just means a graph where if you follow the nodes from one node you can’t get back to the that node. Doesn’t really matter... just think of it as “snapshot” storage.

Delta vs. Snapshot



So the point is that with the snapshot model, each commit takes a full snapshot of your entire working directory. That might seem weird, but has some advantages and it can be done really efficiently. We'll see how when we get into the internals.
Also, this is kind of how we think as developers. Typically you commit when your codebase reaches a certain state regardless of which files you had to mess with.



Git is a distributed VCS that uses the snapshot storage model.

About Git

Linus Torvalds


Linux™

Linus on Git

http://bit.ly/linusongit

This is required viewing.

Git in a Nutshell

Free and Open Source

Distributed Version Control System

Designed to Handle Large Projects

Fast and Efficient

Excellent Branching and Merging

Projects Using Git

Git	Rails
Linux	Android
Perl	PostgreSQL
Eclipse	KDE
Qt	Gnome

Under The Hood

Git Directory

```
$ ls -lA
-rw-r--r--@  1 pbhogan  staff  21508 Jul  3 15:21 .DS_Store
drwxr-xr-x  14 pbhogan  staff    476 Jul  3 14:56 .git
-rw-r--r--@  1 pbhogan  staff    115 Aug 11  2010 .gitignore
-rw-r--r--@  1 pbhogan  staff    439 Dec 27  2010 Info.plist
drwxr-xr-x  17 pbhogan  staff    578 Feb  6 10:54 Resources
drwxr-xr-x   7 pbhogan  staff    238 Jul 18  2010 Source
...
```

Git Directory

```
$ tree .git
.git
├── HEAD
├── config
├── description
├── hooks
│   ├── post-commit.sample
│   └── ...
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

.git only in root of Working Directory (unlike Subversion)

Git Directory

Configuration File

Hooks

Object Database

References

Index

Git Directory

~~Configuration File~~

~~Hooks~~

Object Database

References

Index

Not going to discuss these here. You might never even touch these.

Object Database

The Object Database is where a lot of the Git magic happens.
It’s actually extremely simple. The approach Git takes is to have a really simple data model and then doing really smart things with.

Object Database

≈ NSDictionary

(Hash Table / Key-Value Store)

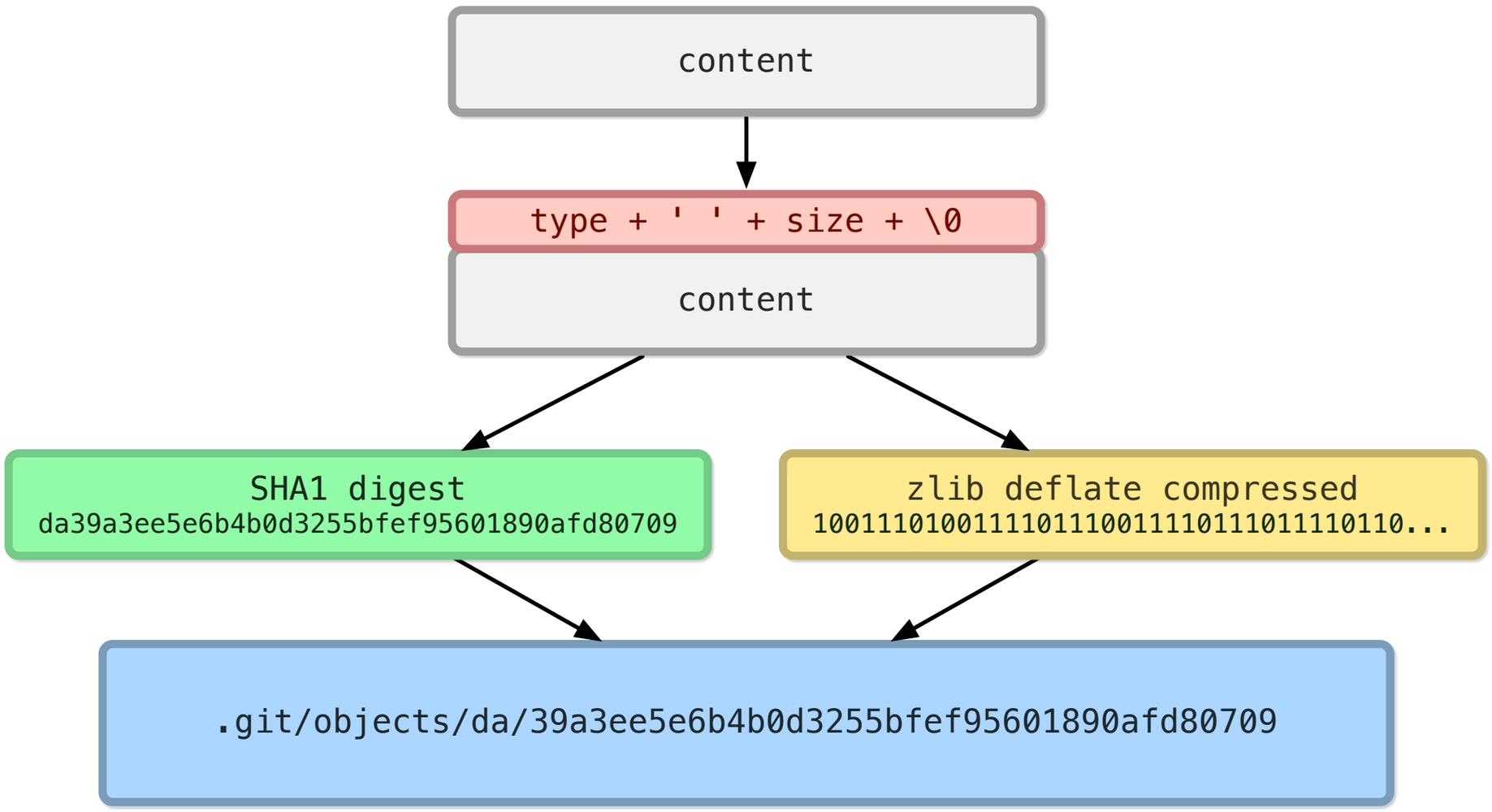
It’s really nothing more than a glorified on-disk NSDictionary -- or a hash table, if you like.

Object Database

```
content
```

Ultimately it all comes down to storing a bit of content. We'll talk about what that content is later, but given a piece of content...

Object Database

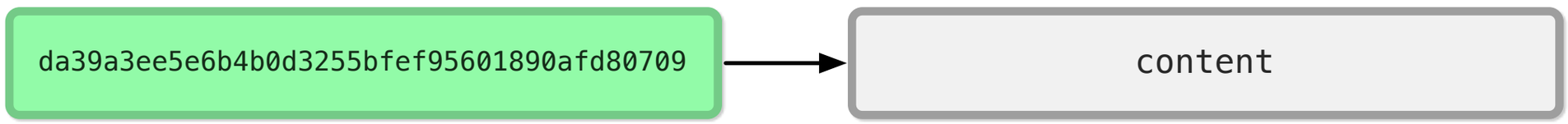


"loose format"

Git appends it to a header with a type, a space, the size and a null byte...
Calculates a hash (using SHA1 cryptographic hash)...
Compresses the content and header...
And writes it into a folder based on the hash.
This is referred to as being stored in loose format.

Object Database

≈ NSDictionary



Pointer / Key **Object / Value**

Again, this is like a key-value database on disk.
The hash is the key and the content is the value.
What's interesting is, because the key is a hash of the content, each bit of content in Git is kind of automatically cryptographically signed, and can be verified.

git cat-file -p da39a

Object Database

```
da39a3ee5e6b4b0d3255bfef95601890afd80709
da39a3ee5e6b4b0d3255...
da39a3ee5e6...
da39a...
```

Equivalent if common prefix is unique!

What's cool is Git considers any first part of the hash a valid key if it is unique so you don't have to keep using a 40 character string.
In fact, that's more or less what I'm going to do for the rest of this talk so it all first on the slides. :)

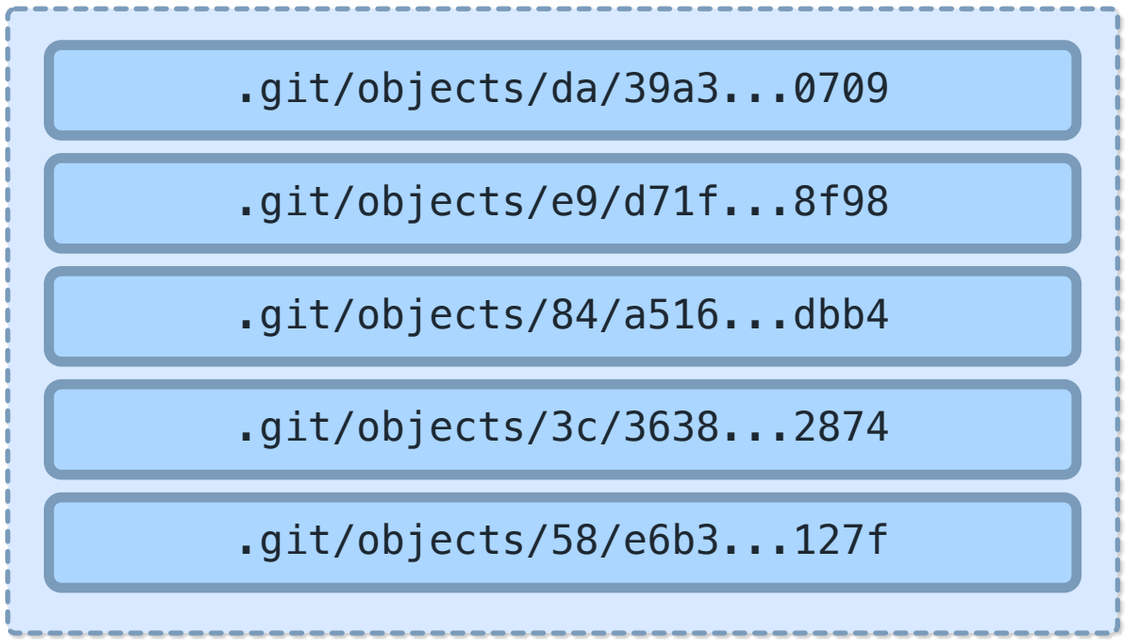
Object Database

Garbage Collection

Git has one more trick up it’s sleeve to keep things efficient.
On certain operations, or on demand, Git will garbage collect, or really optimize the database.

Object Database

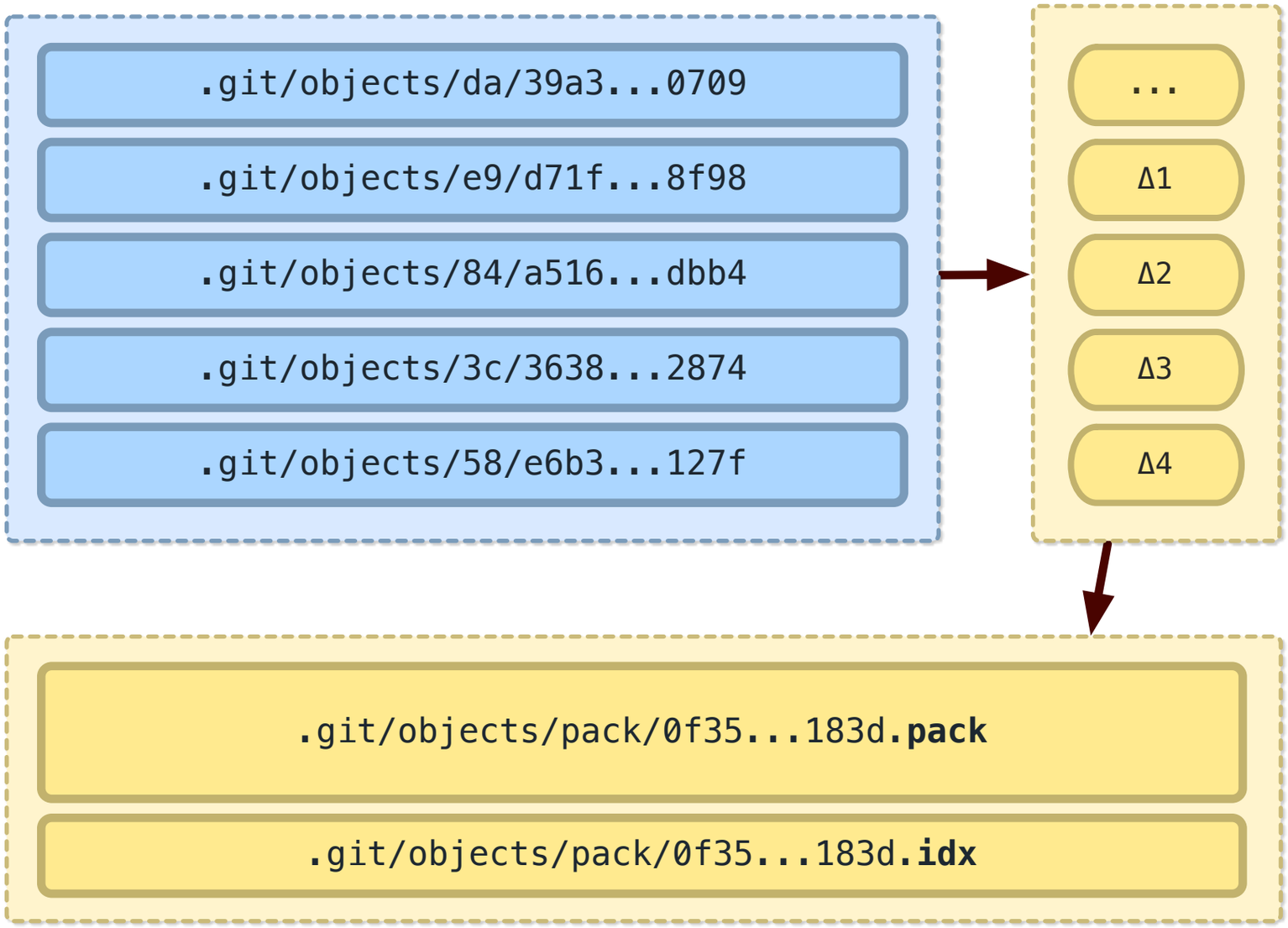
Similar Objects



`git gc`

So if Git knows it has certain similar objects (maybe versions of files, but can be anything really) and you run git gc...

Object Database



"packed format"

It'll calculate deltas between those objects, and save them into a pack file and an index. This is referred to as being stored in packed format.

Object Database

Four Object Types

The Object database has four data types (recall that it stores the type in the header)...

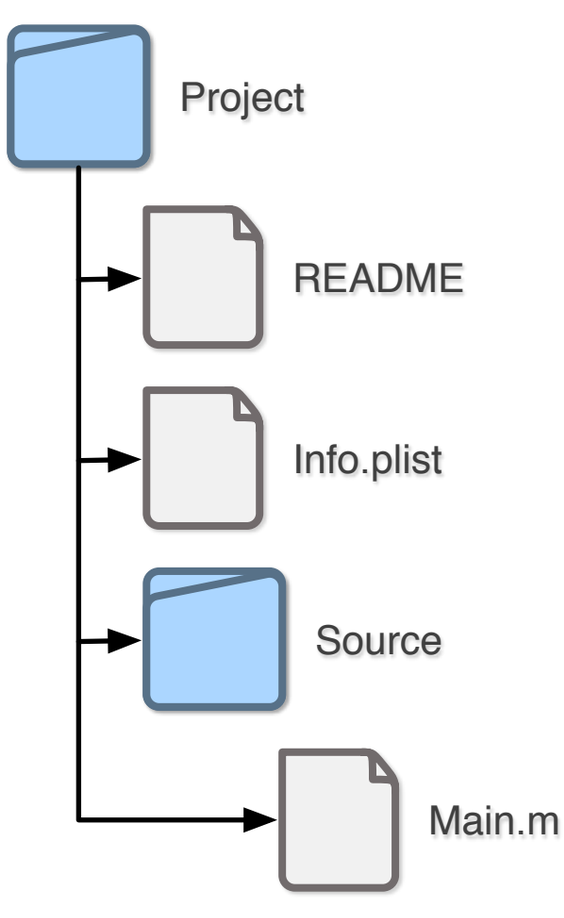
Object Database

Blob... Tree... Commit... Tag...
First up... blobs

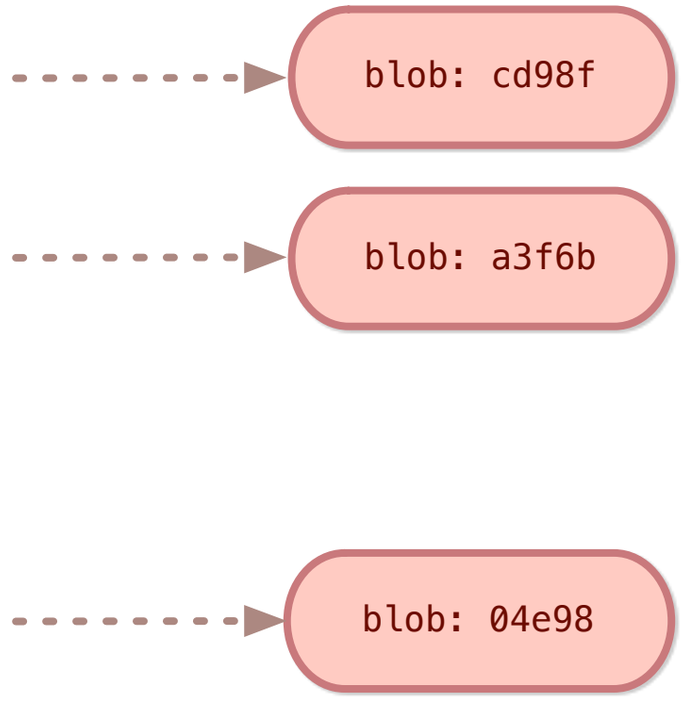
Object Database

blob

Working Directory



Git Directory



Blobs essentially correspond to files.

Object Database

blob

```
blob 109\0
```

```
#import <Cocoa/Cocoa.h>

int main(int argc, const char *argv[])
{
    return NSApplicationMain(argc, argv);
}
```

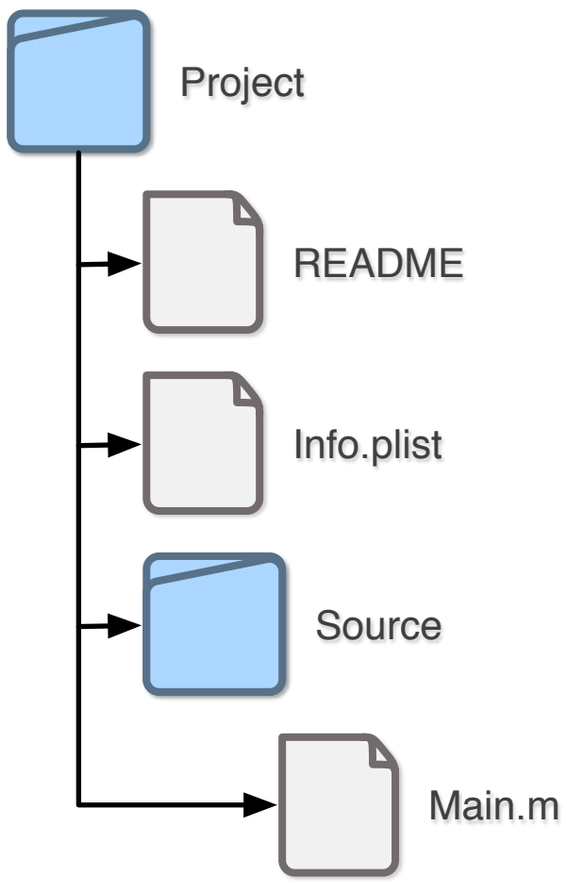
This is how it's stored, SHA1 hashed and compressed.
Keep in mind that the same content will always have the same hash, so multiple files or versions of files with the exact same content will only be stored once (and may even be delta packed).
So Git is able to be very efficient this way.

Object Database

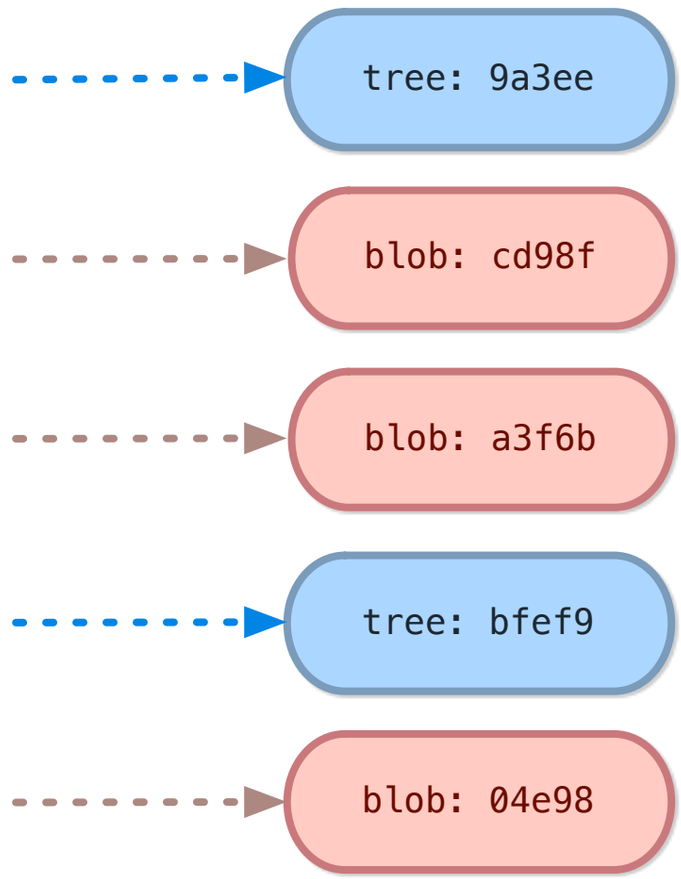
Next... trees

Object Database

Working Directory



Git Directory



Trees correspond to directories or folders...

Object Database

tree

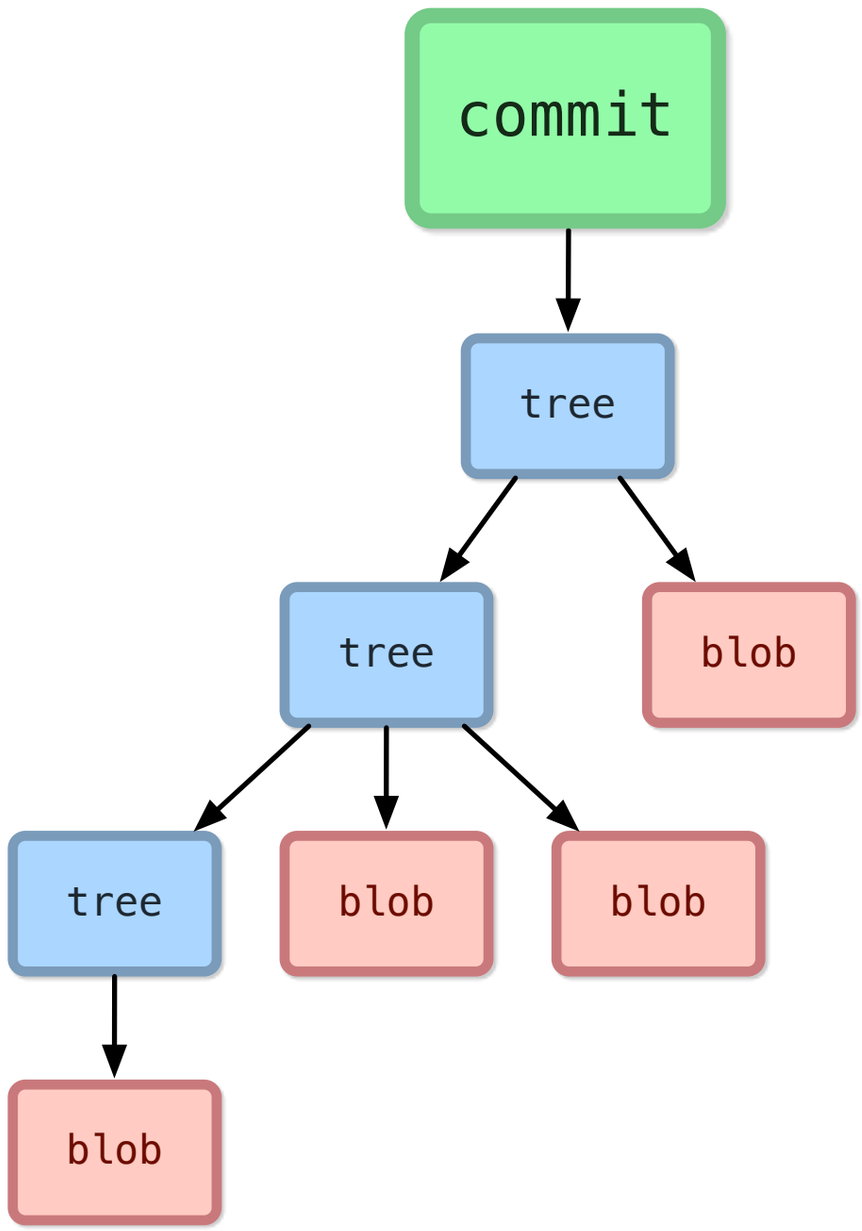
```
tree 84\0

100644 blob cd98f    README
100644 blob a3f6b    Info.plist
040000 tree bfef9    Source
```

In this case the content is a POSIX like directory list of files (blobs) along with their hashes and some posix information.
Given a tree, it's easy to find the files and other trees in it by just looking for the hashes in the object database.

Object Database

Next... commits

Object Database



A commit essentially just points to a tree (which is the pretty much the root of your working directory).
So here you can see the snapshot model in action. Given a snapshot you can follow it and get your entire project -- all the files and folders -- and extract them from the database just by following the hashes.

Object Database

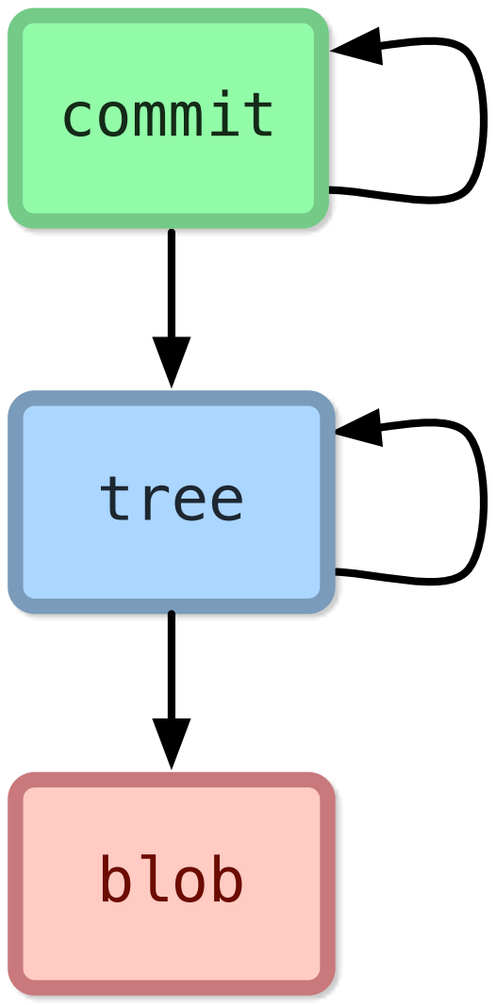
commit

```
commit 155\0
```

```
tree 9a3ee
parent fb39e
author Patrick Hogan <pbhogan@gmail.com> 1311810904
committer Patrick Hogan <pbhogan@gmail.com> 1311810904

Fixed a typo in README.
```

Header...
Type, Hash...
Parent commits (0 or more) -- 0 if first, 1 for normal commit, 2 or more if merge
Author, Committer, Date
Message

Object Database

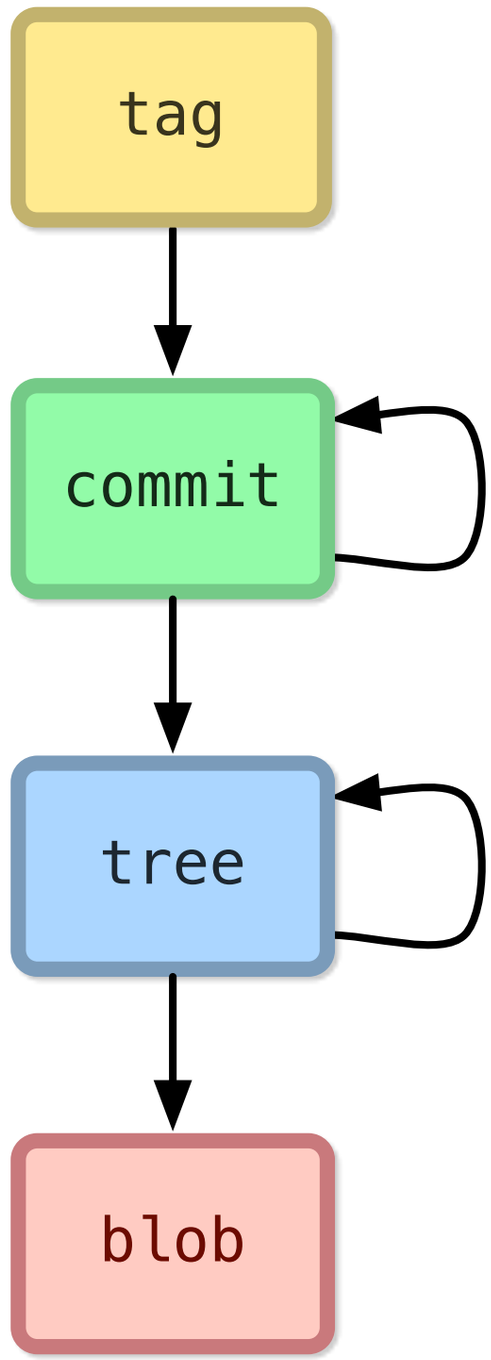


Direct Acyclic Graph

Object Database

And finally... tags

Object Database



Tags just point to a commit.
It’s really just a kind of named pointer to a commit since all commits are named by their hash.

Object Database

tag

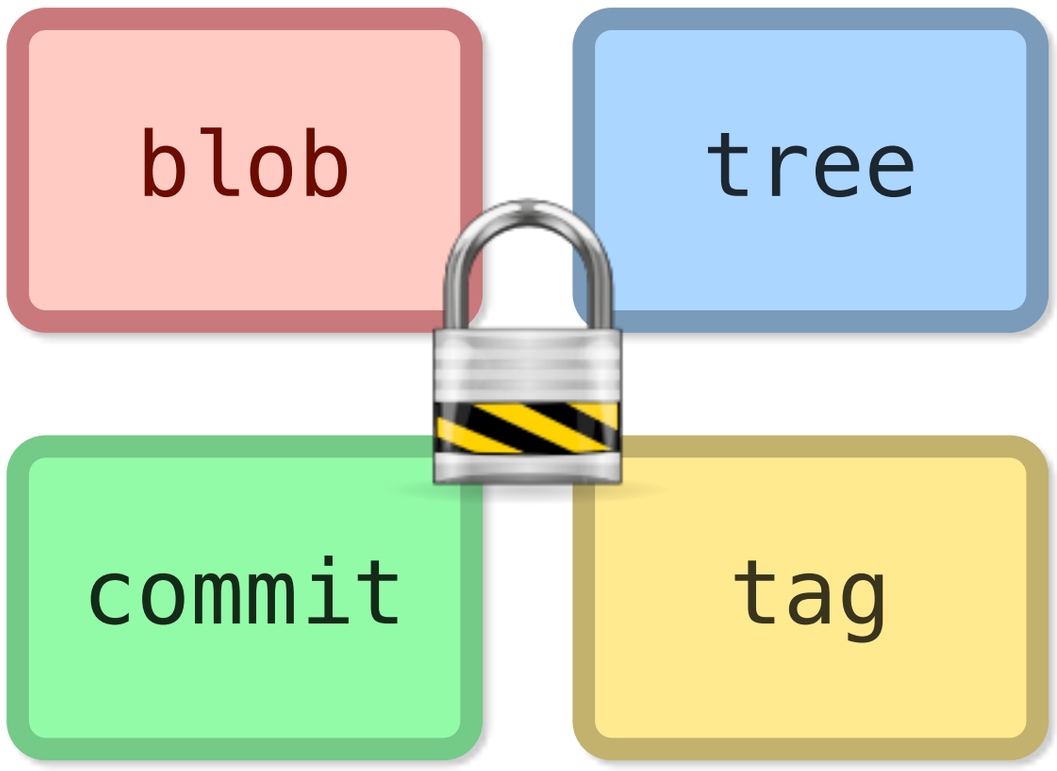
```
tag 121\0
```

```
object e4d23e
type commit
tag v1.2.0
tagger Patrick Hogan <pbhogan@gmail.com> 1311810904

Version 1.2 release -- FINALLY!
```

.git/objects/20/c71174453dc760692cd1461275bf0cffeb772f

.git/refs/tags/v1.2.0

Object Database



Immutable!

All of these objects are immutable. They cannot be changed.
Content for a given key is essentially cryptographically signed.

Never Removes Data

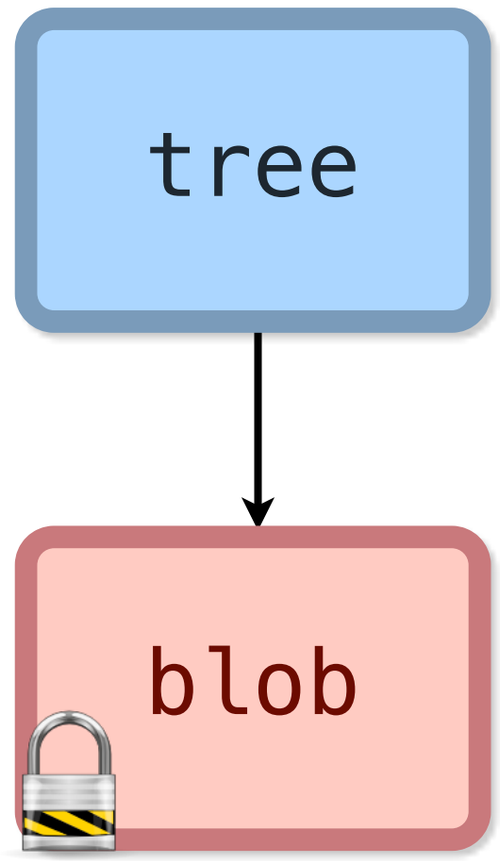
(Almost)

Once committed -- Git almost never removes data.
At least, Git will never remove data that is reachable in your history.
The only way things become unreachable is if you “rewrite history”
It’s actually very hard to lose data in Git.

"Rewriting History" Writes Alternate History

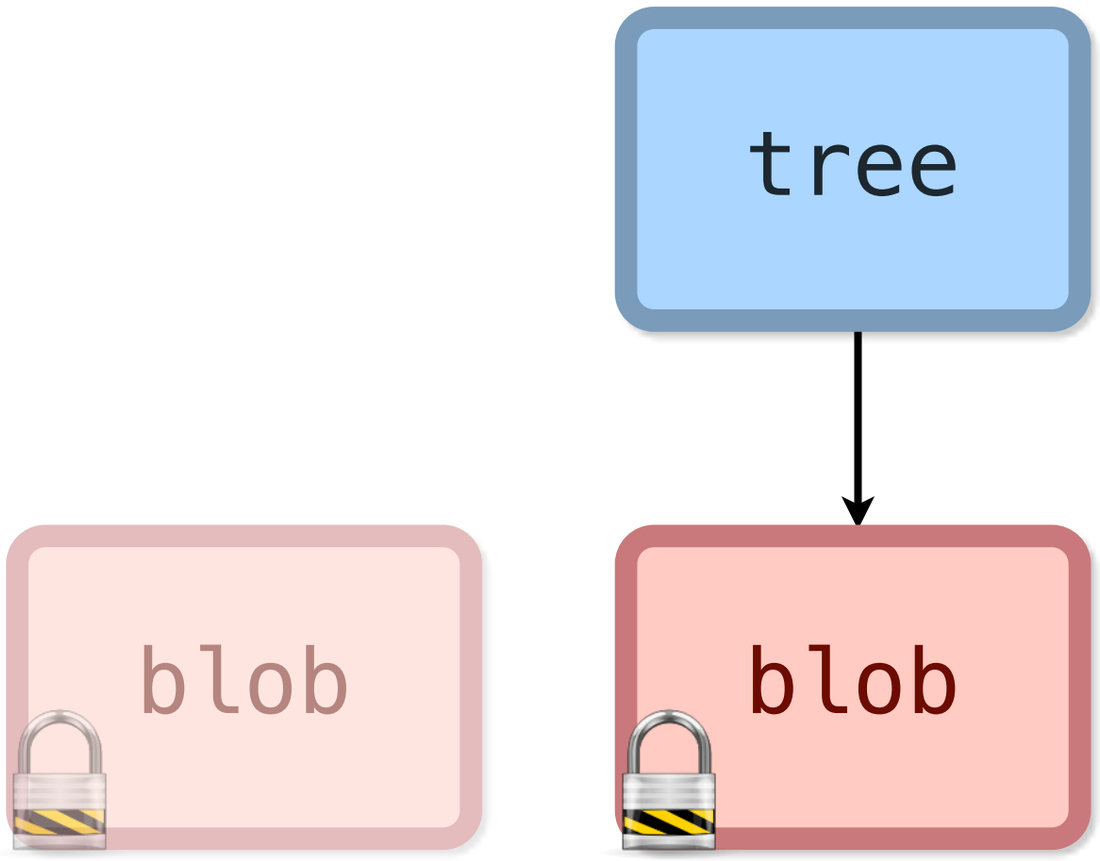
While you'll hear this phrase a lot, it actually isn't true.
Git doesn't rewrite history. It simply writes an alternate history and points to that.
git commit --amend, git commit --squash, git rebase

Object Database



So if we have a file, change it and we amend a commit...

Object Database



It keeps the old object, writes a new one and moves a pointer.
This is called an unreachable object.
These can be pruned and will not push to remotes.
This is really the only way Git will lose data.
And even then, you have to run git prune or equivalent.

Git Directory

Configuration File

Hooks

Object Database

References

Index

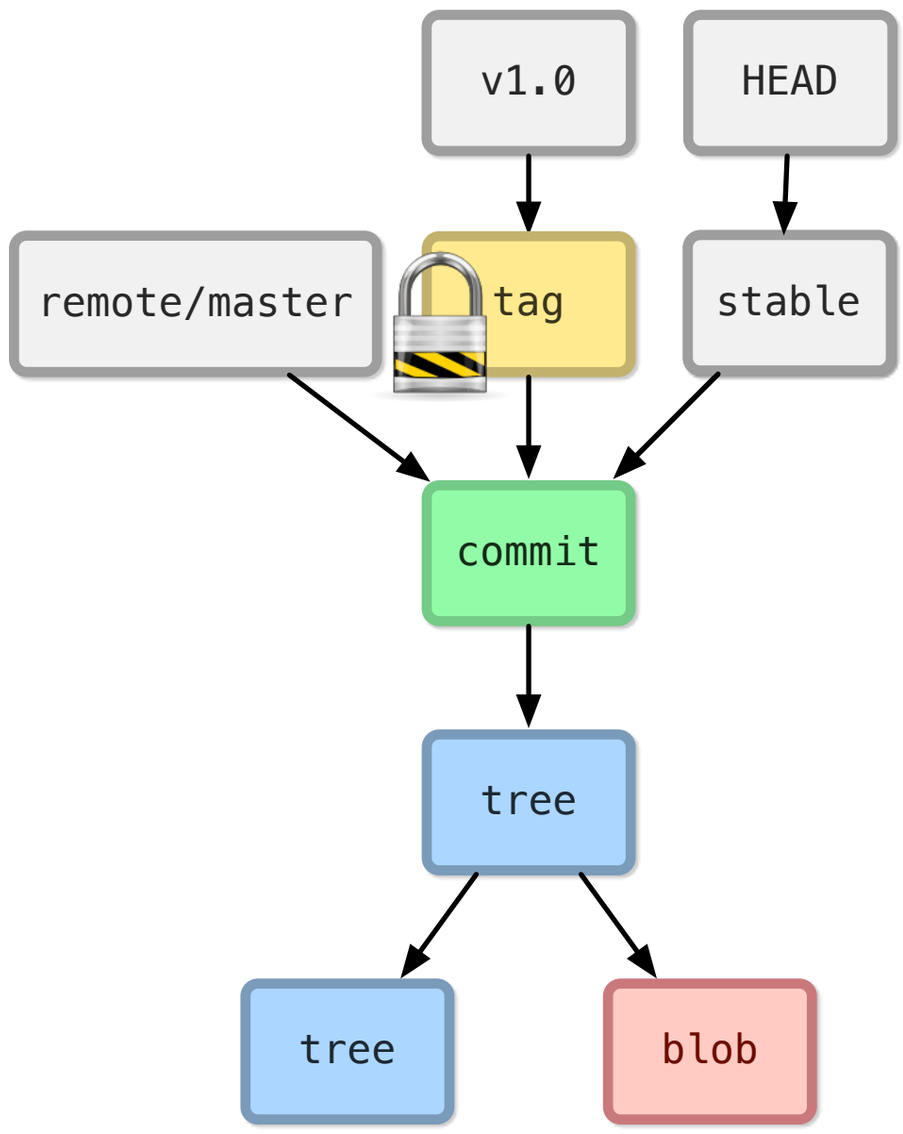
Next up... references

References

References

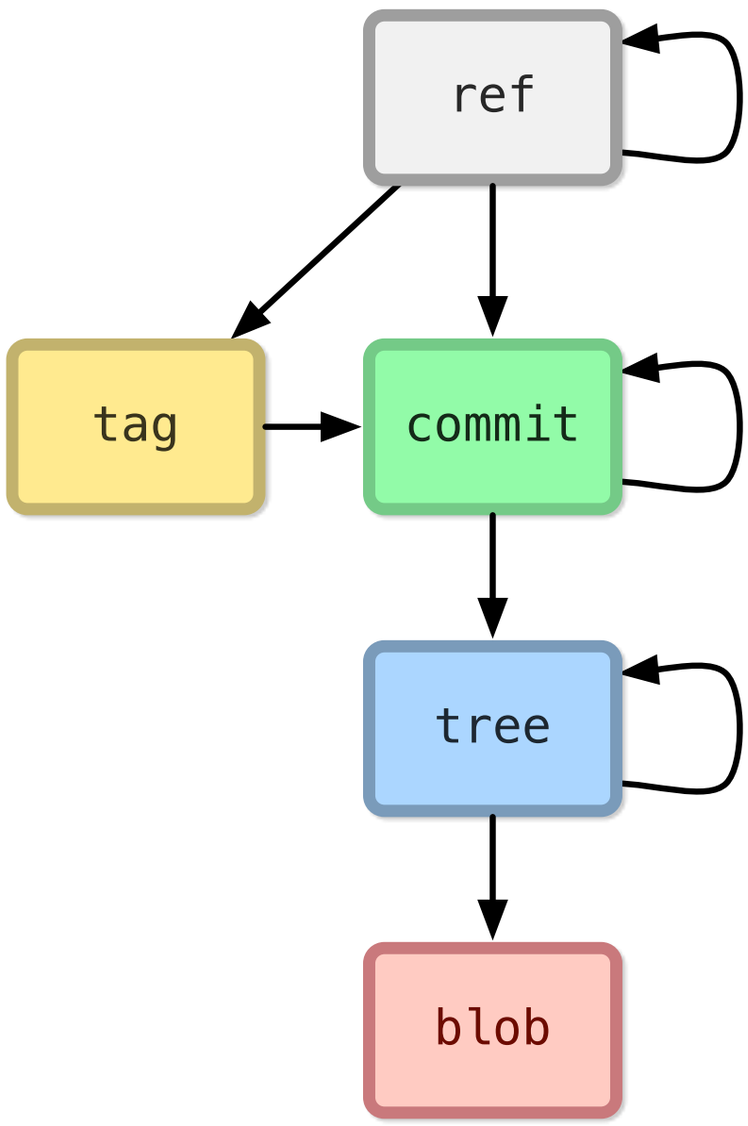
Lightweight, Movable Pointers to Commits (and other things)

References



Every git object is immutable, so a tag cannot be changed to point elsewhere.
But we need pointers that can change... so we have refs.
Refs are things like branch names (heads) which point to the latest commit in a given branch.
There's also HEAD (uppercase) which points exclusively to the latest commit of your currently active (checked out) branch. This is where Git operations will do their work.

References

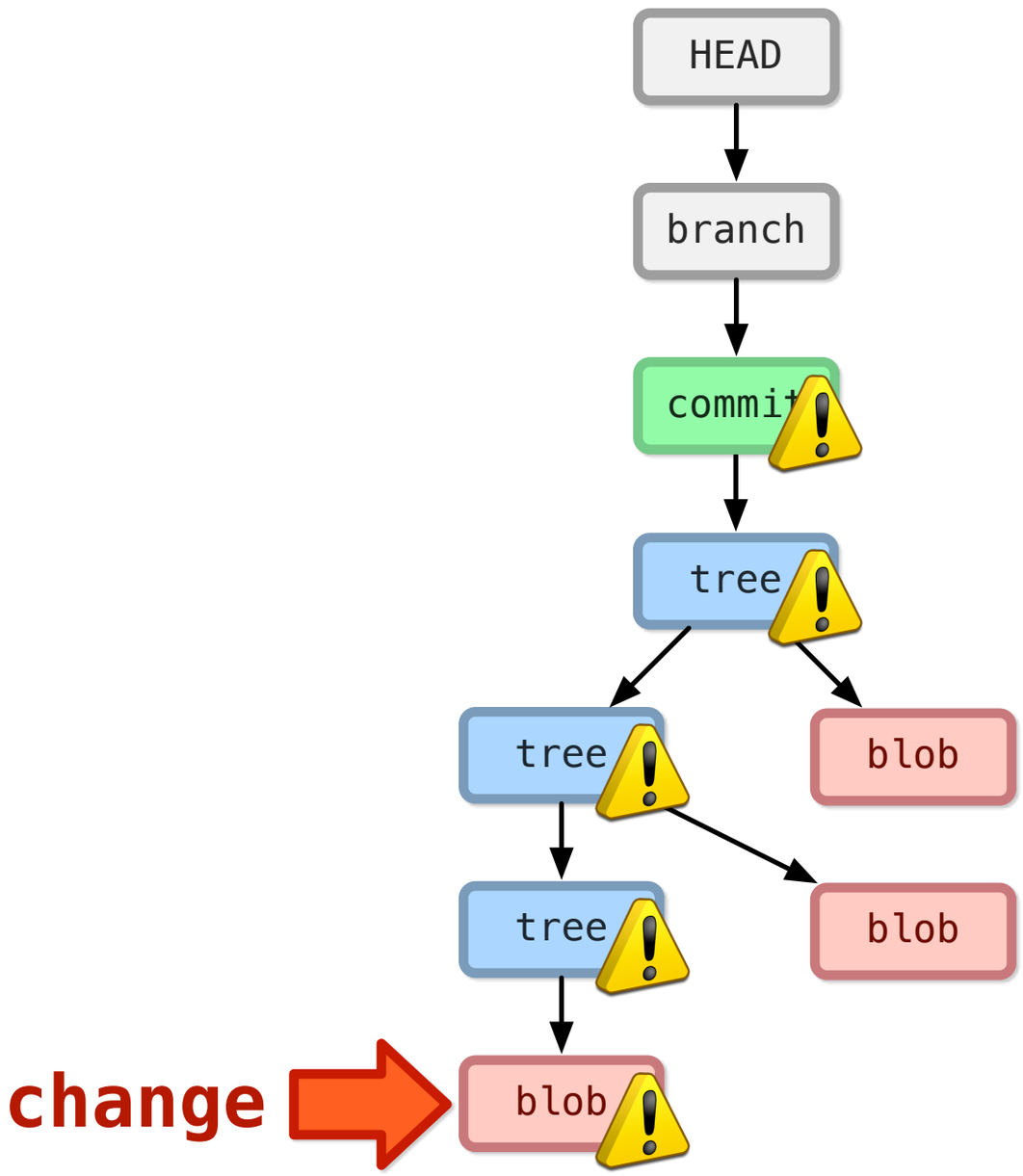


So here’s the whole Git object model.
Everything in Git operates within this framework.
It’s really simple, but it allows many complex operations.

Scenario

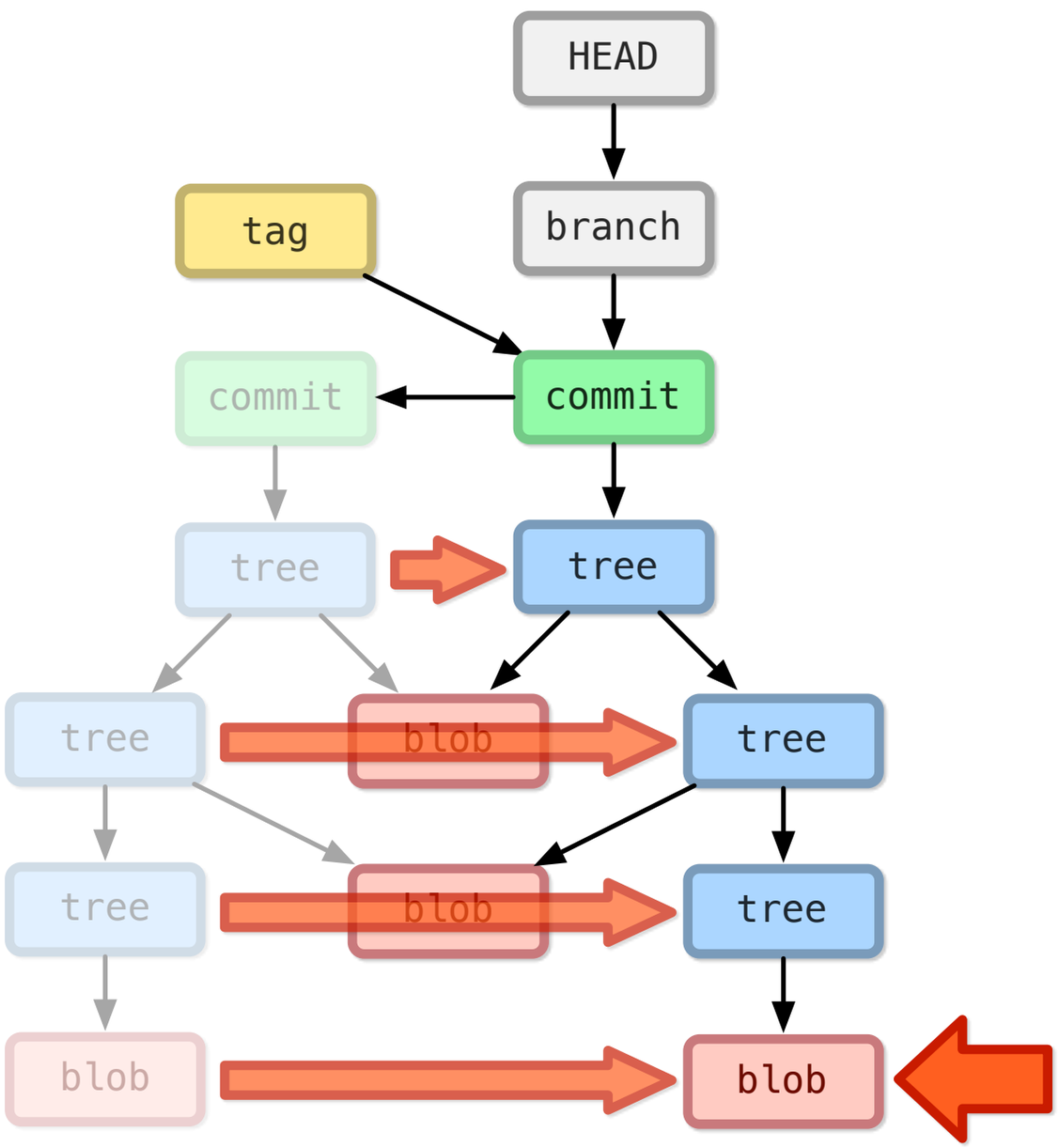
So let’s run through a scenario to see this in action.

Scenario



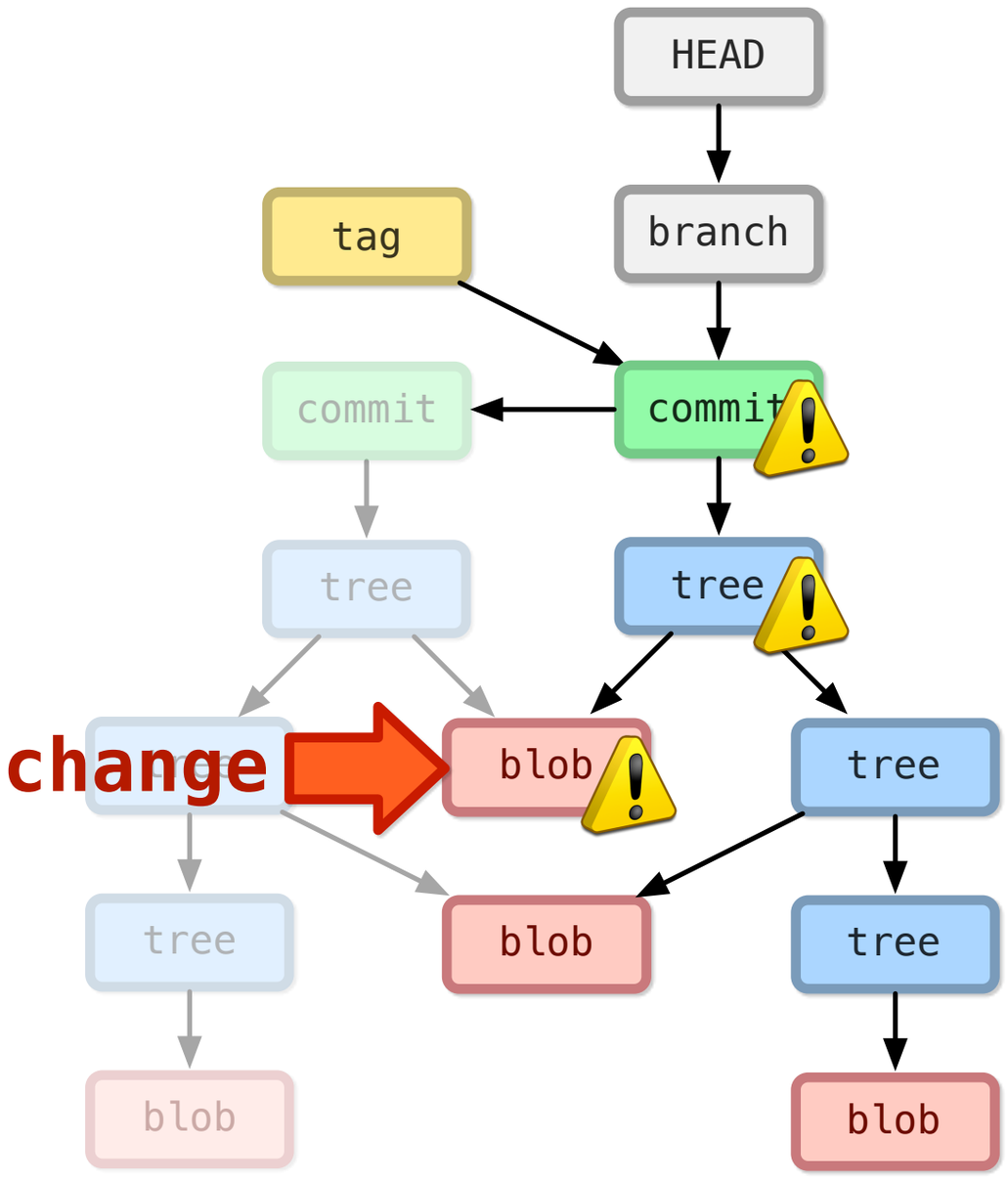
So here we have our first commit. It has a few directories and three files...
If we change this file at the bottom and commit...
All of these other objects need to change too, because it's parent tree points to it by hash and so on up the chain.
But all objects are immutable, so...

Scenario



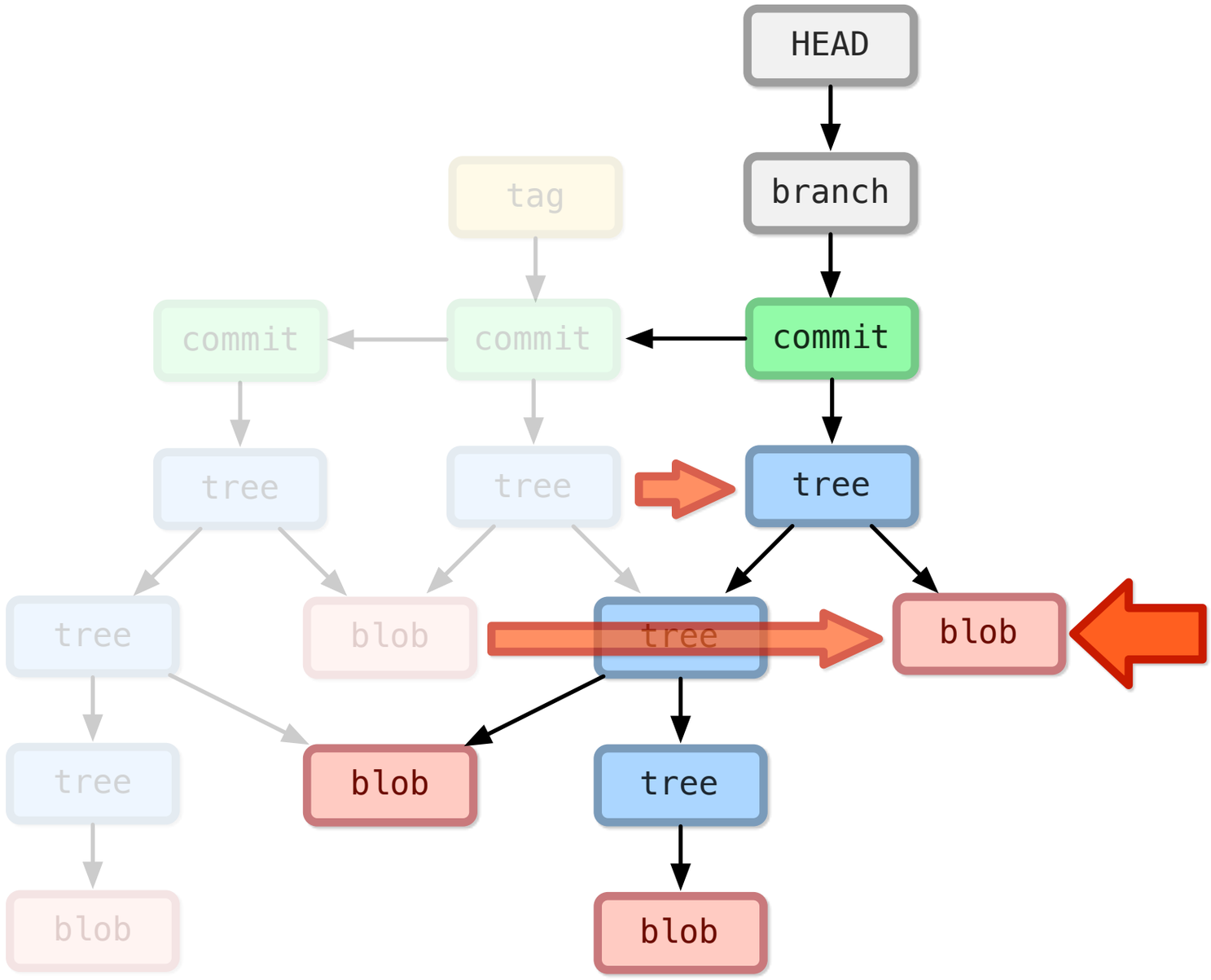
So git makes new objects with updated pointers...
It writes the new blob, then updates its parent up the chain...
Notice the commit points to its parent commit, and the two unchanged files are still pointed to...
The branch and head can change because they’re references...
Finally, we could tag this commit. Maybe it’s a release.

Scenario



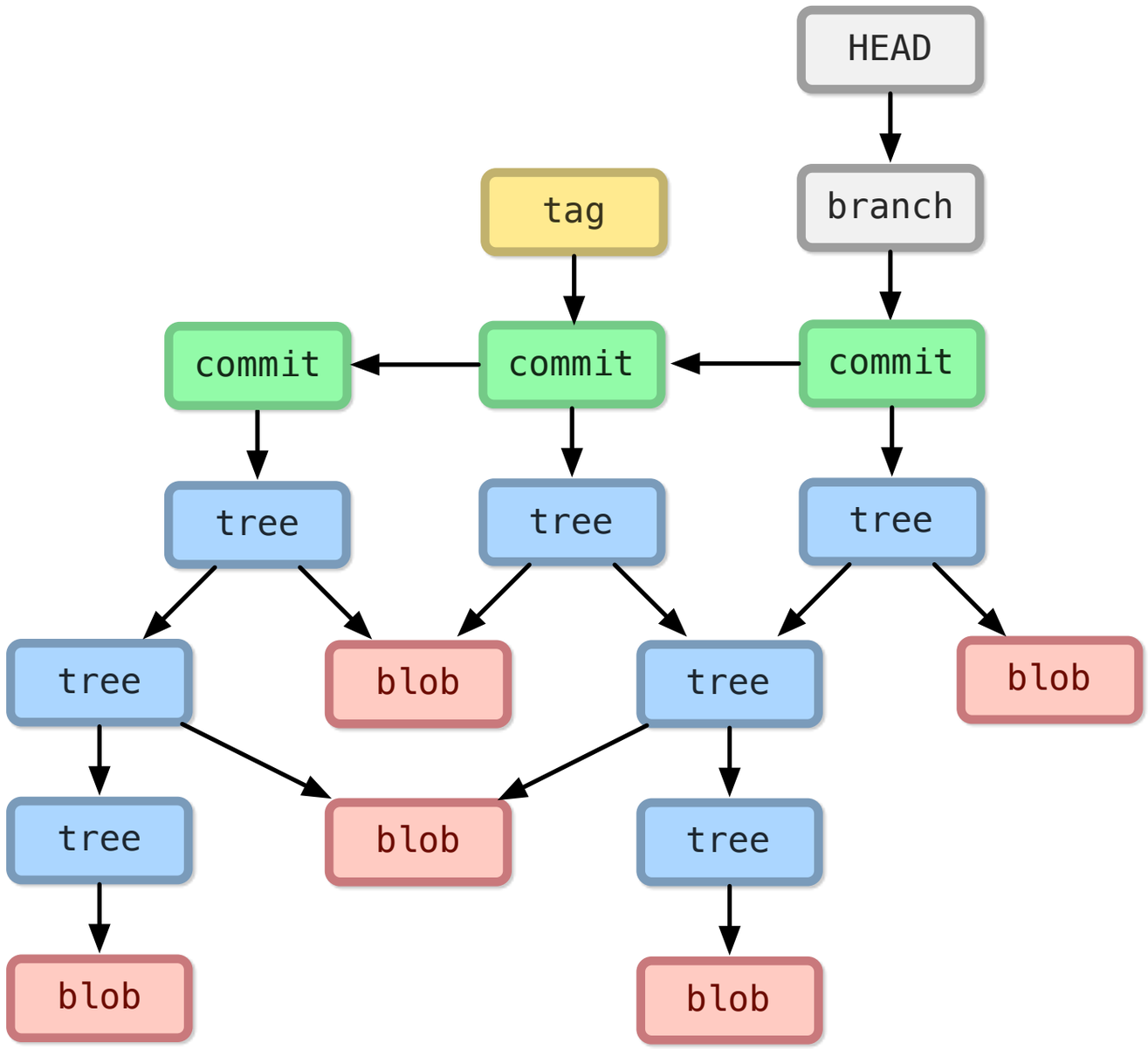
So if we change this top blob now and make a third commit, it affects all the nodes above it...

Scenario



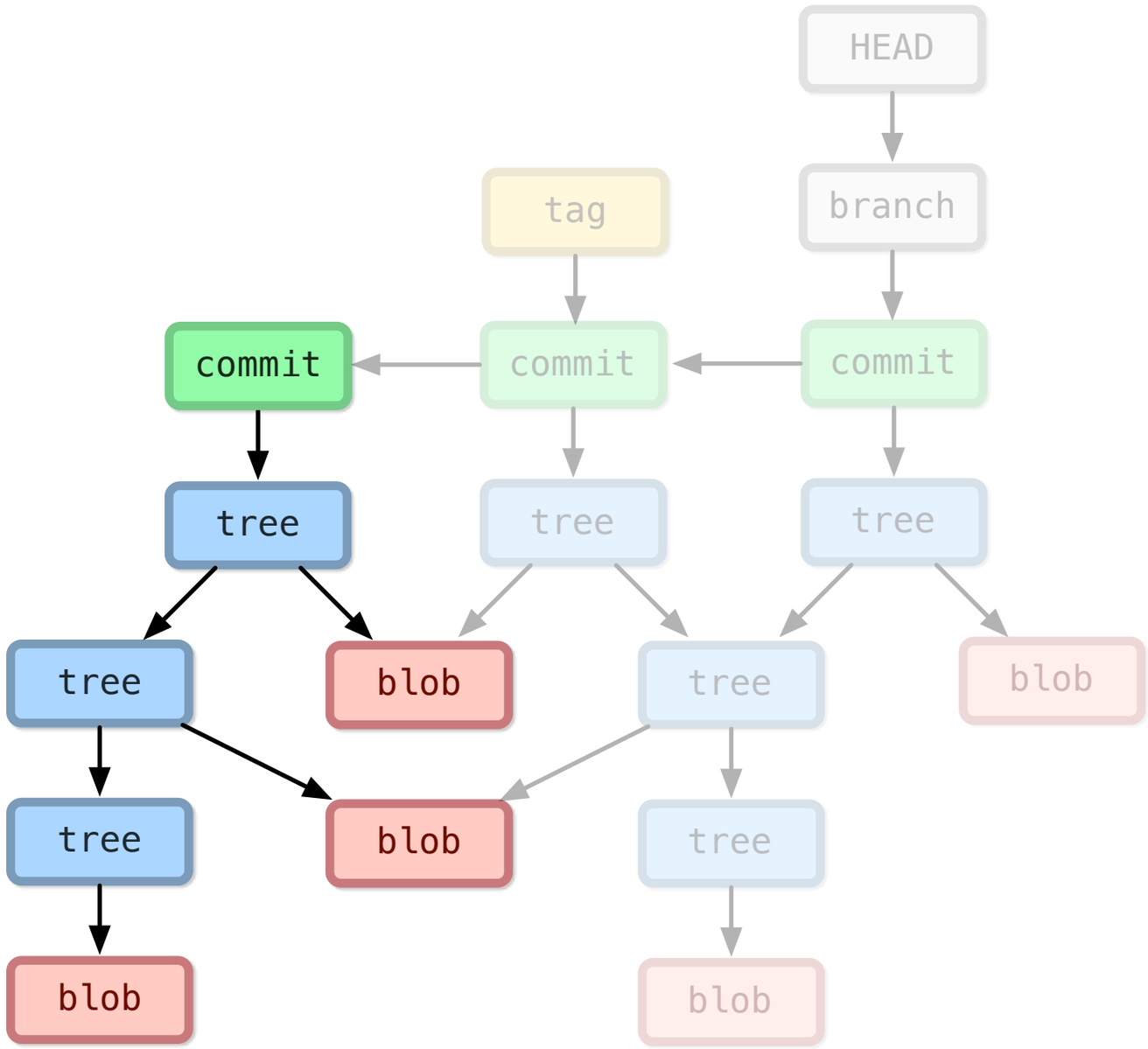
Again, Git makes new objects...
It writes a new blob...
And a new tree...
But not this tree...
And a new commit and moves the branch head.

Scenario



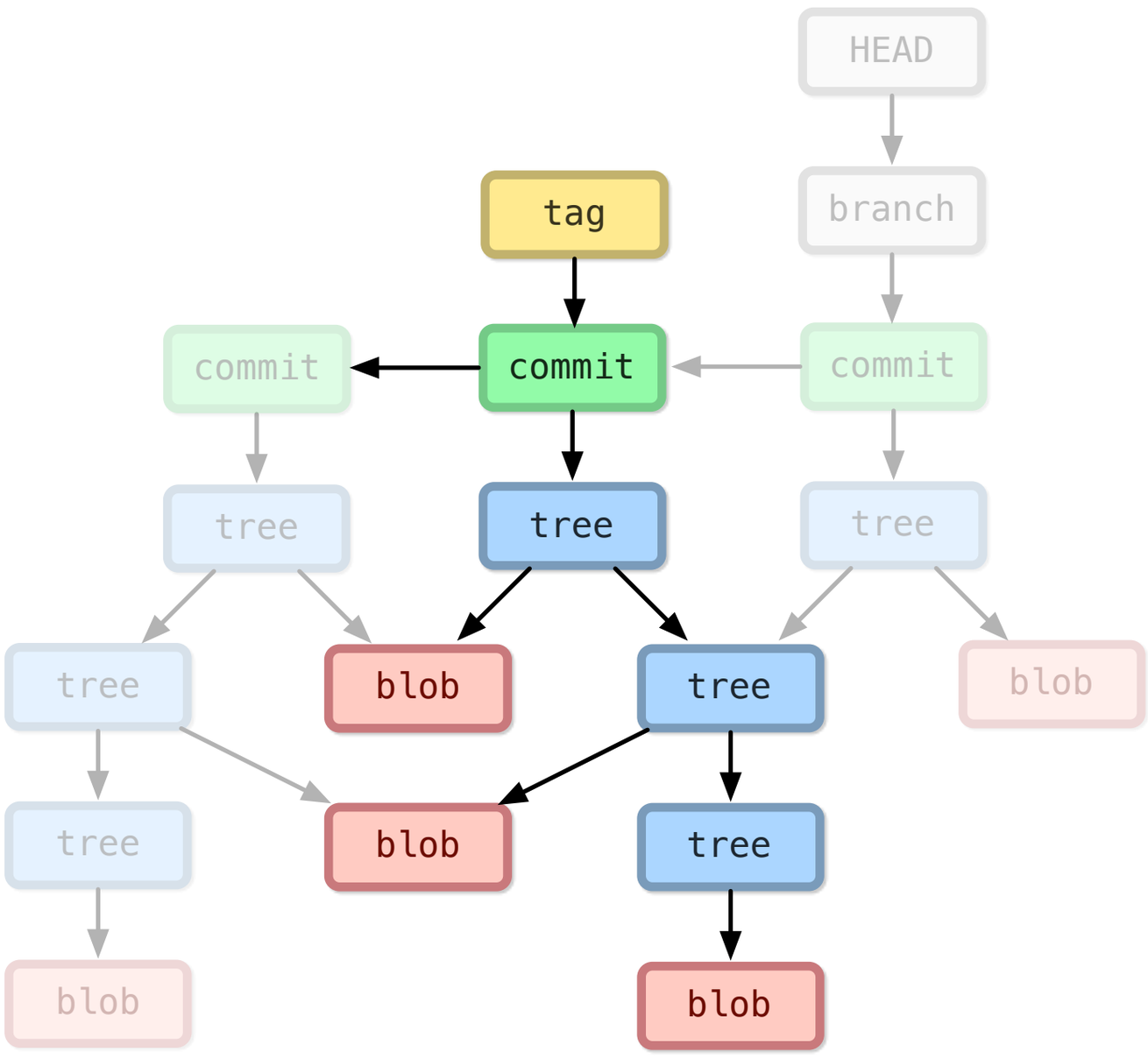
So that gives us this graph in our object database.
And we can pull out...

Scenario



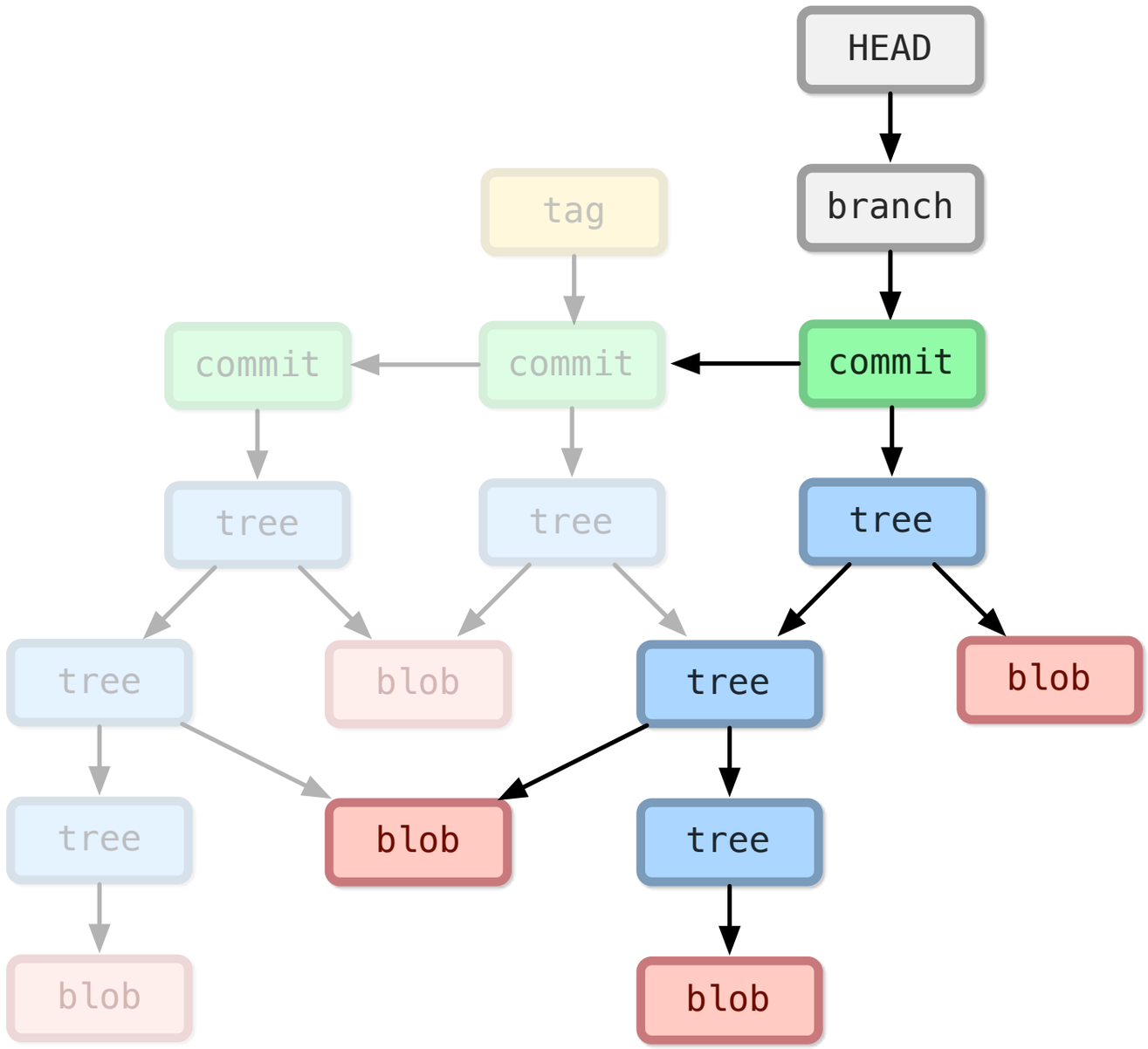
This commit, or...

Scenario



This commit, or...

Scenario



This commit.
And it’s all very fast because all Git needs to do is follow the links and extract the objects.

Git Directory

Configuration File

Hooks

Object Database

References

Index

Finally, the index....

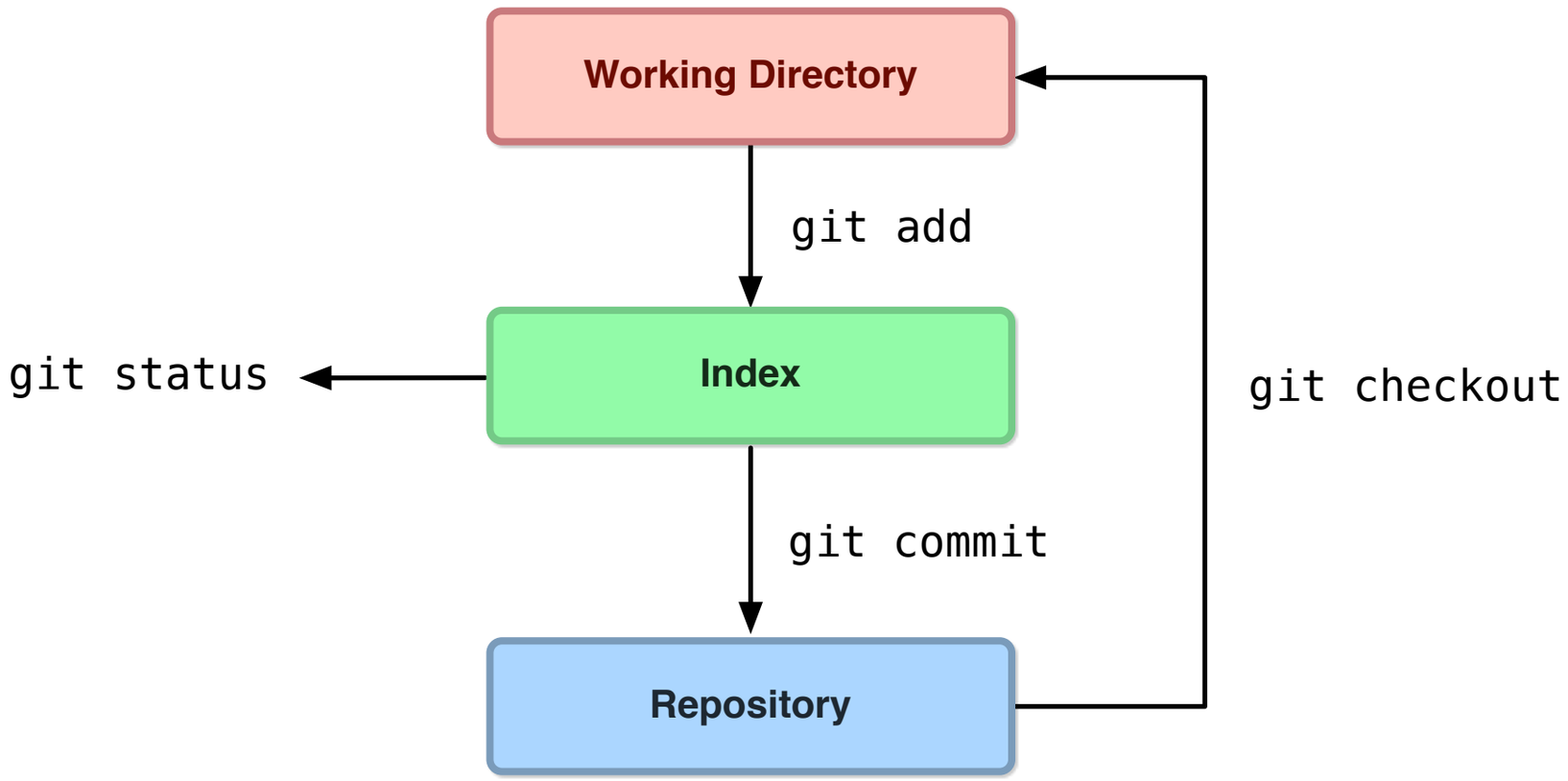
Index

Index

== Staging Area

The index is essentially a staging area.
It lets you craft your commits to be exactly what you want before you commit them.

Index



It works like this.
You change files in your working directory.
Then you add files to your commit

Index FTW

No Need To Commit All At Once

Pick (Stage) Logical Units to Commit

Helps You Review Your Changes

Lets You Write Your History Cleanly

Git Started

New Project

Create and initialize the Git Directory (.git)

```
$ git init
```

Existing Project

Create and pull down remote repository.

```
$ git clone git://github.com/pbhogan/Archivist.git
```

.gitignore

Specify files which will be ignored by Git

```
$ cat .gitgnore
.svn
.DS_Store
build
*.pbxuser
*.perspective
*.perspectivev3
*.mode1v3
*.mode2v3
*.xcuserstate
*.xcworkspace
xcuserdata
```

Staging

Stage files to the index.

```
$ git add .
```

Committing

Create a commit tagged with a message.

```
$ git commit -m "My first commit!"
```

Git will force you to add a commit message.

Branching & Merging

Branching & Merging

Create new branch (from current branch)

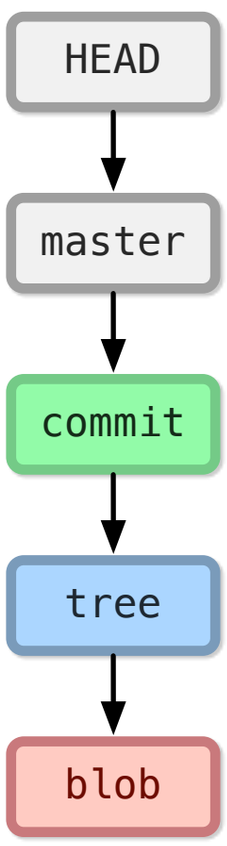
```
$ git branch <name>
```

Branching & Merging

Switch to branch (overwrites Working Dir!)

```
$ git checkout <name>
```

```
$ git commit –m “First commit!”
```

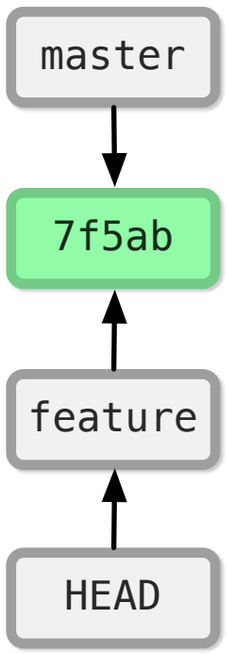


OK, so here we have our first commit with one file.

```
$ git commit –m “First commit!”
```

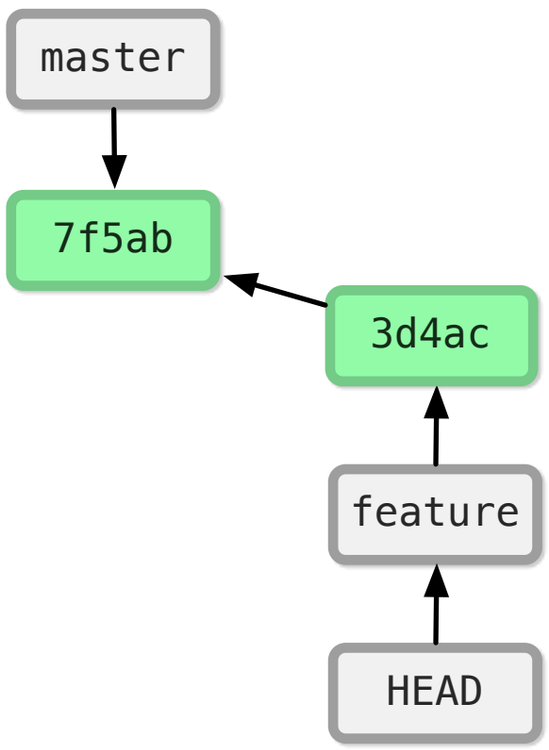
For simplicity, let’s just view the commit and everything beneath it like this.

```
$ git branch feature
$ git checkout feature
```



Now we create a new branch off of master.

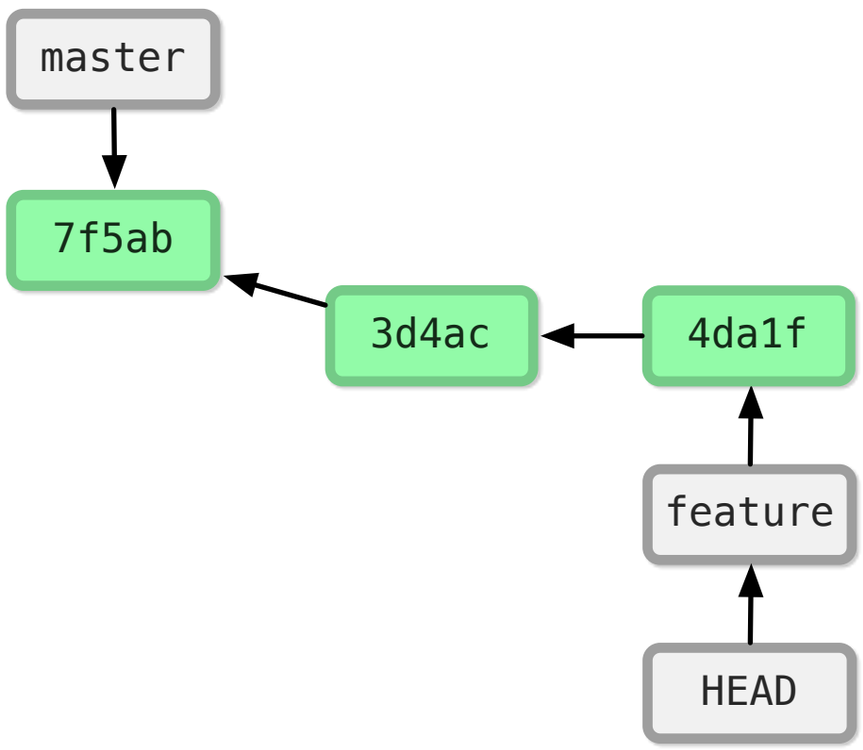
```
$ git add feat.c
$ git commit -m “Added feat.c”
```



Let’s change something and commit.

```
$ git commit -a -m “Updated feat.c”
```

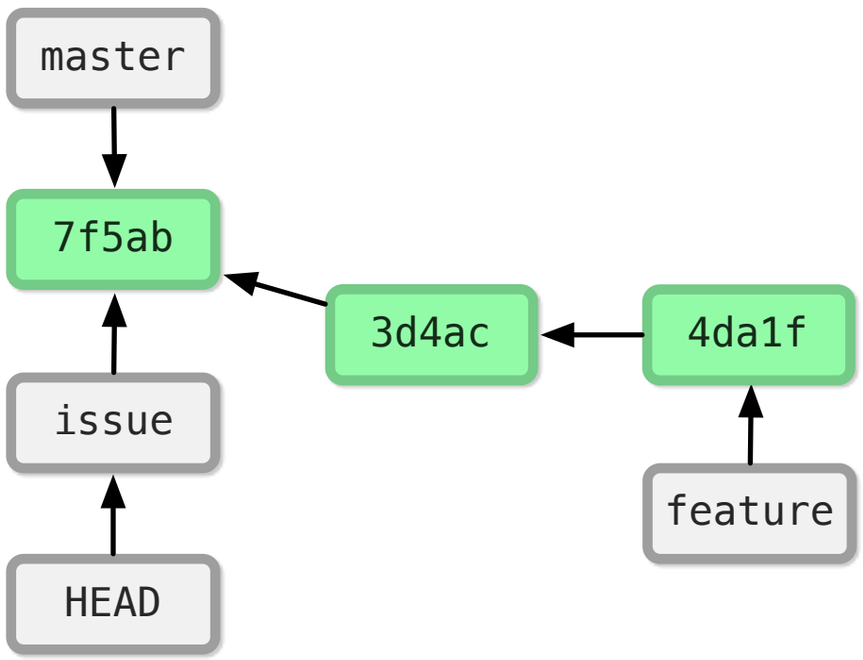
feat.c already tracked so -a automatically stages.



And do it again. You can see we’ve left master behind, but the commits point back to where they came from.

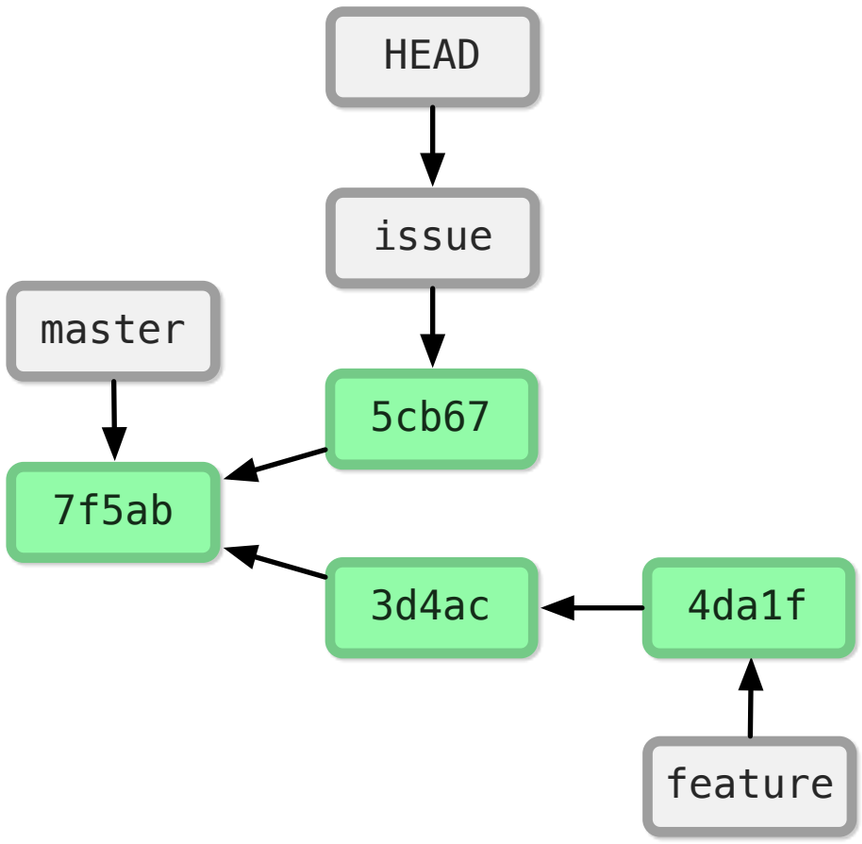
```
$ git checkout -b issue master
```

```
git checkout master
git branch issue
git checkout issue
```

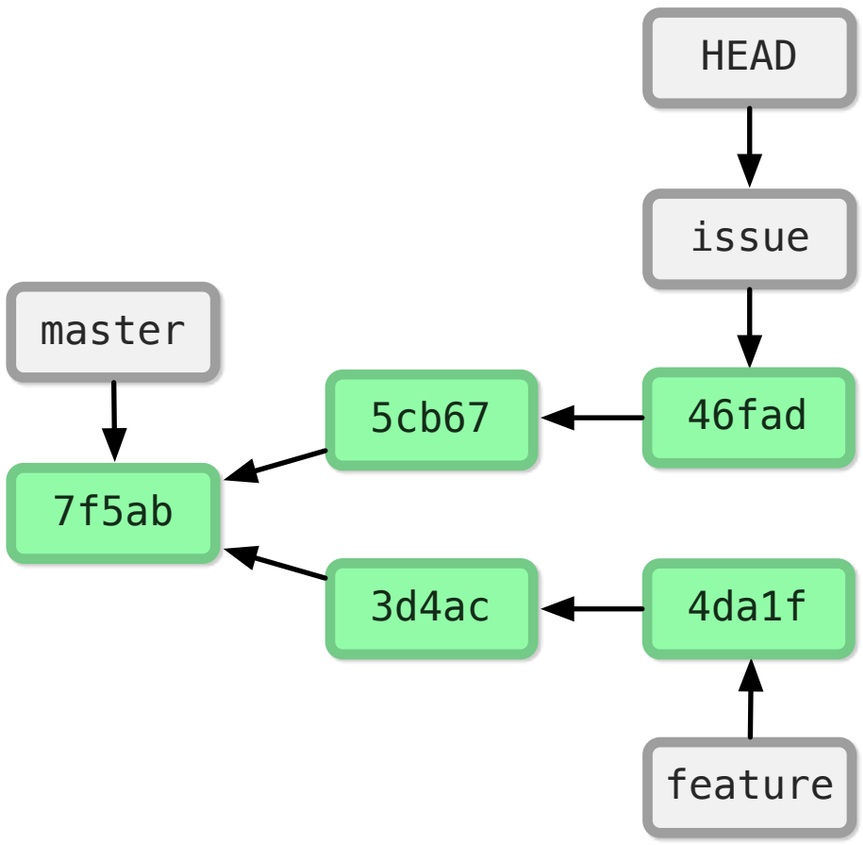


Now lets create yet another branch off of master.
If you come from subversion, this many branches would probably give you an apoplexy, but it’s okay. Git is good at branches.
That command at the top is a shortcut...

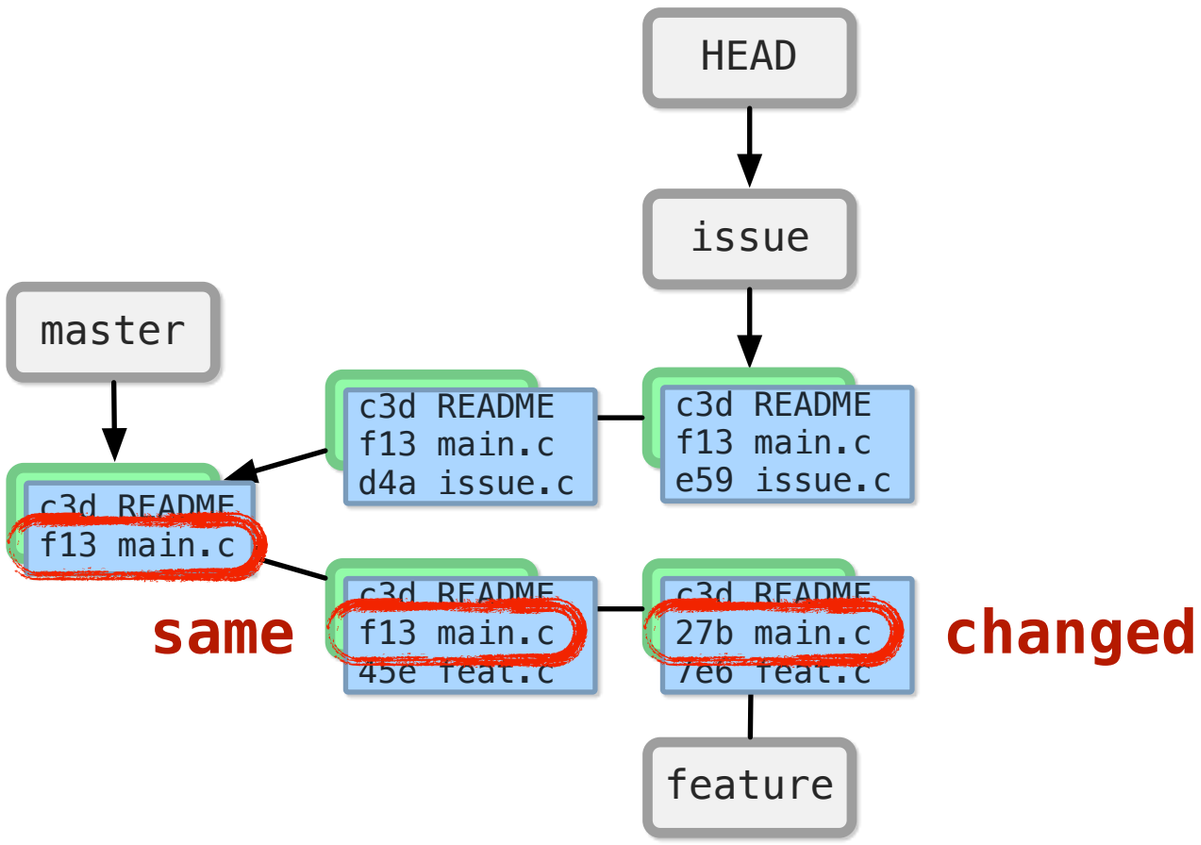
```
$ git add issue.c
$ git commit –m “Added issue.c”
```



```
$ git commit -a -m “Updated issue.c”
```

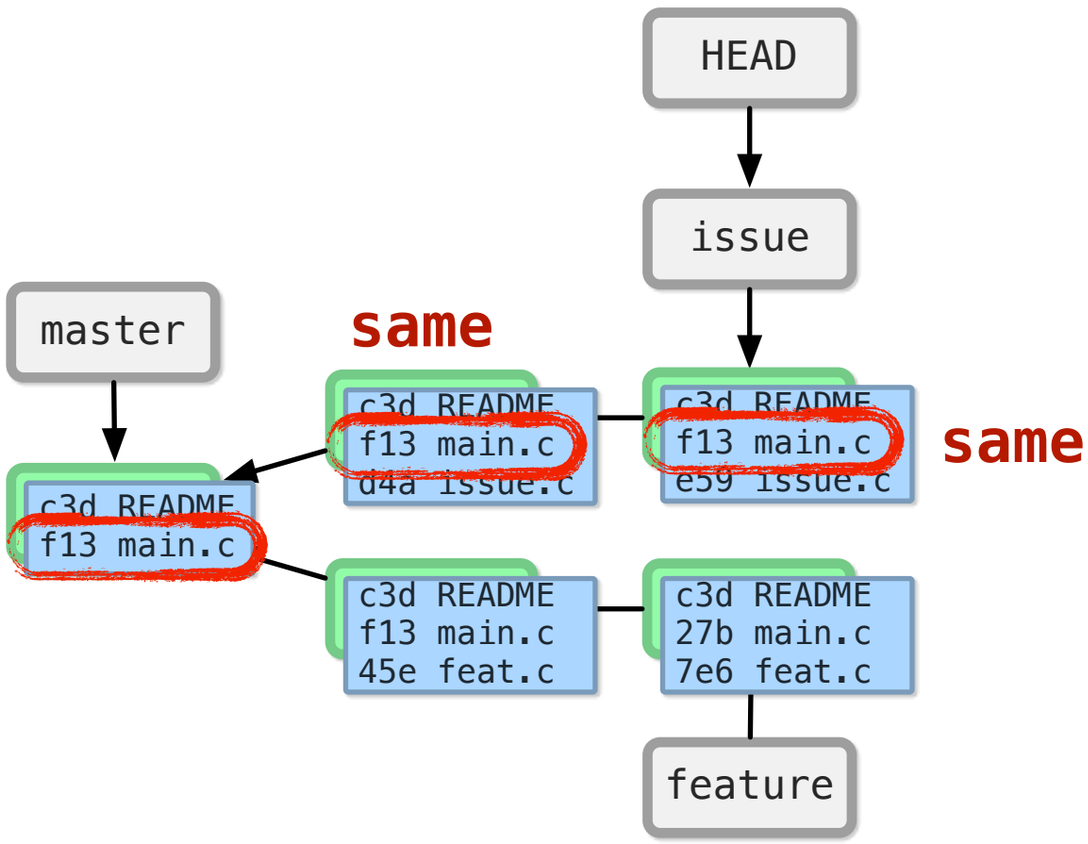


```
$ git log --stat
```

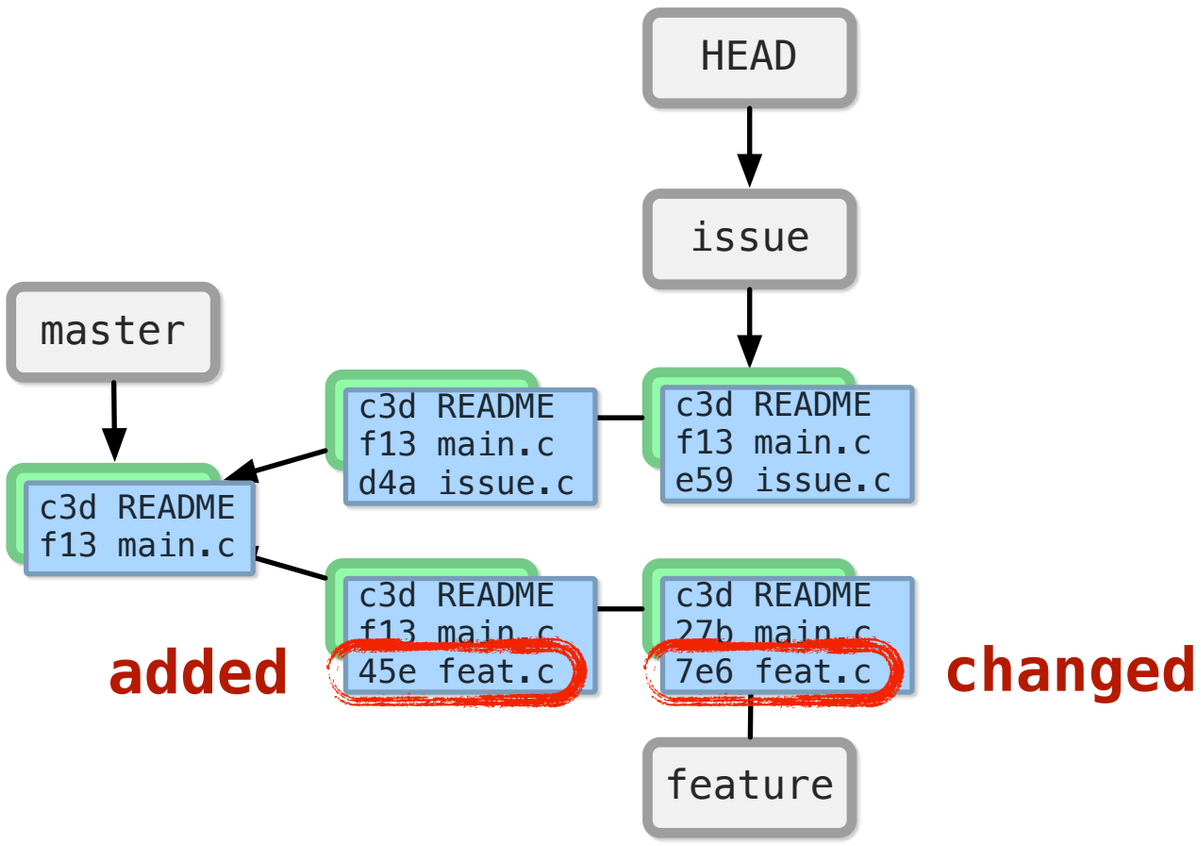


So if we run git log...
We can see what Git sees... if we look at main.c, it’s the same in the 2nd commit and changed in the third.

```
$ git log --stat
```

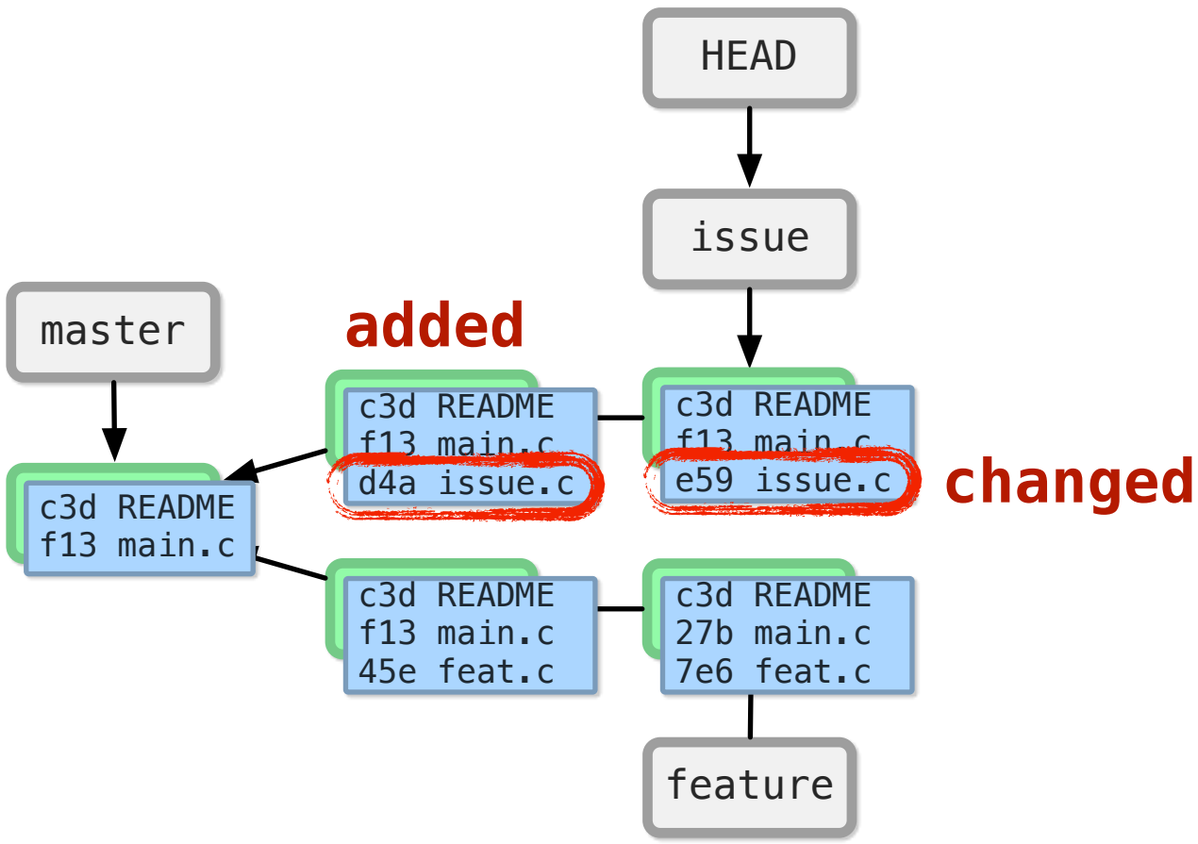

HEAD
issue
master
same
c3d README
f13 main.c
d4a issue.c
c3d README
f13 main.c
e59 issue.c
same
c3d README
f13 main.c
c3d README
f13 main.c
45e feat.c
c3d README
27b main.c
7e6 feat.c
feature

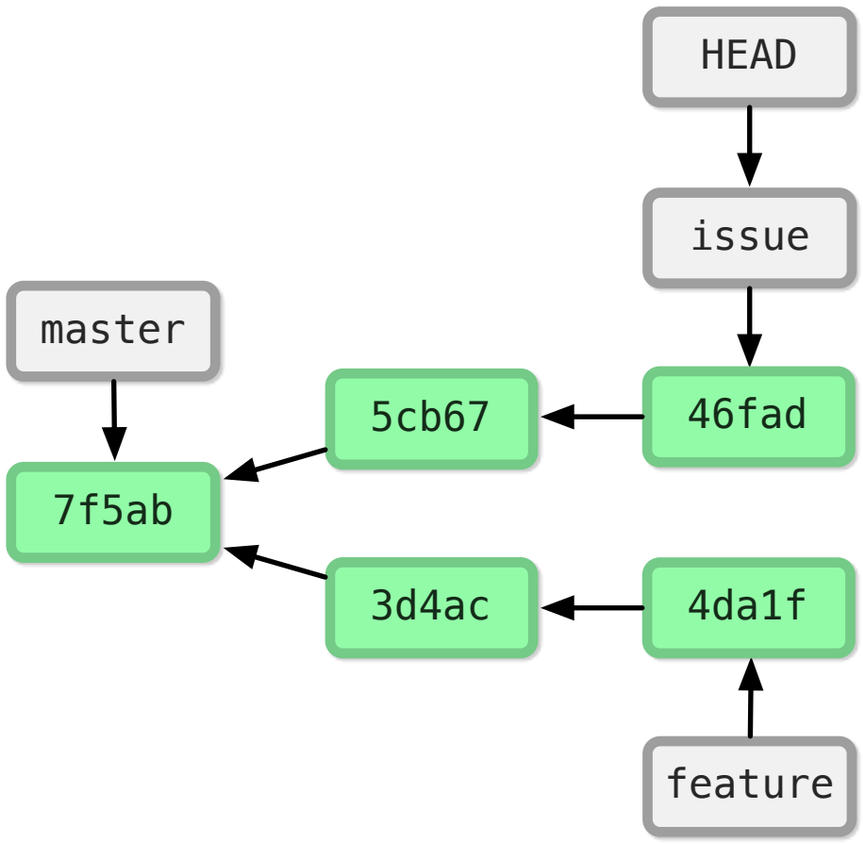
```
$ git log --stat
```



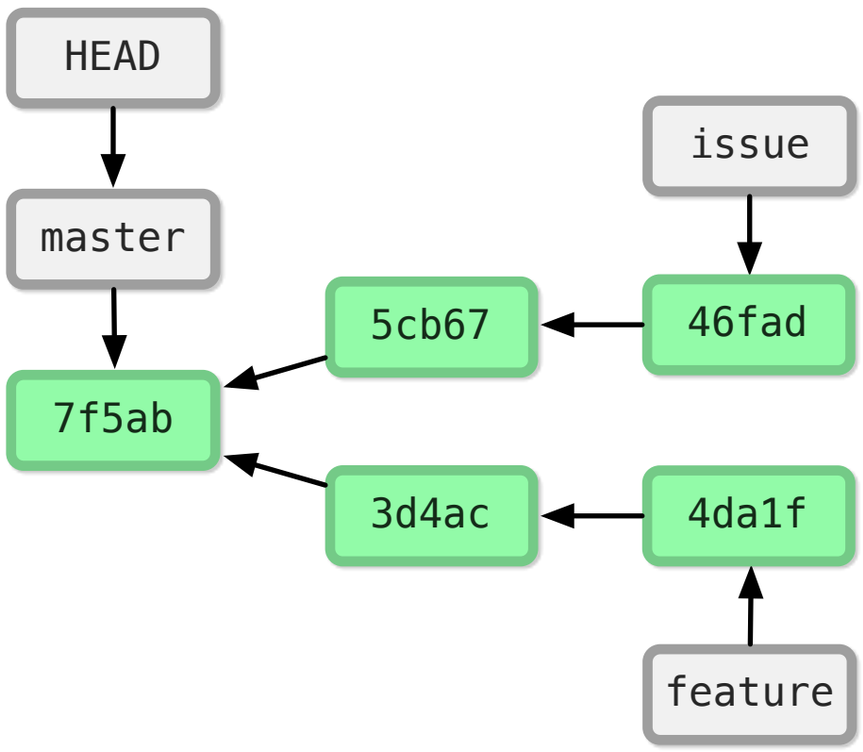
If we look at feat.c, we can see it was added in the 2nd commit and then changed in the 3rd.

```
$ git log --stat
```

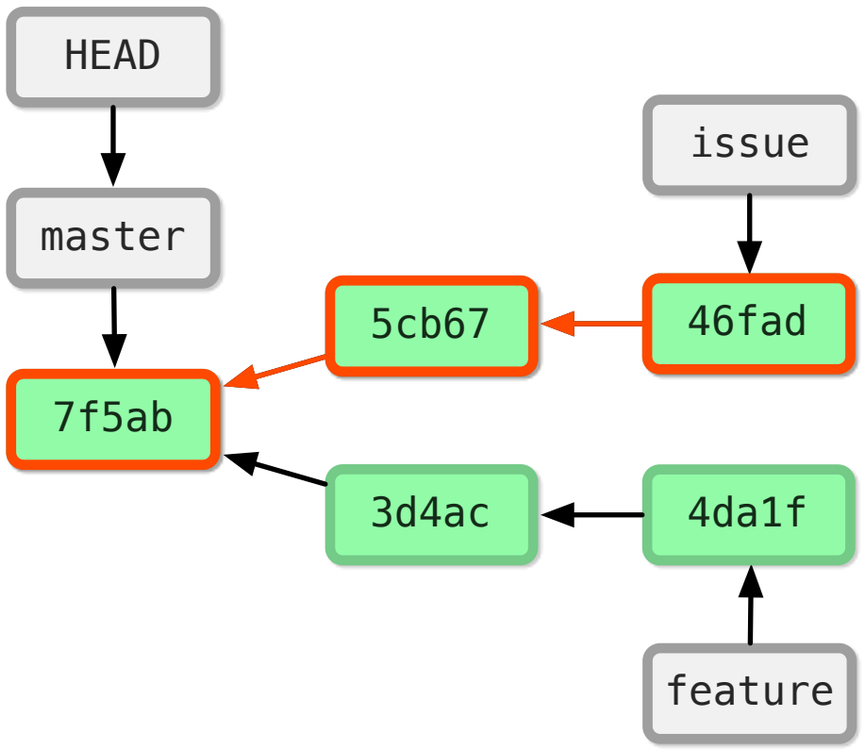

HEAD
issue
master
added
c3d README
f13 main.c
d4a issue.c
c3d README
f13 main.c
e59 issue.c
changed
c3d README
f13 main.c
c3d README
f13 main.c
45e feat.c
c3d README
27b main.c
7e6 feat.c
feature


HEAD
issue
master
5cb67
46fad
7f5ab
3d4ac
4da1f
feature

```
$ git checkout master
```

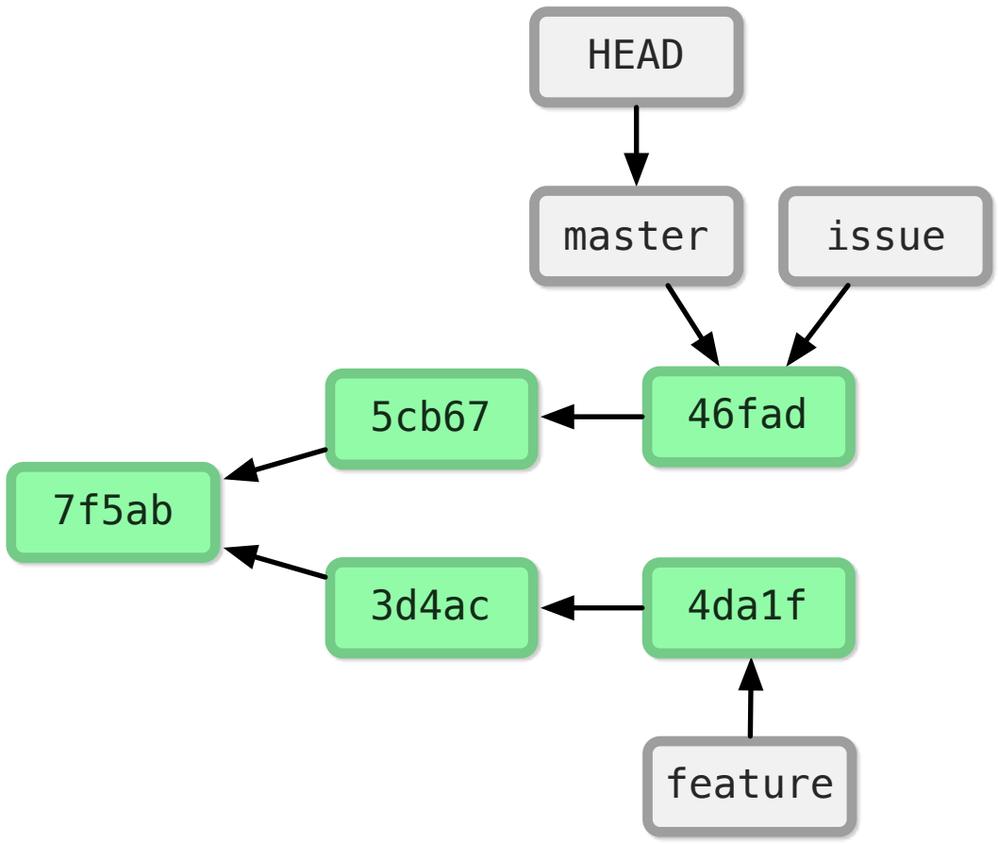


```
$ git merge issue
```

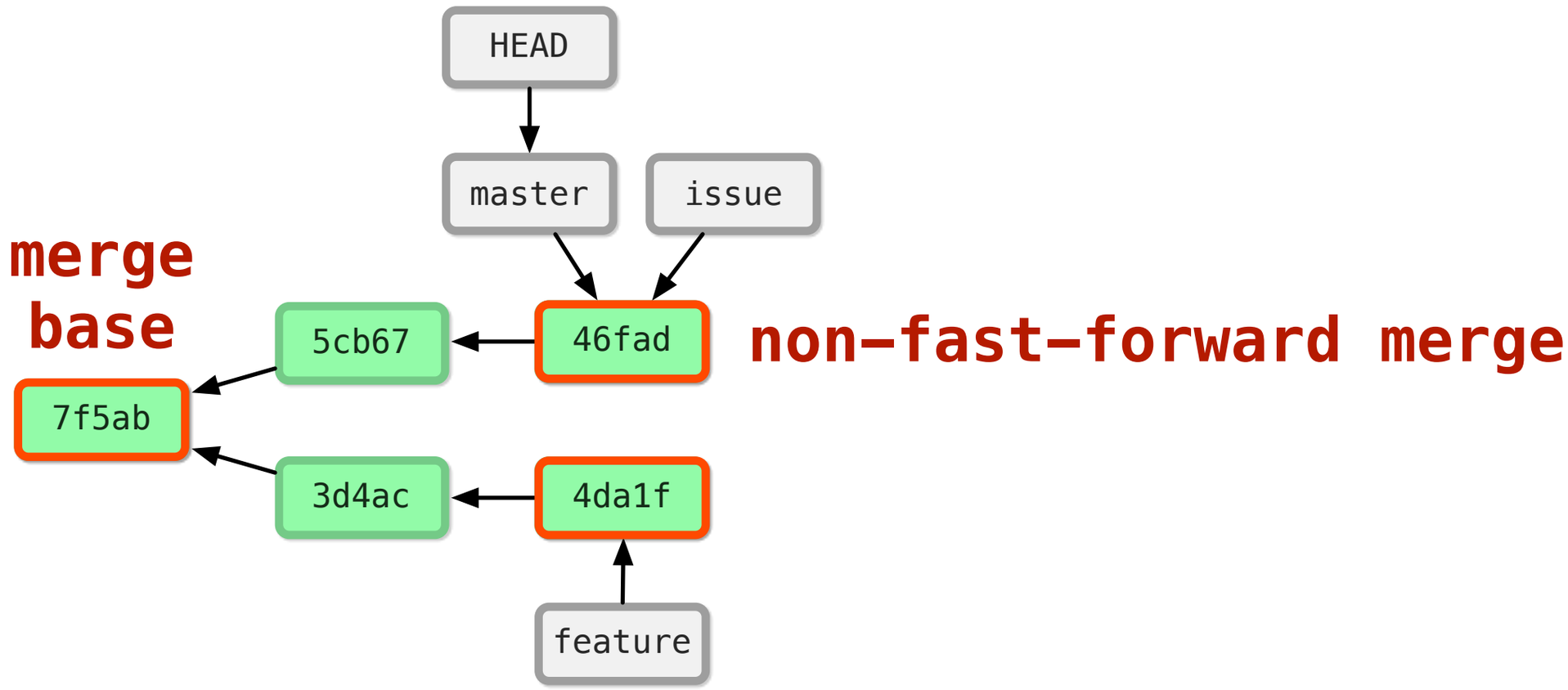


fast-forward merge

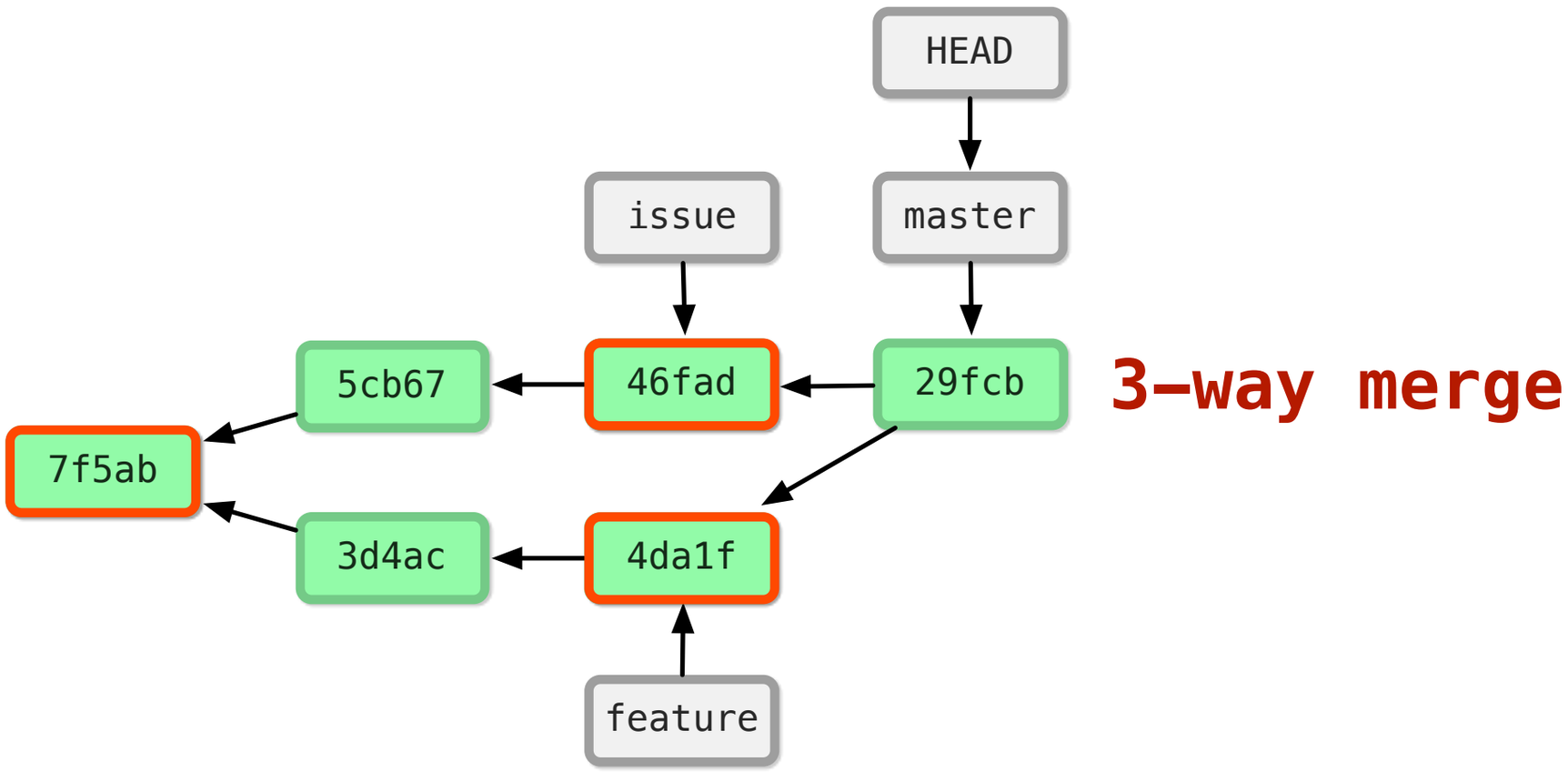
```
$ git merge issue
```



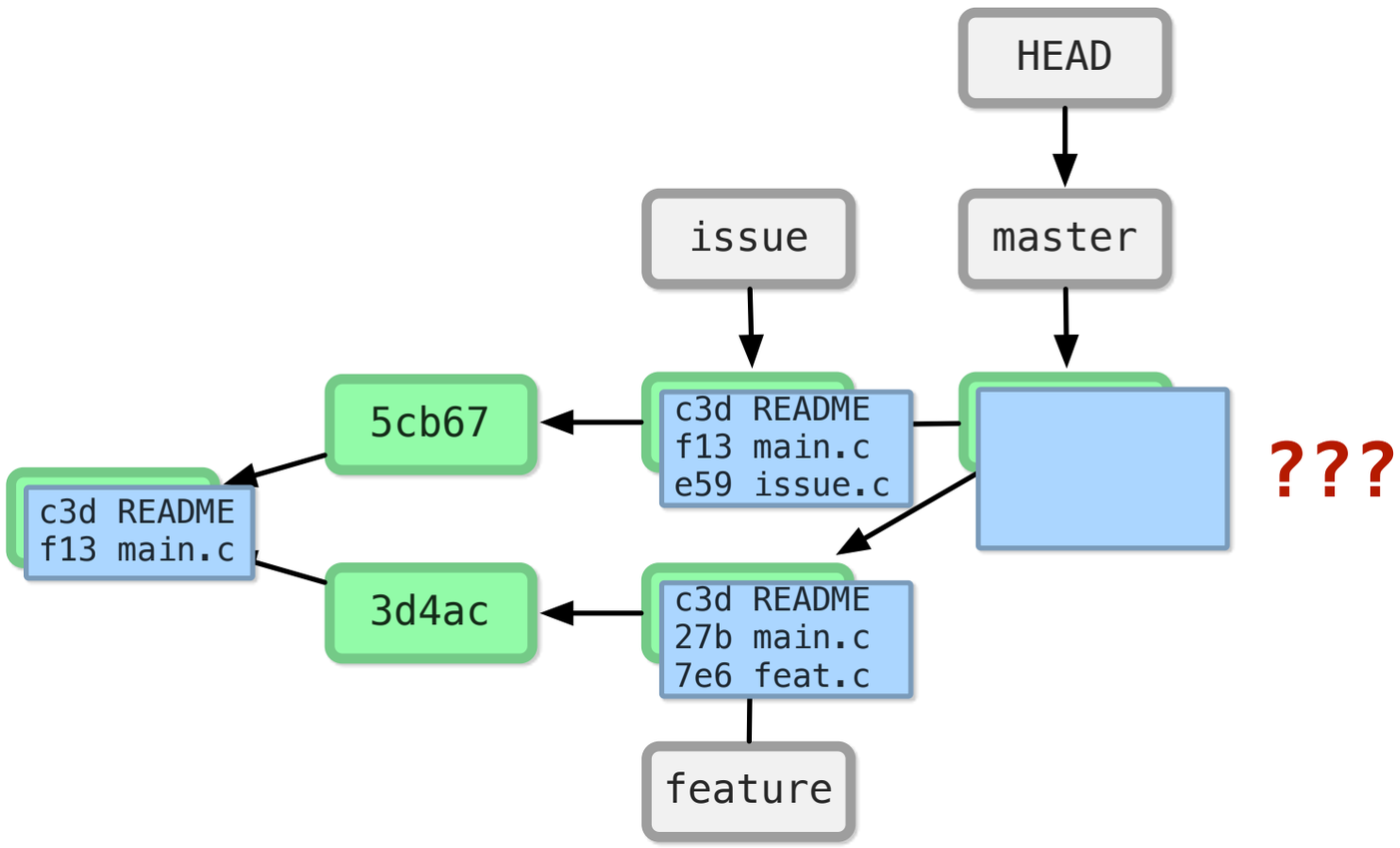
```
$ git merge feature
```



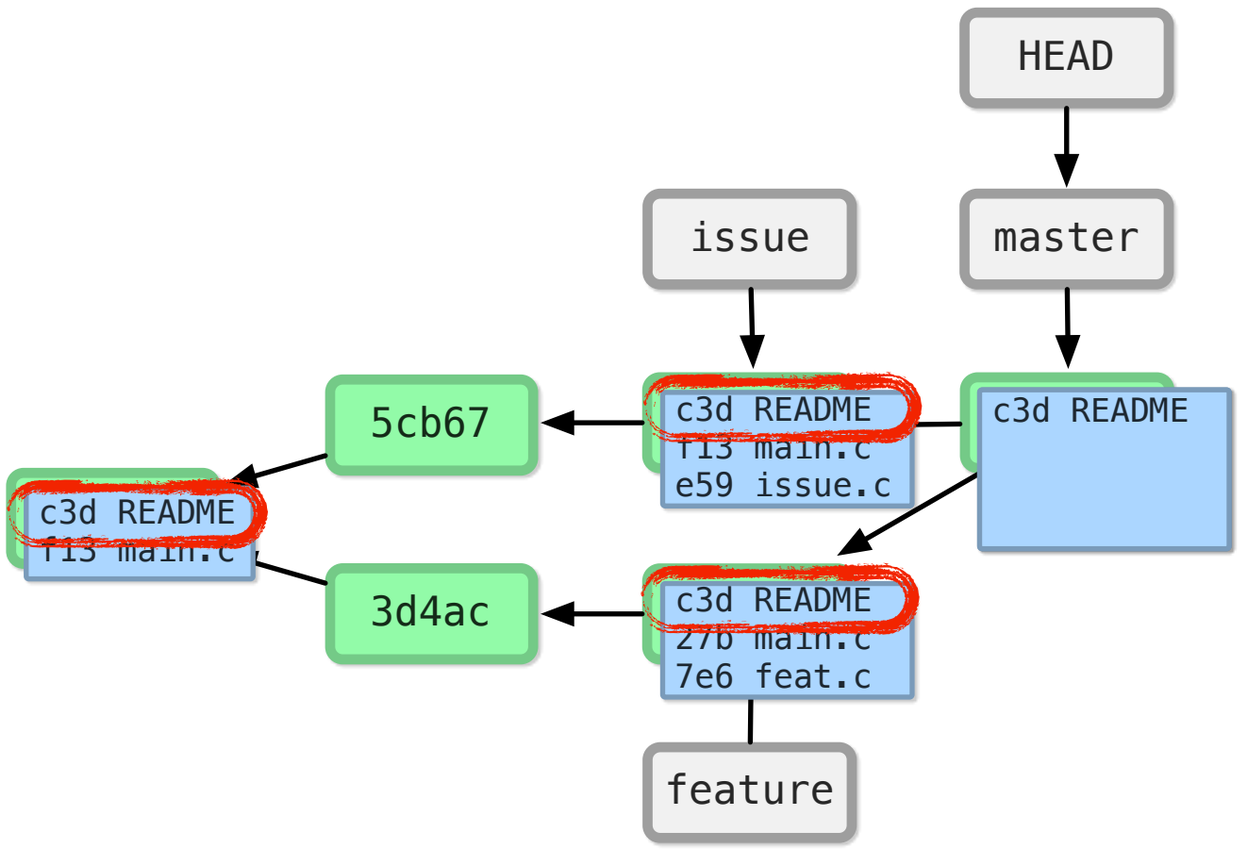
```
$ git merge feature
```



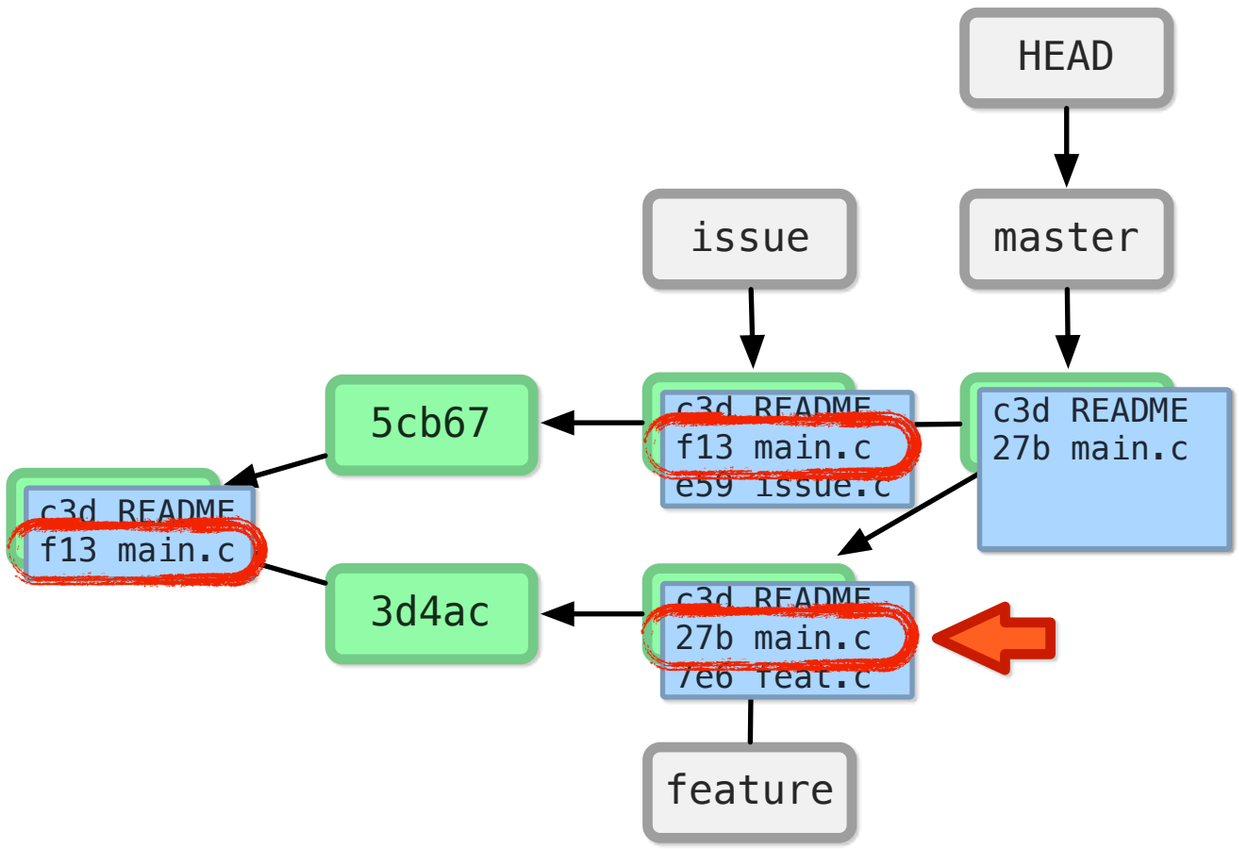
```
$ git merge feature
```



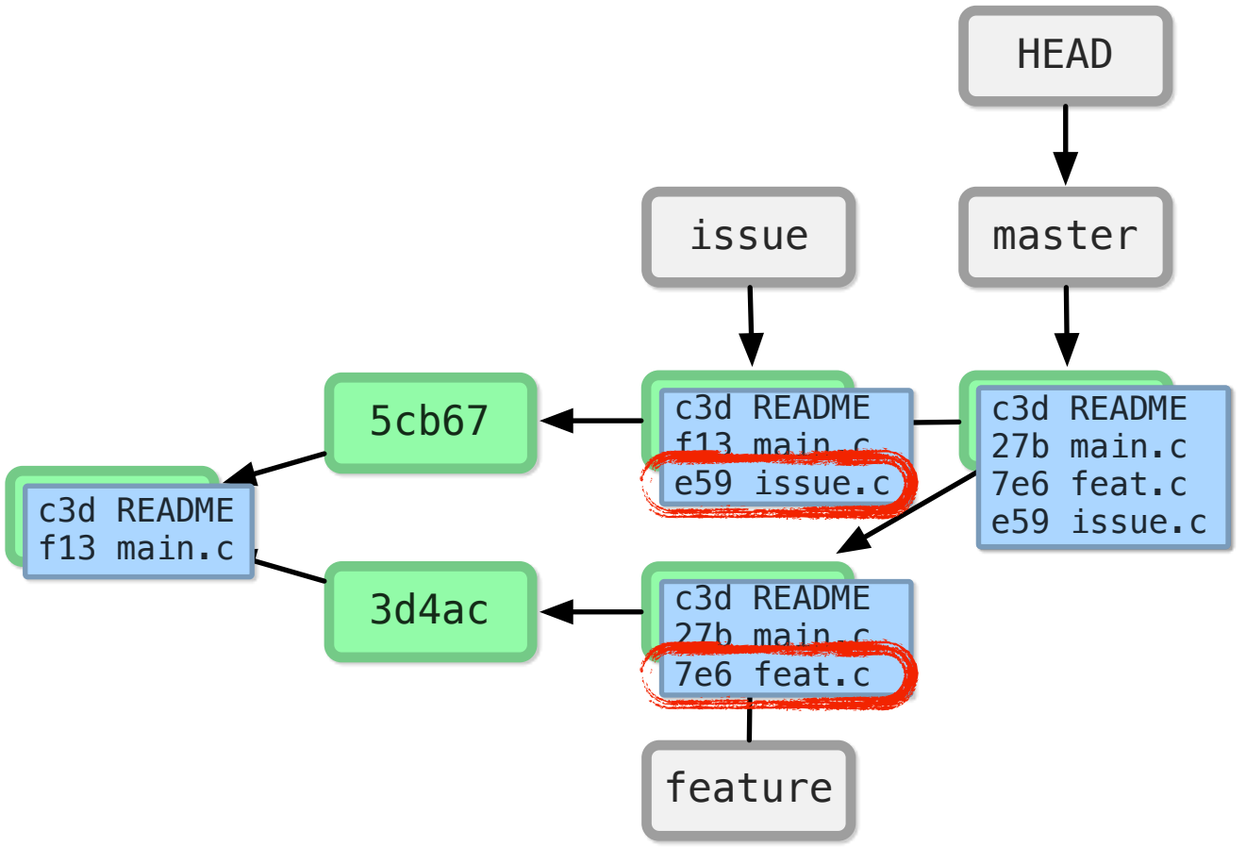
```
$ git merge feature
```



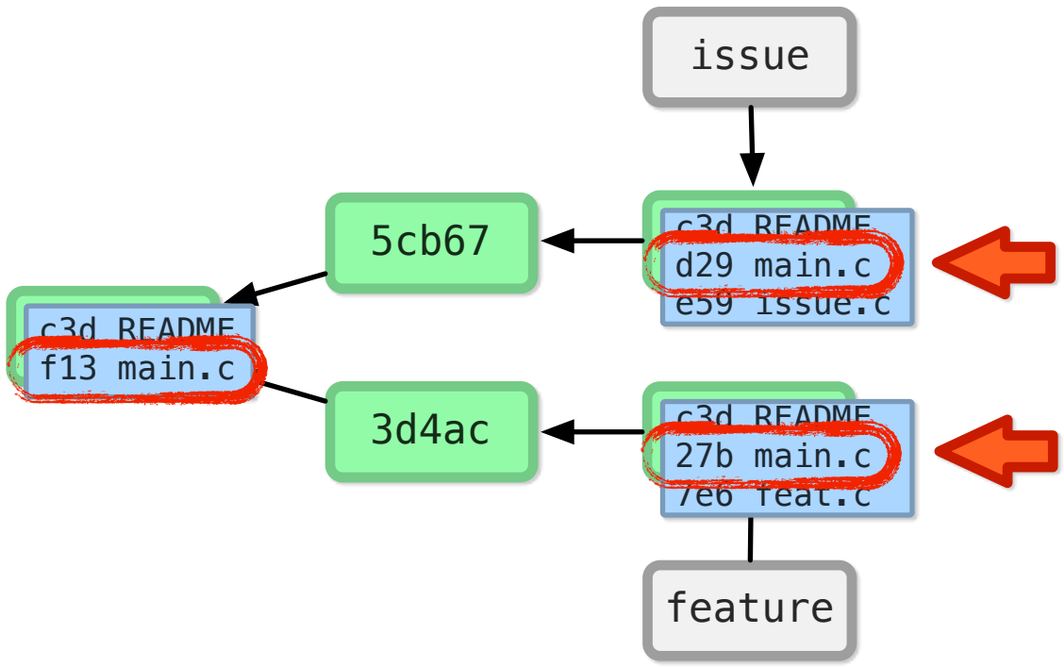
```
$ git merge feature
```



```
$ git merge feature
```



```
$ git merge feature
```



What if main.c
MERGE CONFLICT!
in both branches?

Merge Conflict

```
$ git merge feature

Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then
commit the result.
```

Merge Conflict

```
$ git merge feature

# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate
to mark resolution)
#
#  both modified:      main.c
#
no changes added to commit (use "git add" and/
or "git commit -a")
```

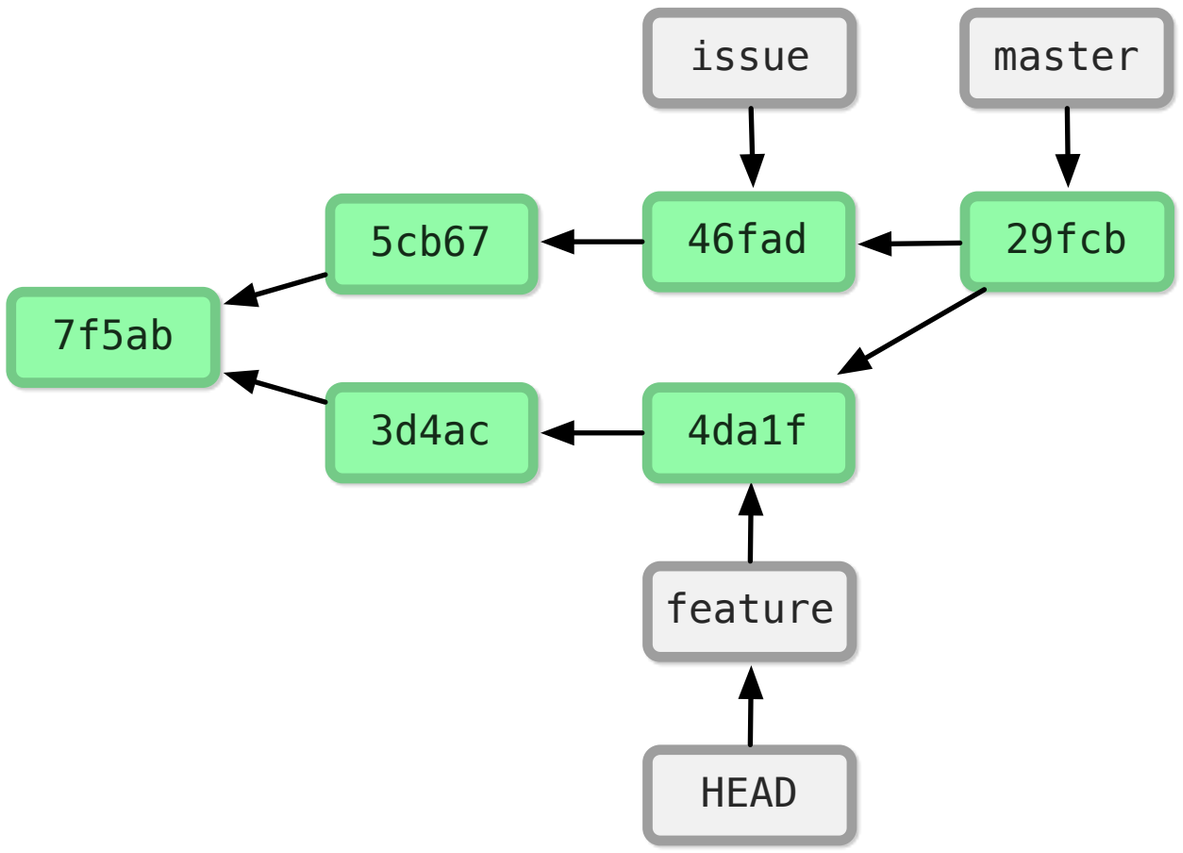
Merge Conflict

```
$ cat main.c

int main (int argc, char const *argv[])
{
<<<<<<< HEAD
   printf( "Hola World!");
=======
   printf("Hello World!");
>>>>>>> feature
   return 0;
}
```

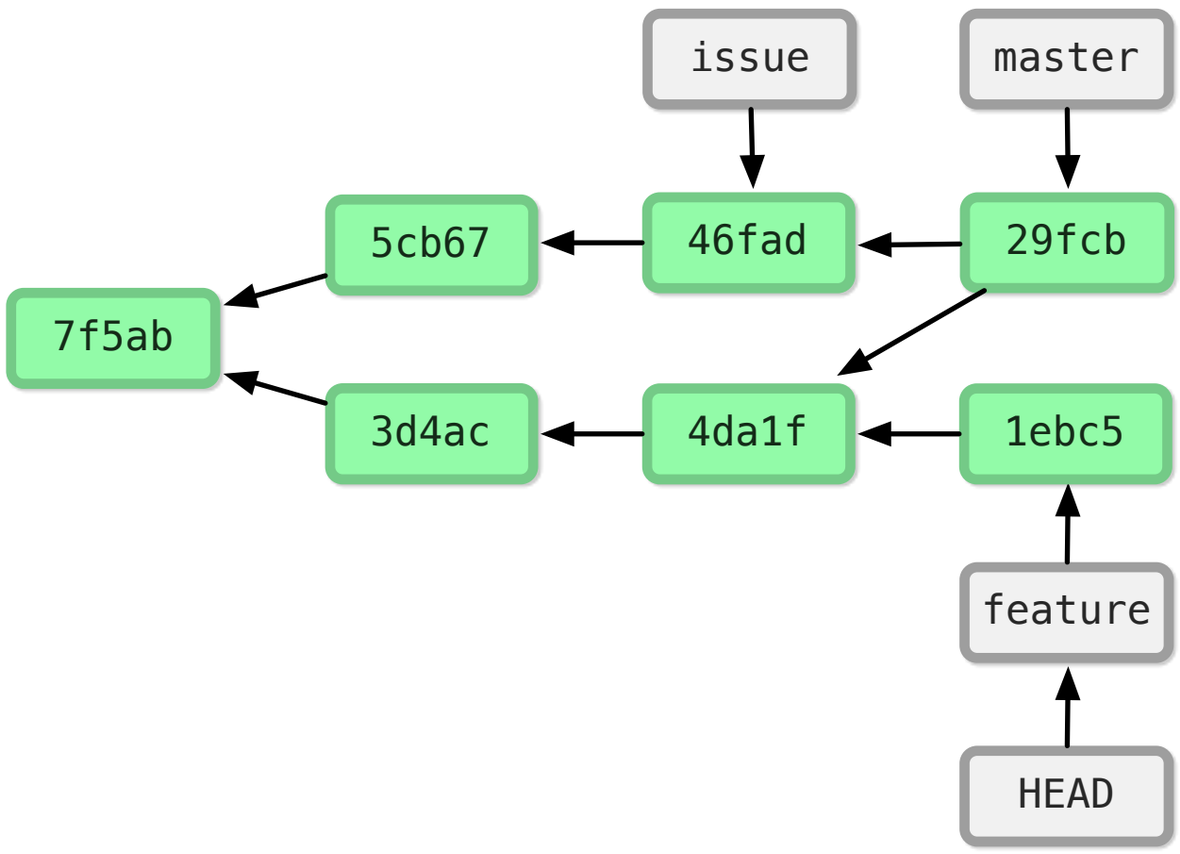
Merge Conflict

```
$ git mergetool
$ git add main.c
$ git commit -m “Merged feature”
```

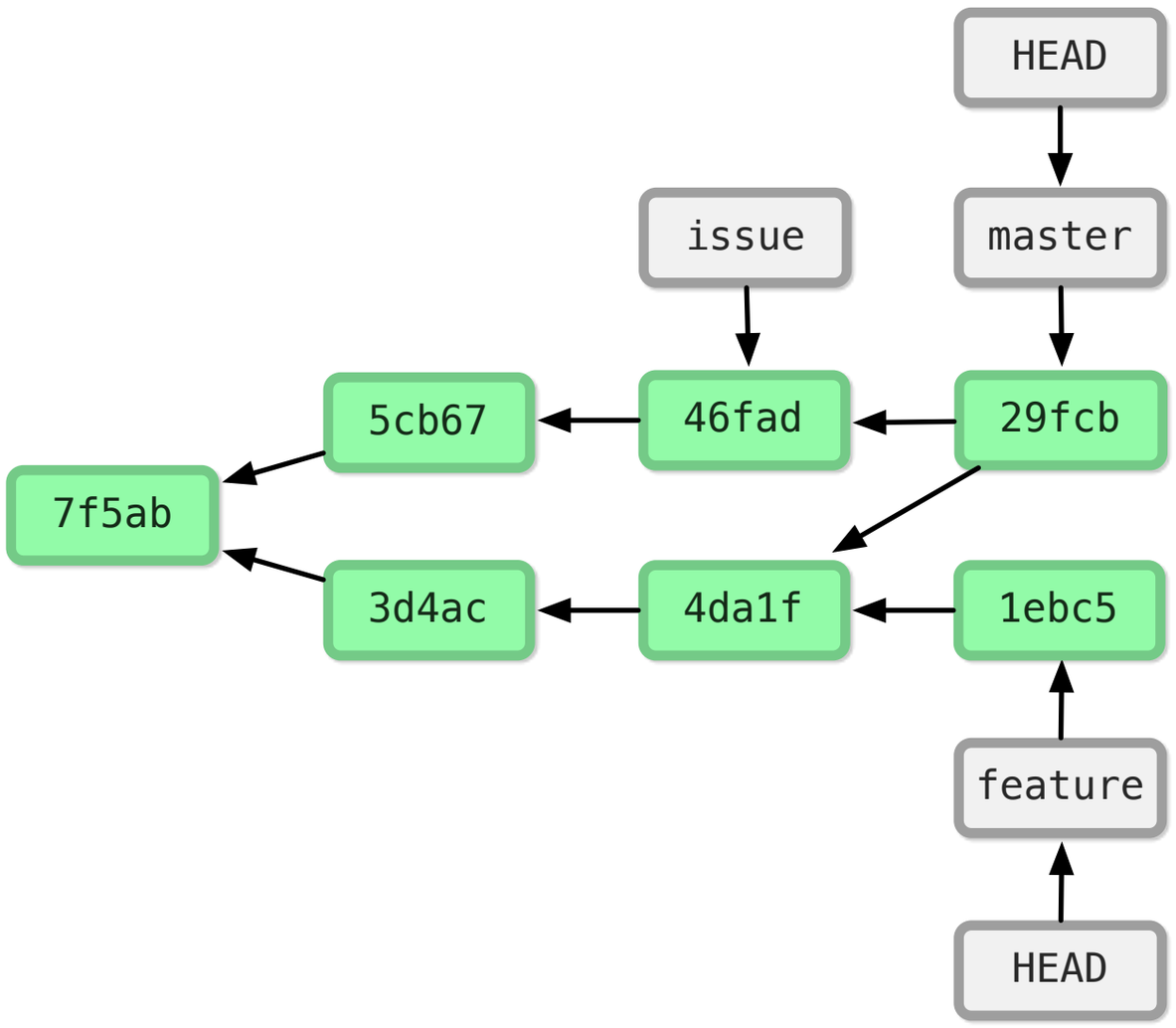
```
$ git checkout feature
```



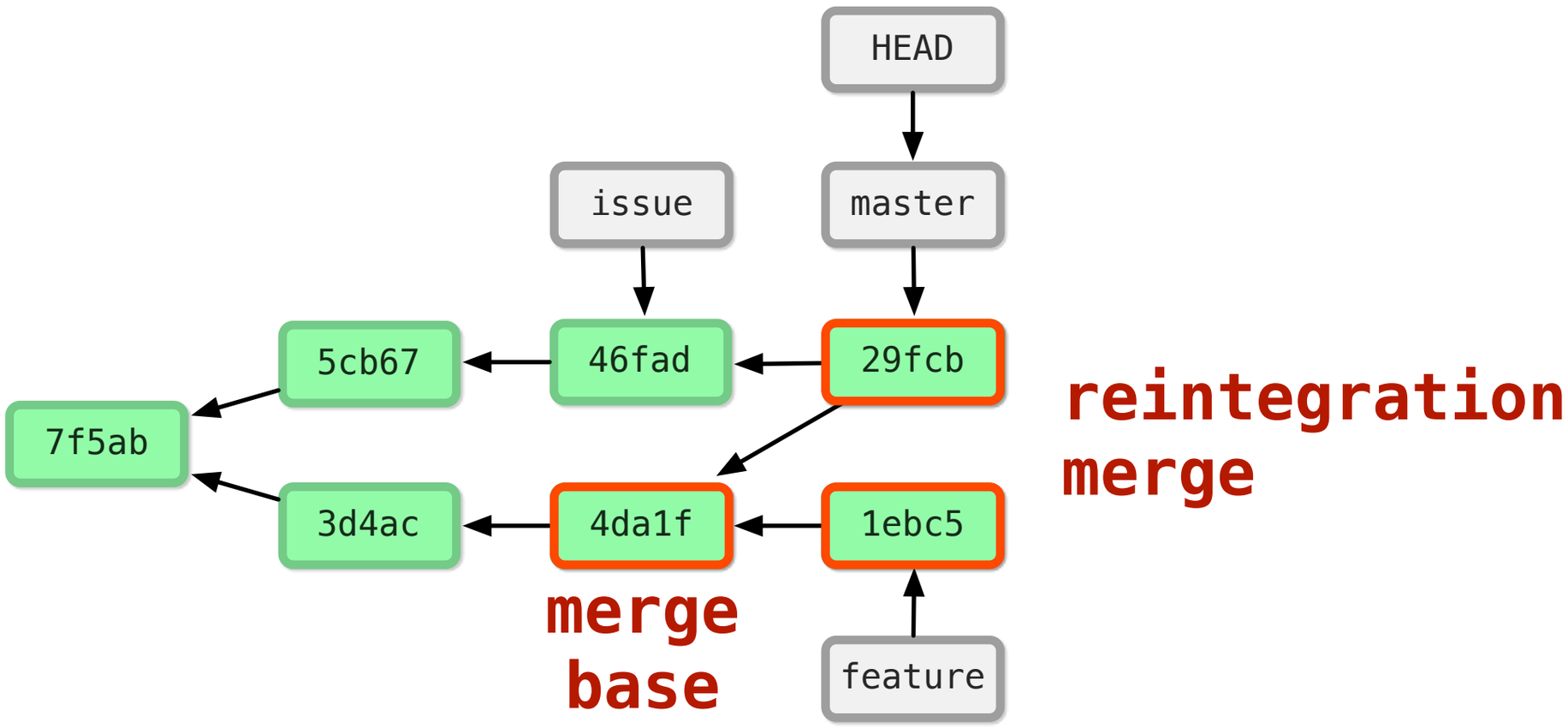
```
$ git commit -am “More on feature.”
```



```
$ git checkout master
```

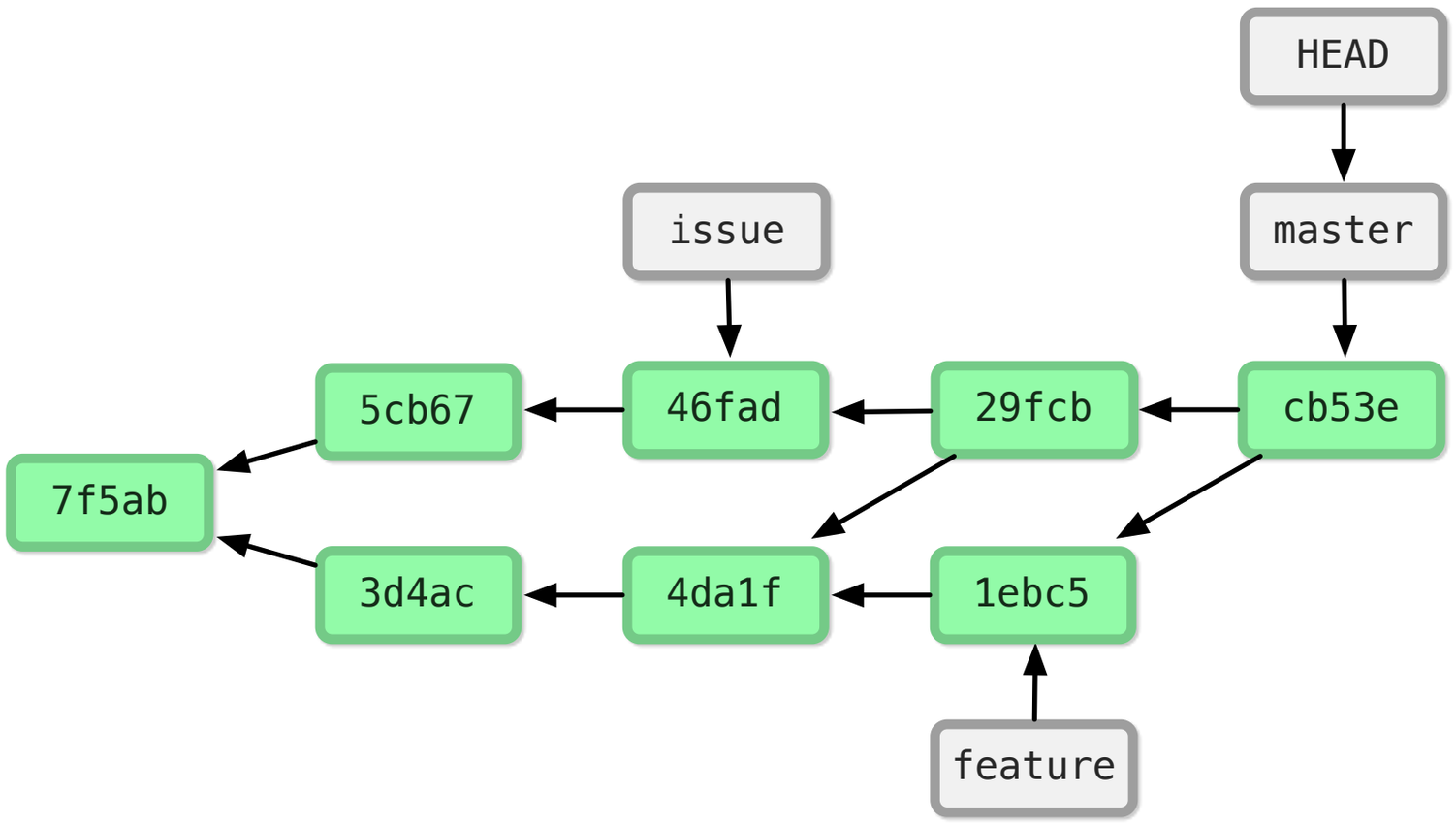


```
$ git merge feature
```



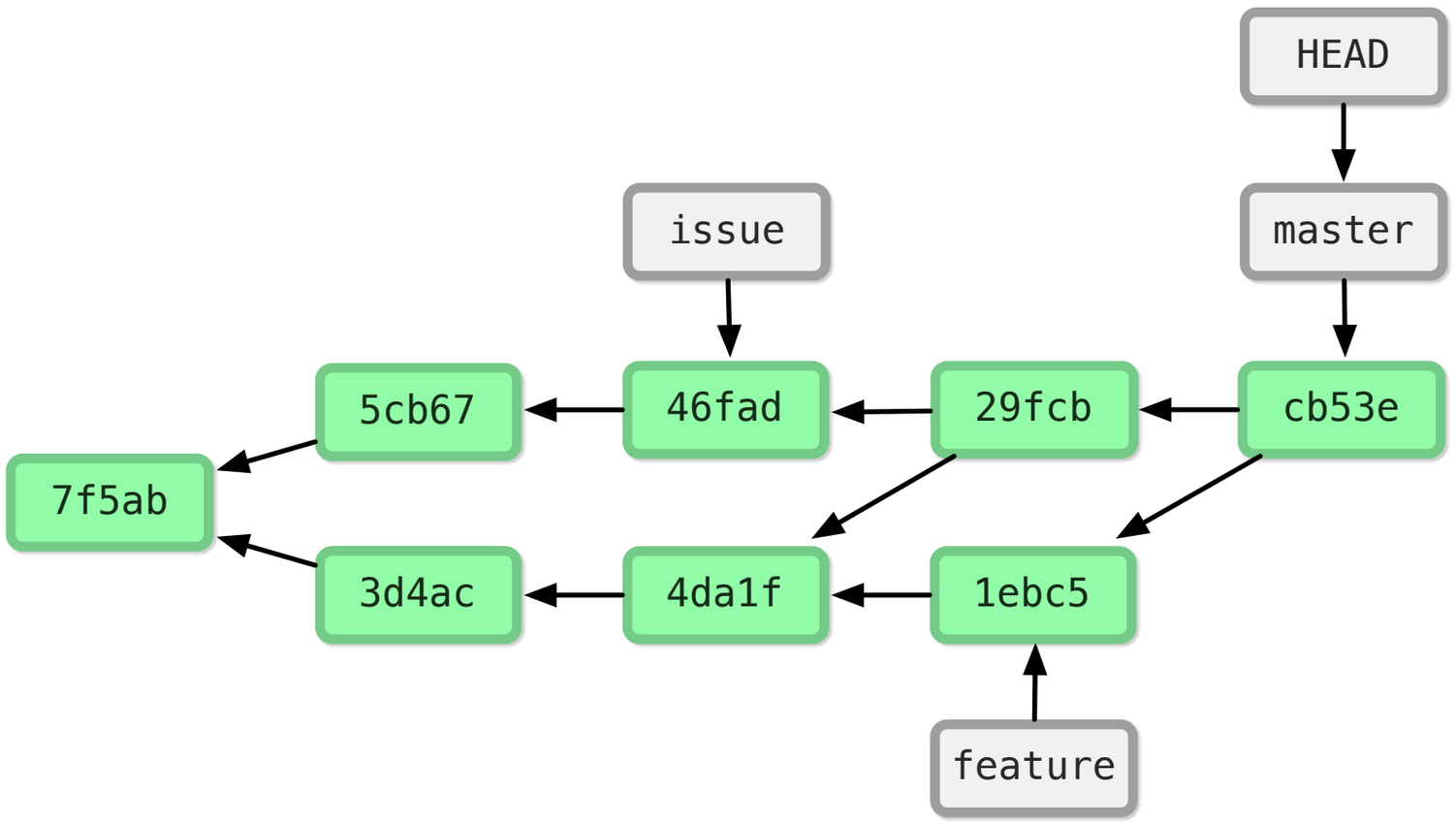
Reintegration merge

```
$ git merge feature
```

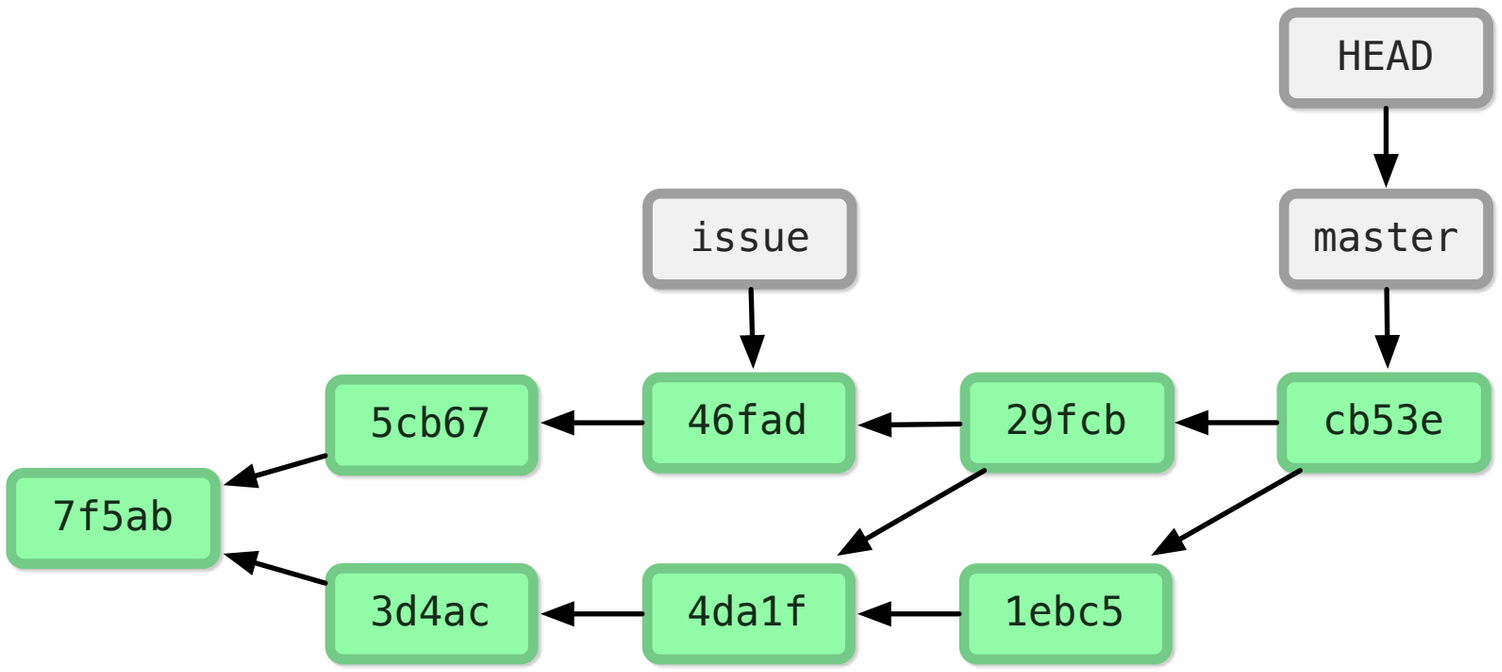


Reintegration merge

```
$ git branch -d feature
```



```
$ git branch -d issue
```



Branching & Merging

Delete branch with extreme prejudice.

```
$ git branch -D <name>
```

Branching & Merging

Isolate Experiments

Isolate Work Units

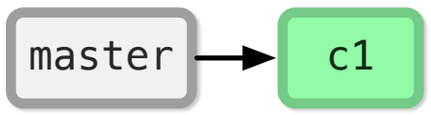
Parallelize

Long Running Topics

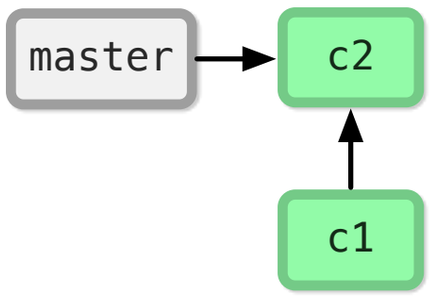
Hot Fix

Merge vs. Rebase

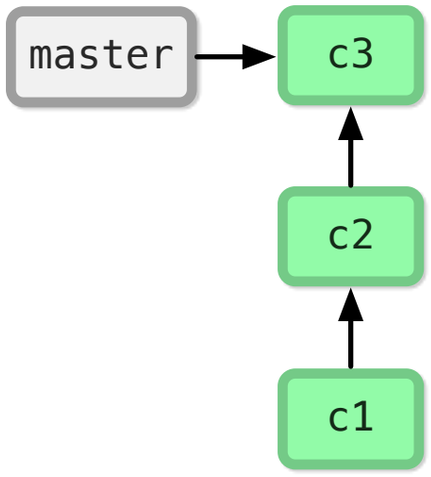
Merge vs. Rebase



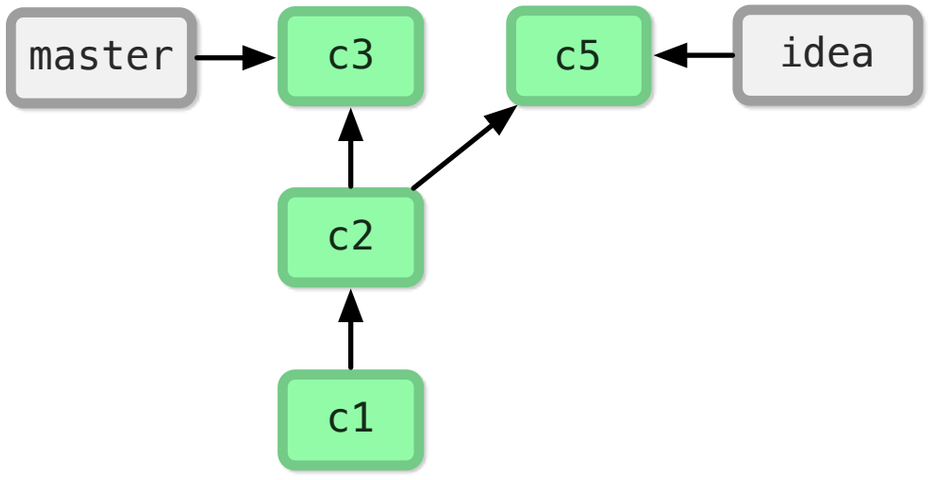
Merge vs. Rebase



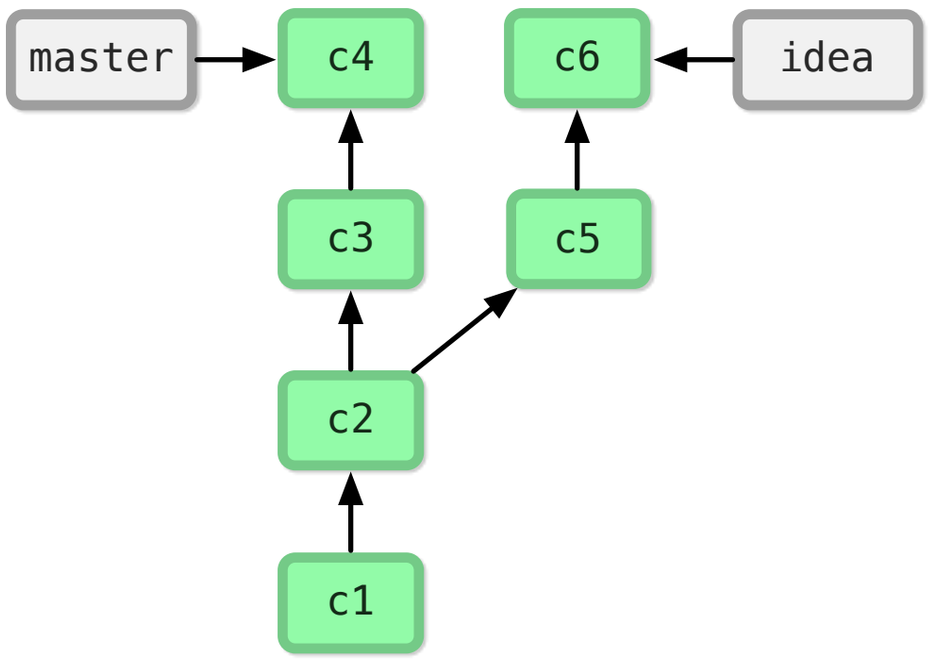
Merge vs. Rebase



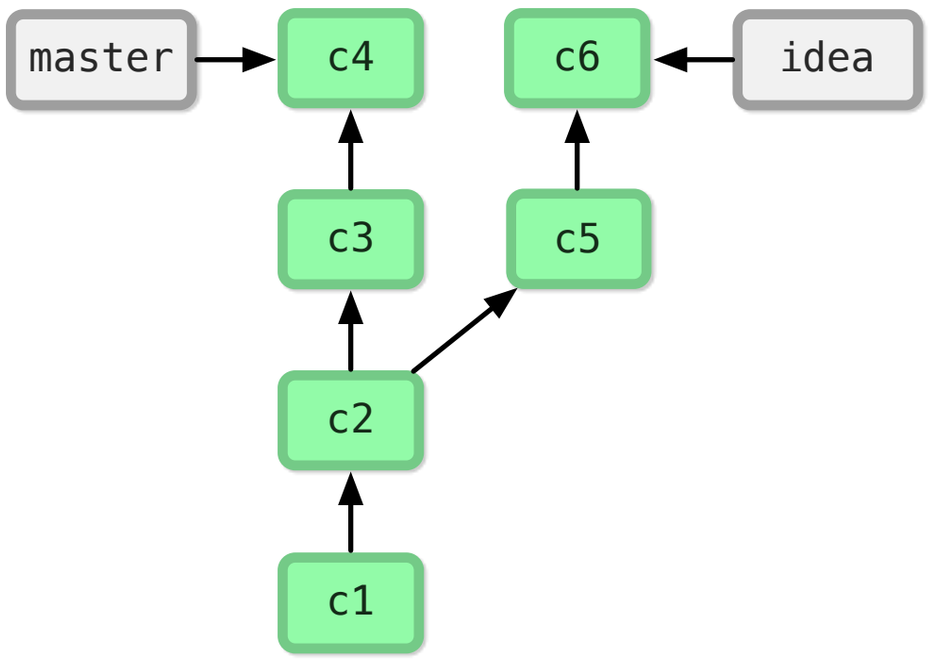
Merge vs. Rebase



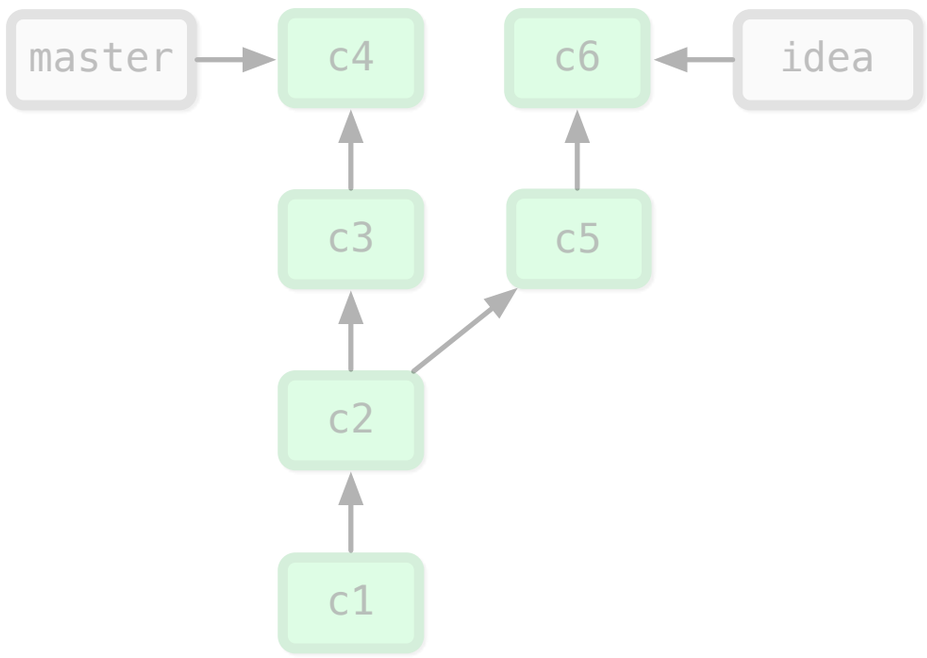
Merge vs. Rebase



Merge vs. Rebase

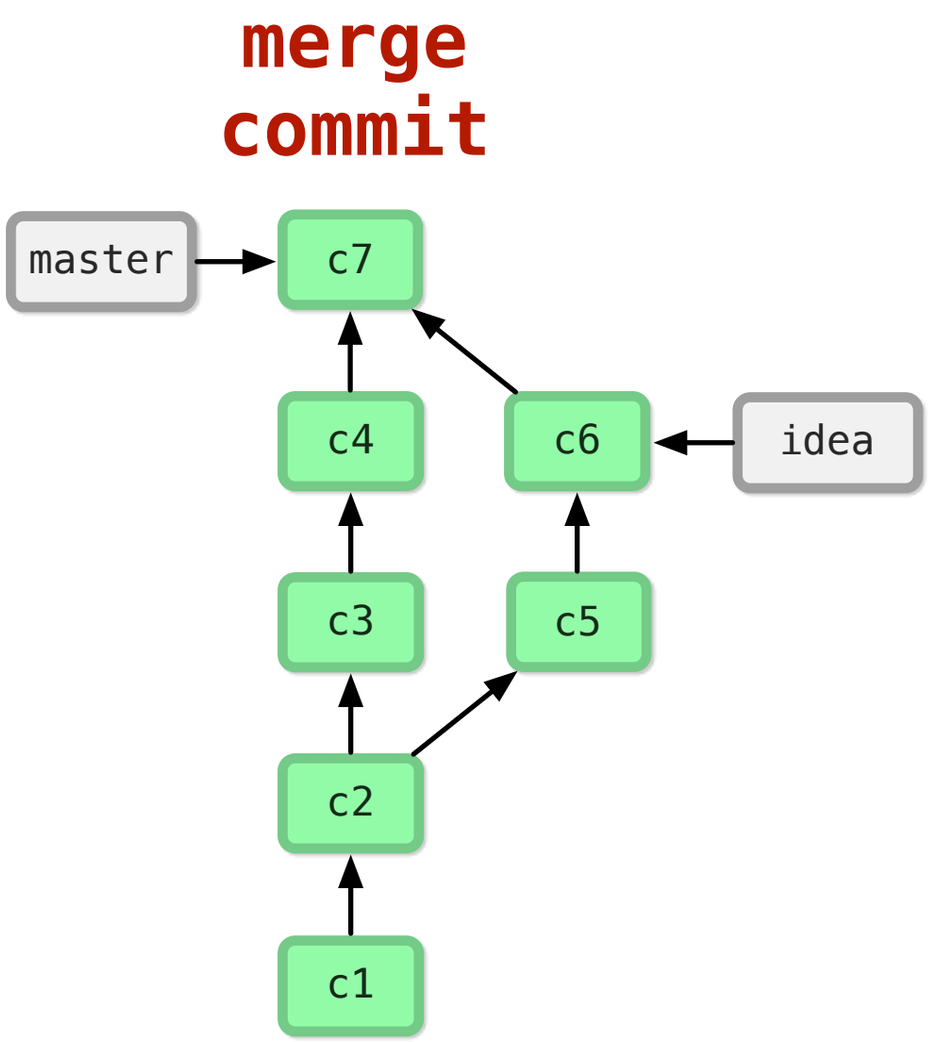


merge

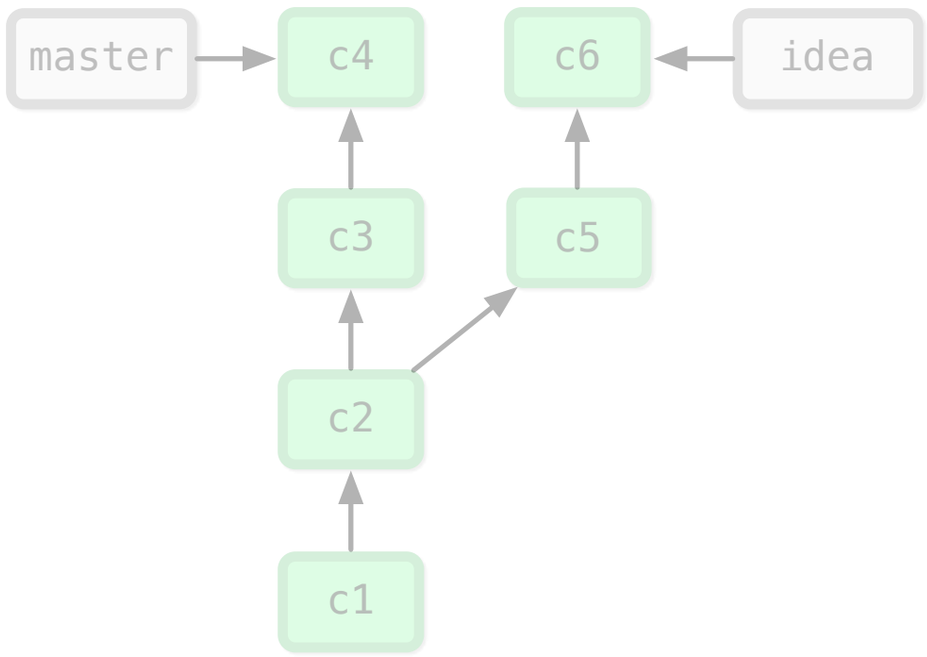


rebase

Merge vs. Rebase

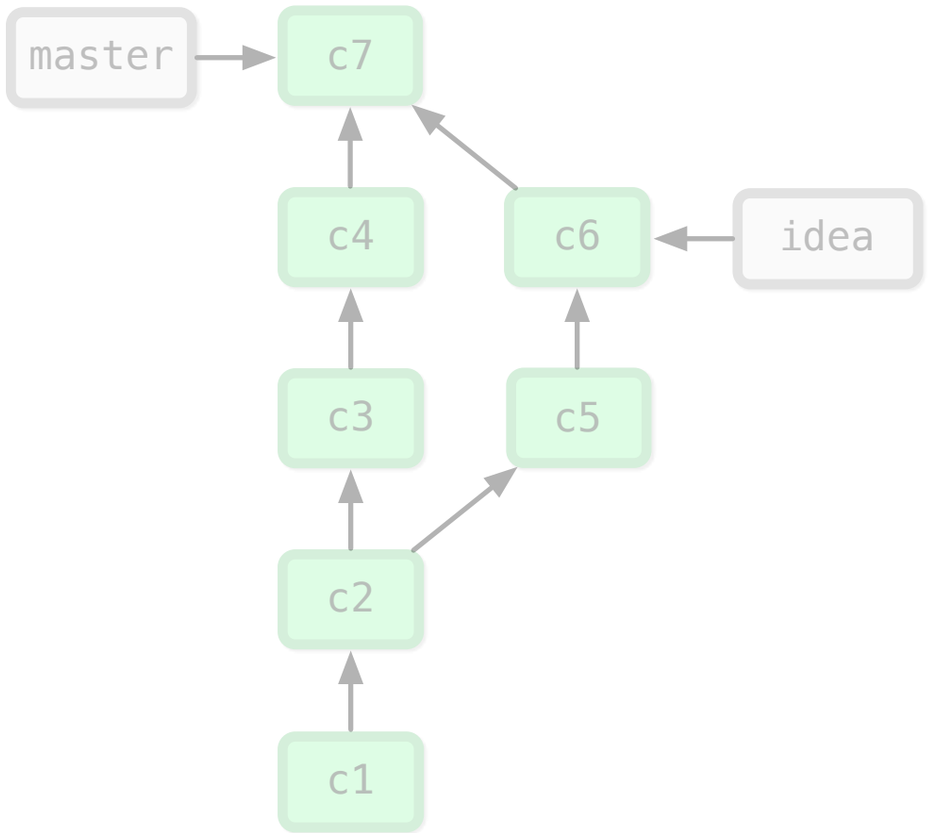


merge

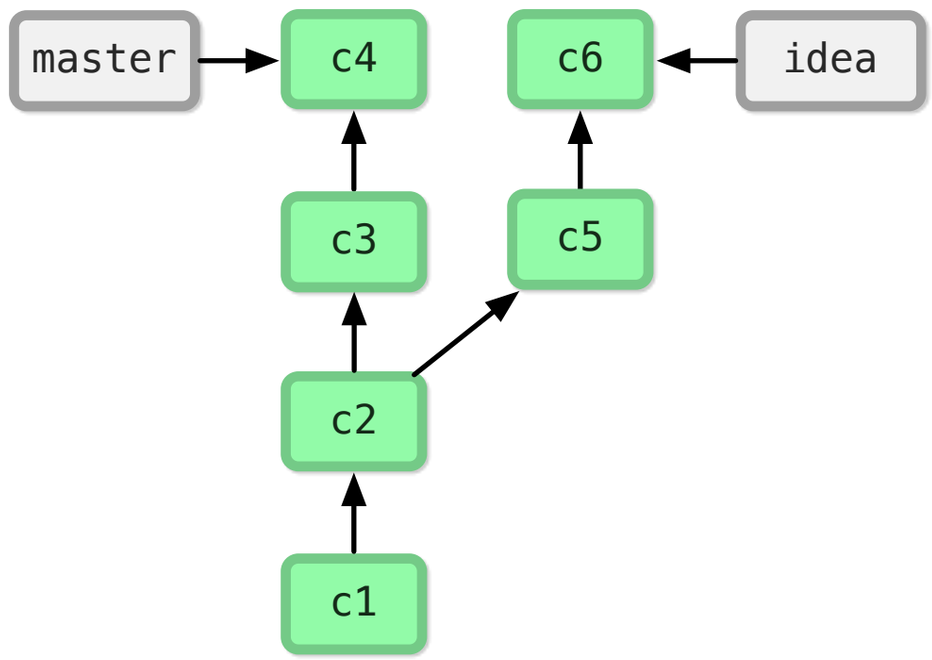


rebase

Merge vs. Rebase

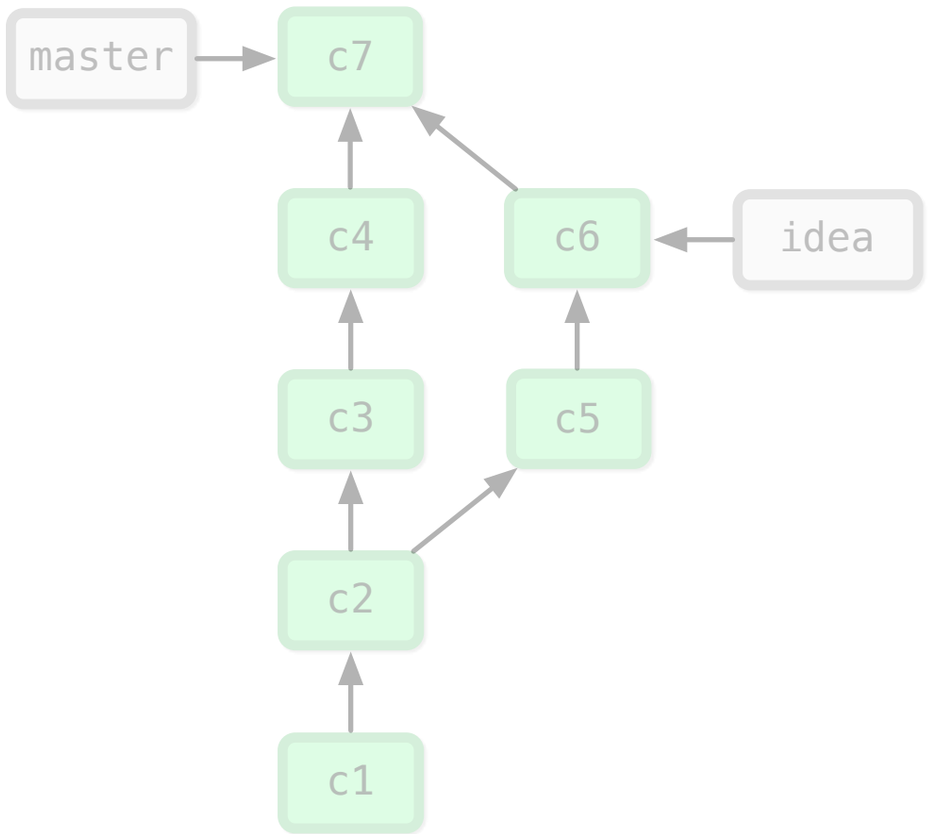


merge

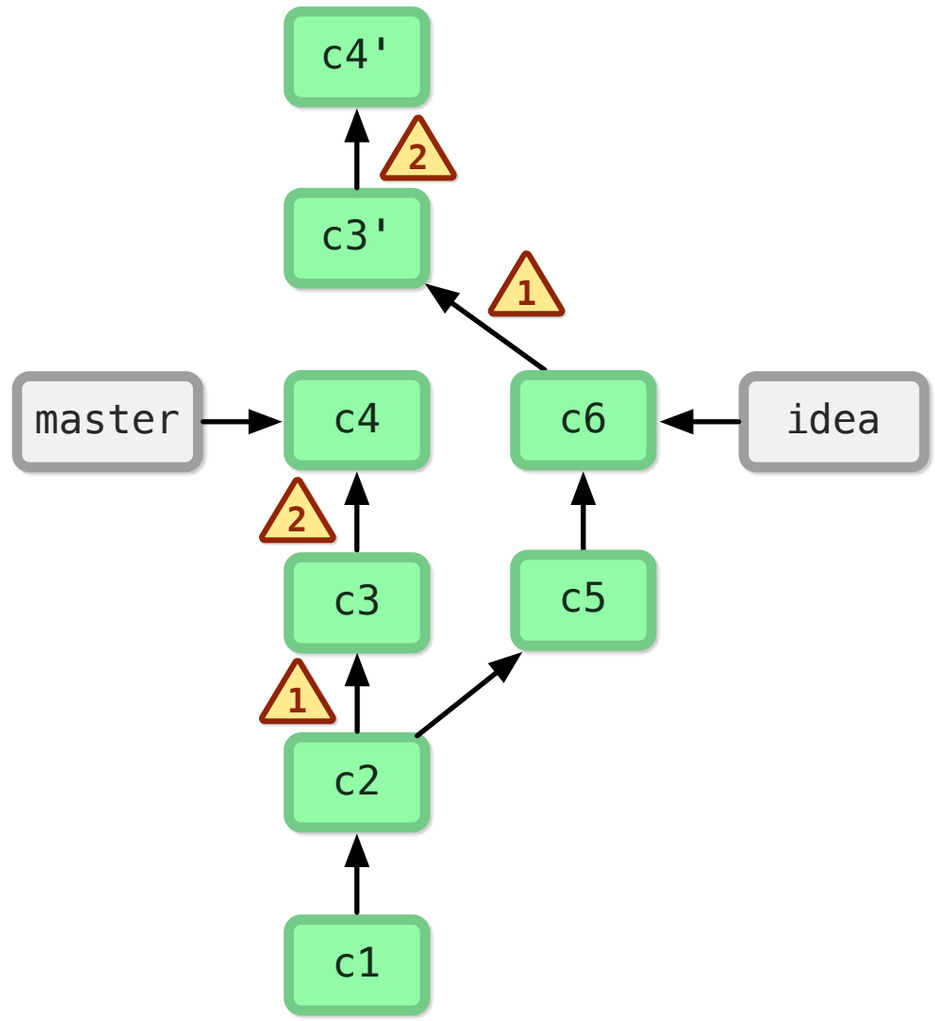


rebase

Merge vs. Rebase

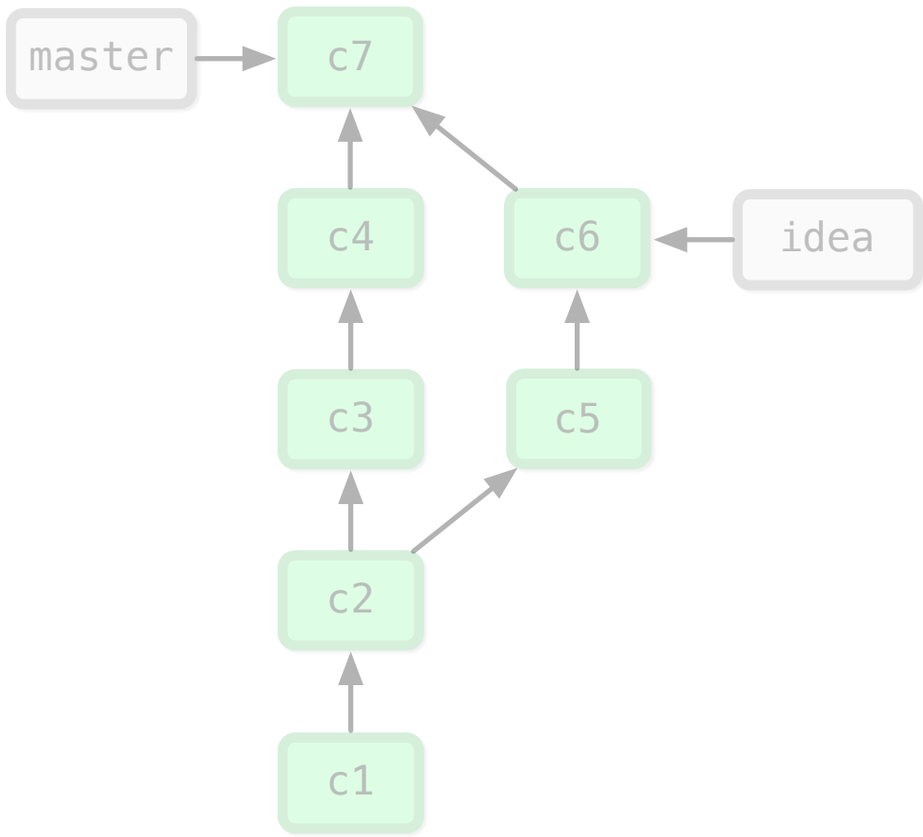


merge

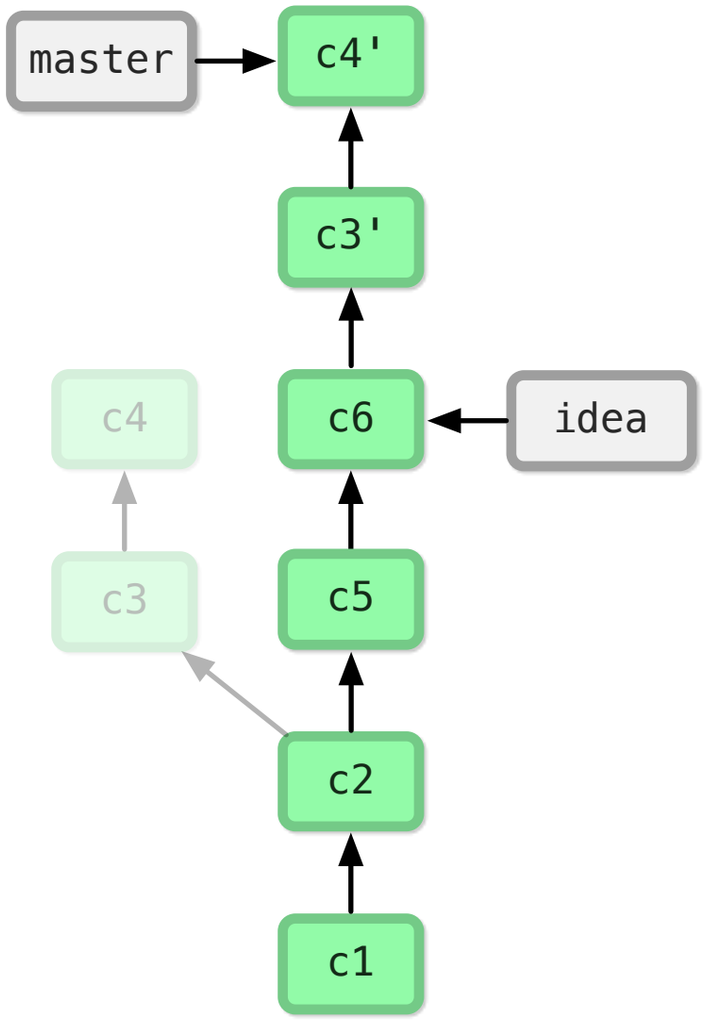


rebase

Merge vs. Rebase

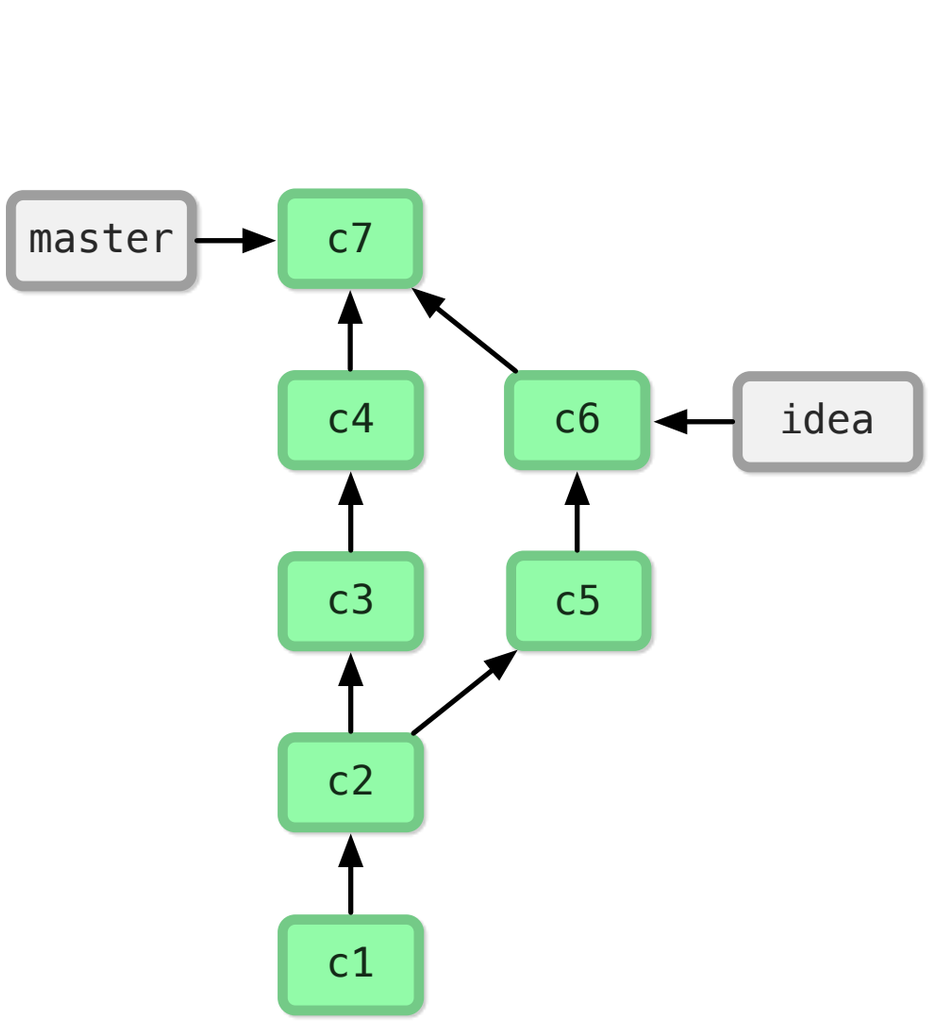


merge

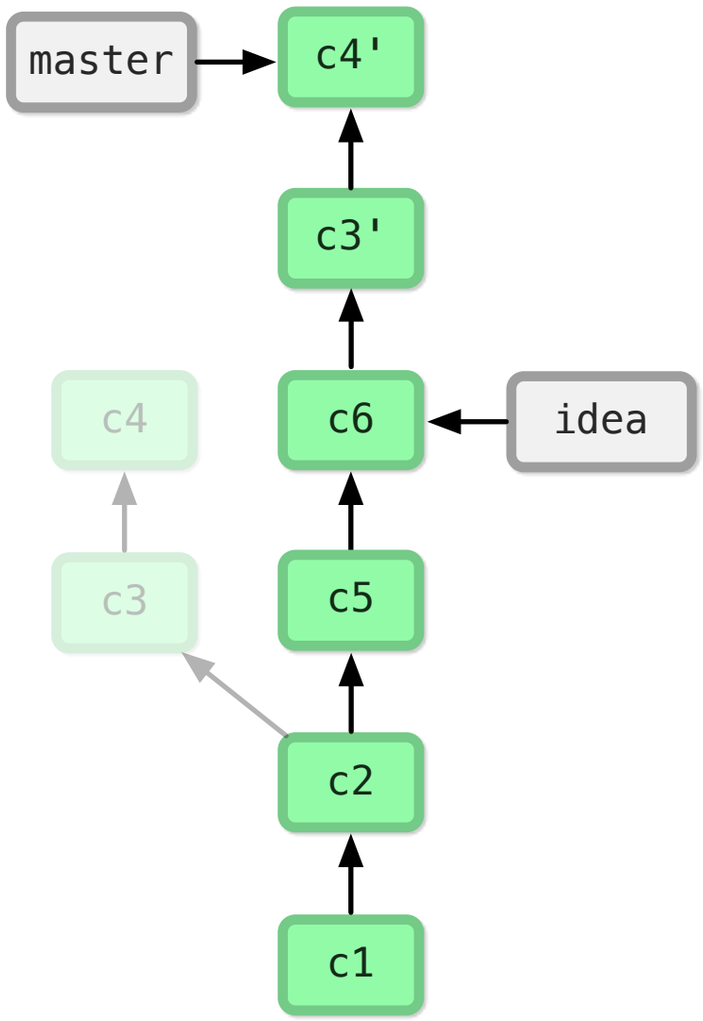


rebase

Merge vs. Rebase



merge



rebase

Remotes

Git does not have a concept of a central server.
It only has the concept of nodes -- other repositories.
That can be another computer, or somewhere else on your file system.

Remotes

`remote == URL`

Universal Repository Locator :)

Protocols

ssh://

http[s]://

git://

file://

rsync://

ftp://

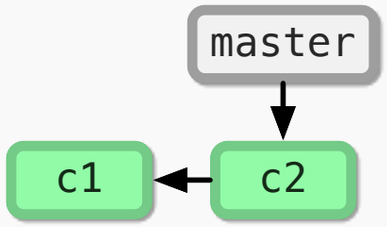
Protocols

push	ssh://	pull
	http[s]://	pull
	git://	pull
push	file://	pull
~~push~~	~~rsync://~~	~~pull~~
~~push~~	~~ftp://~~	~~pull~~

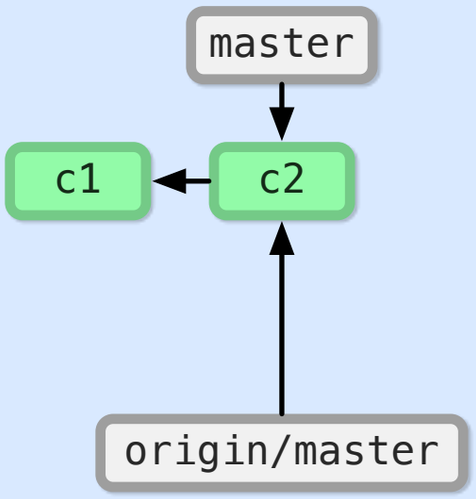
Ann's Computer

Bob's Computer

Company Server



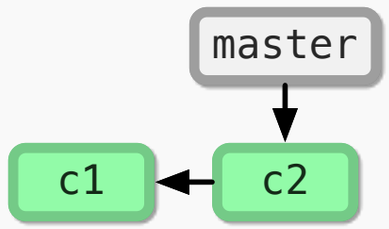
Ann's Computer

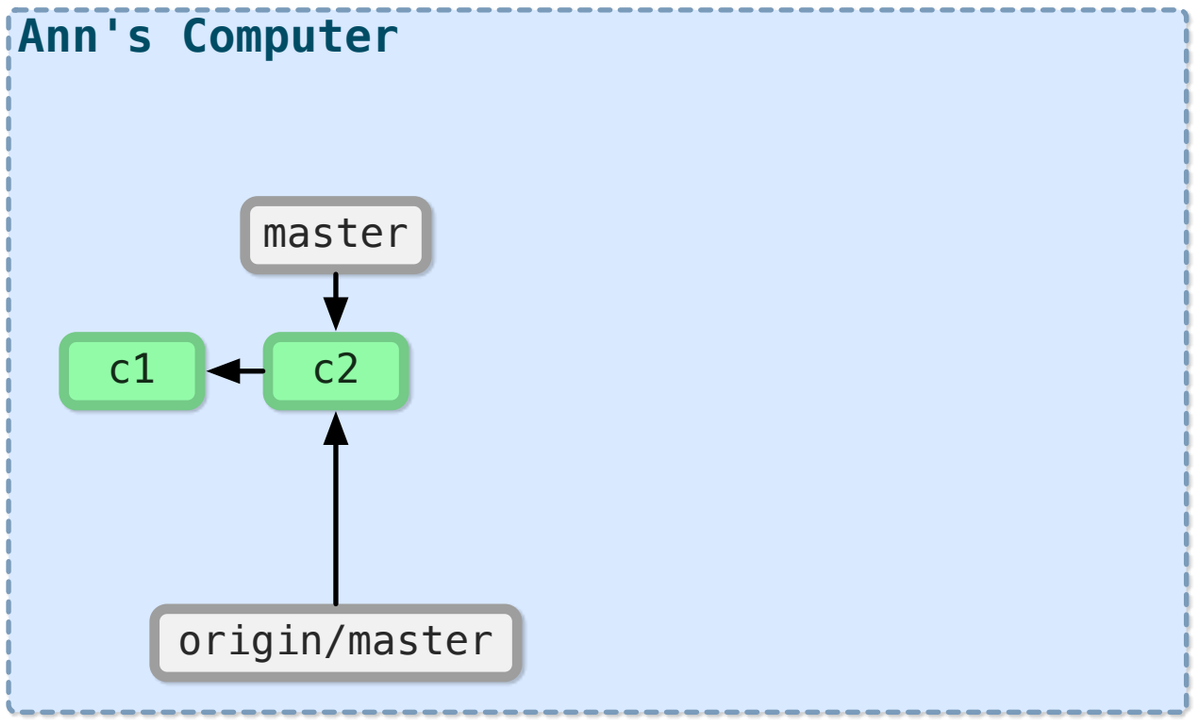


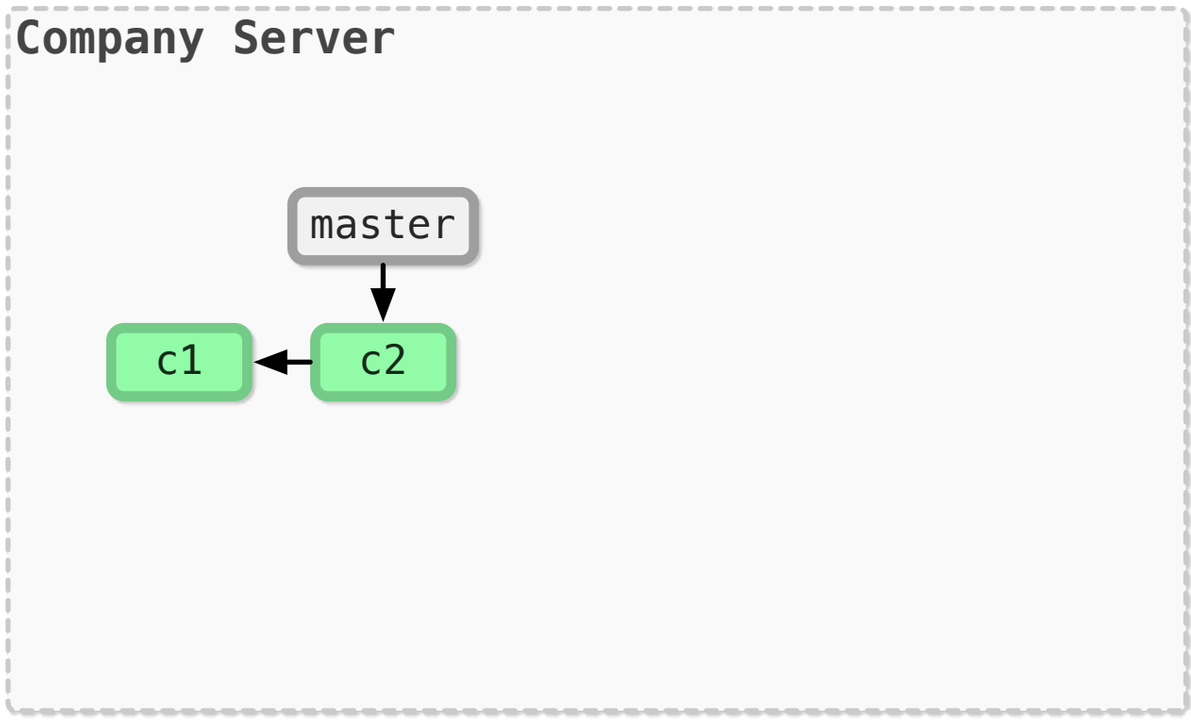
Bob's Computer

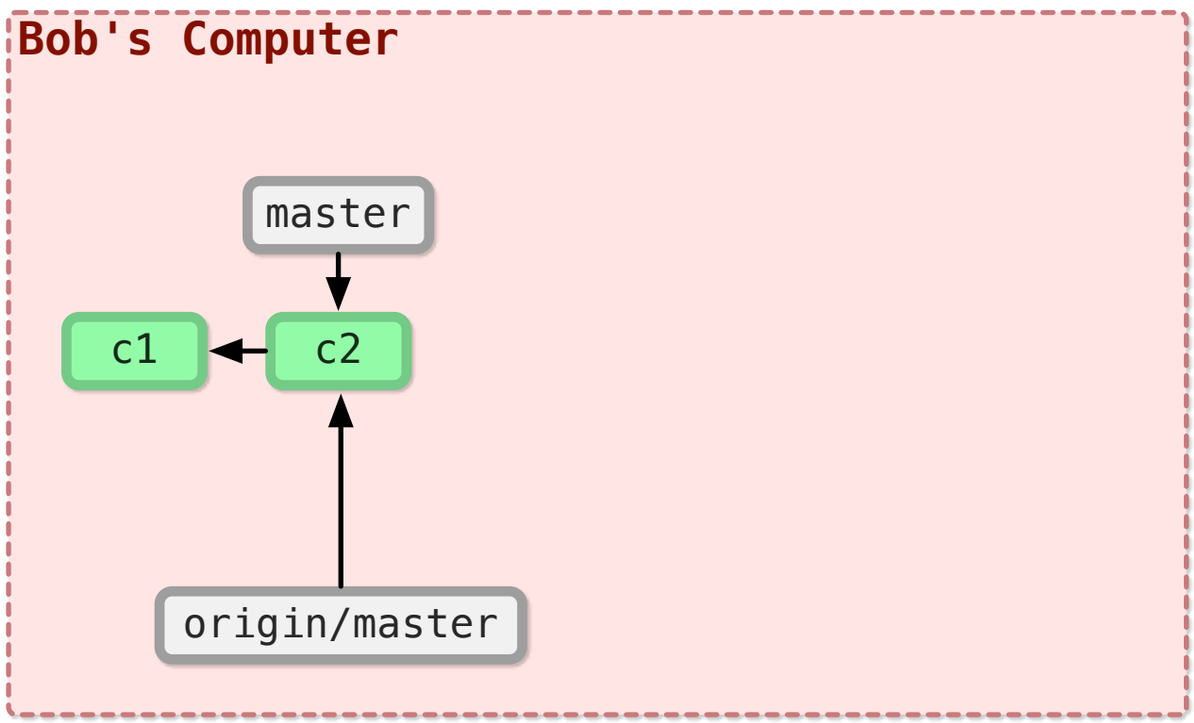
```
git clone ann@git.company.com:project.git
```

Company Server



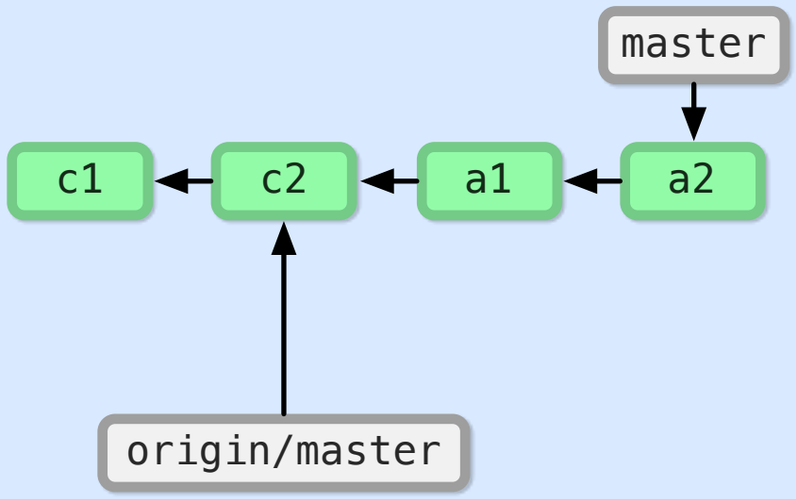






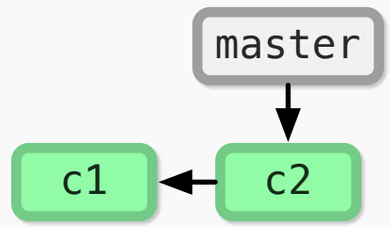

```
git clone bob@git.company.com:project.git
```

Ann's Computer

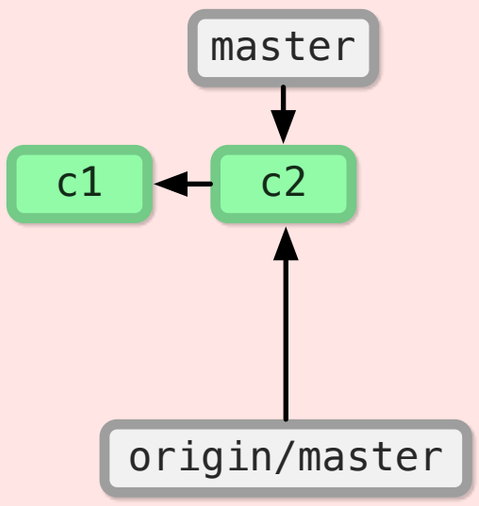



```
git commit -a -m "Ann's new feature"
```

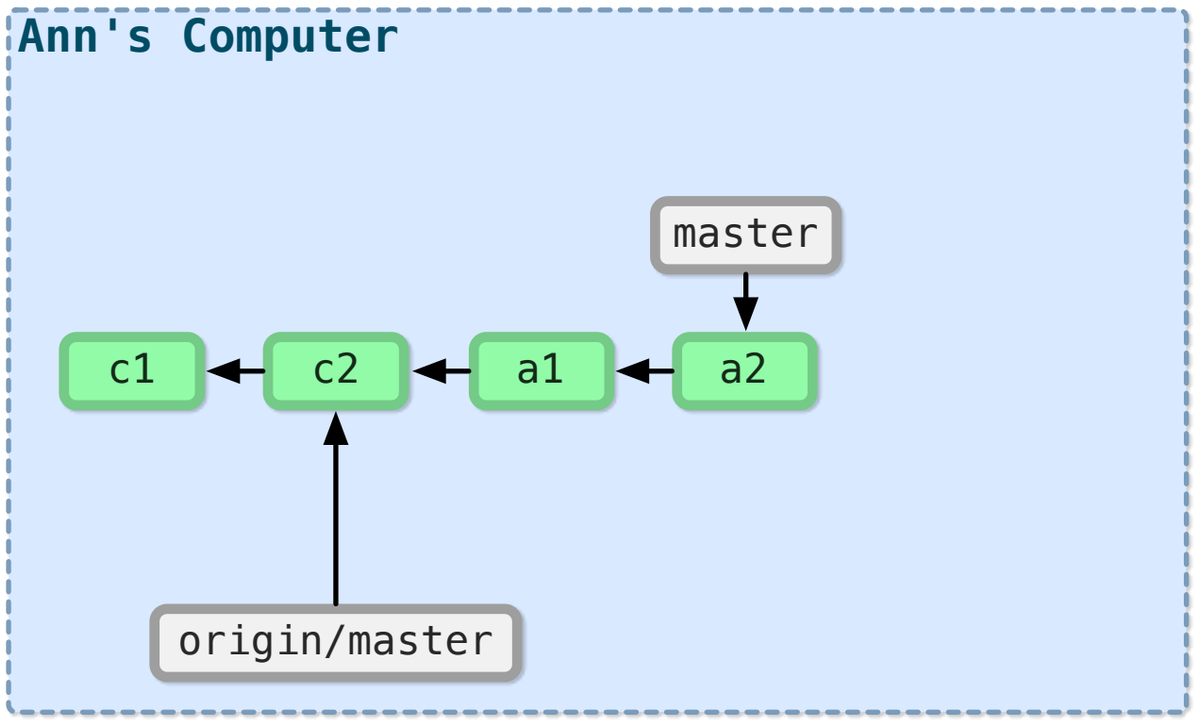
Company Server

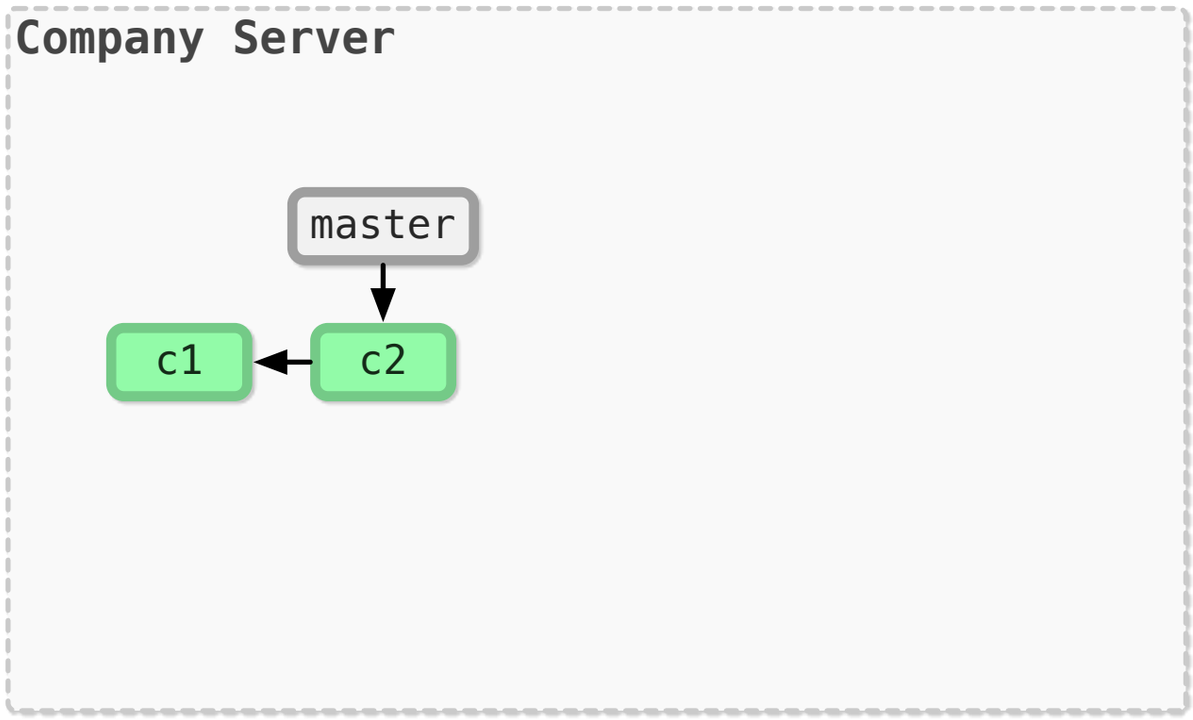


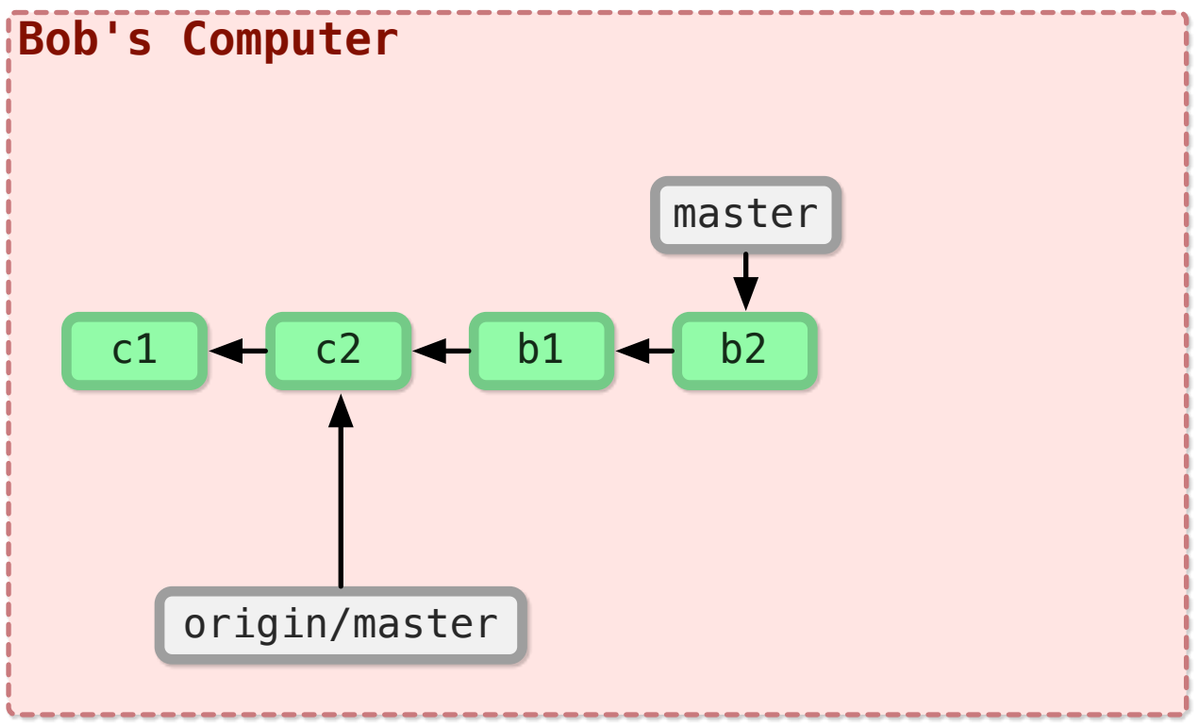
Bob's Computer



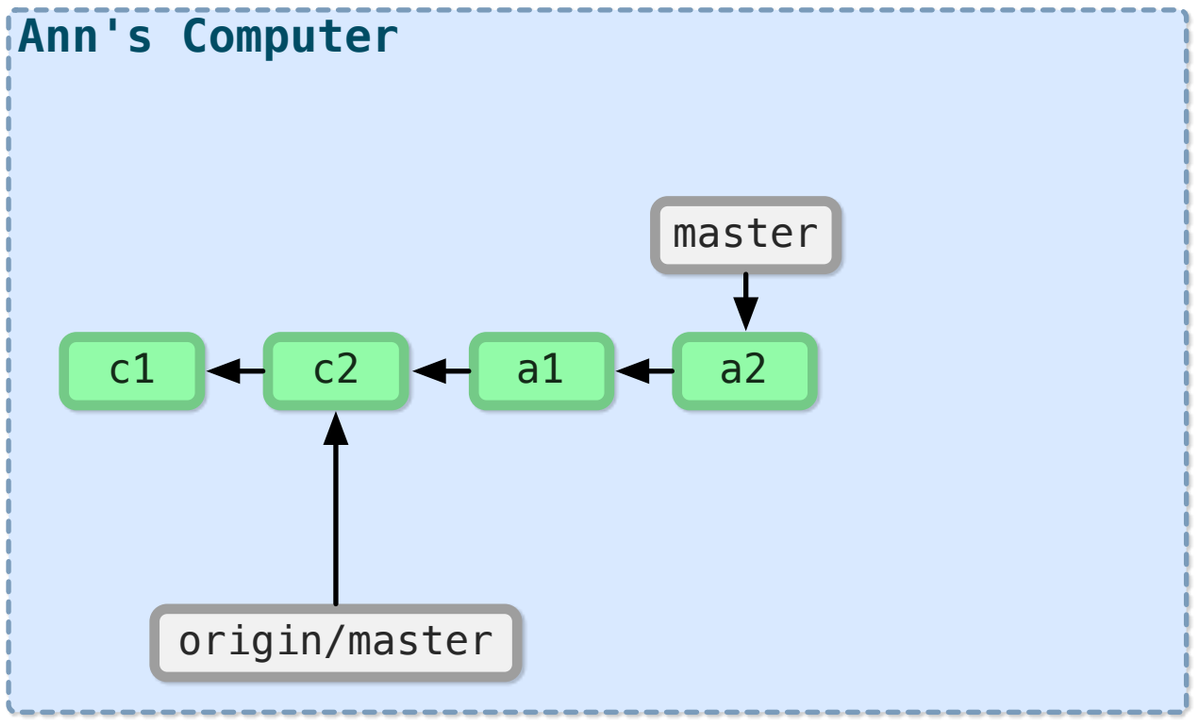

```

```

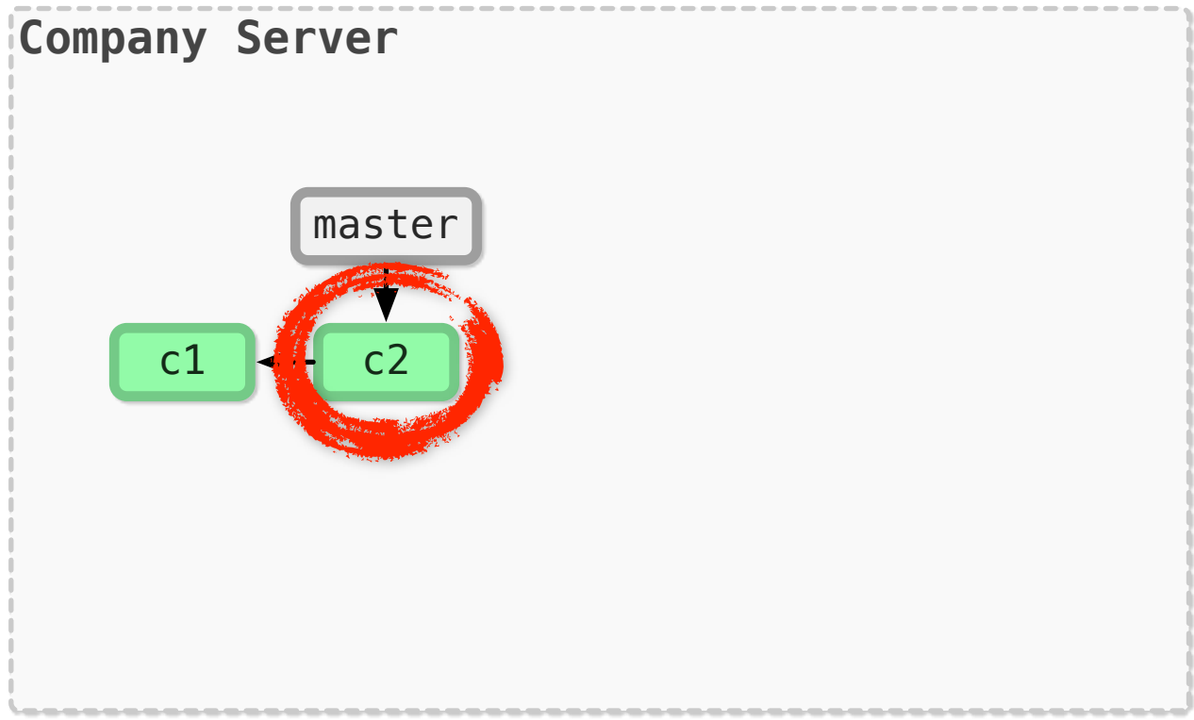


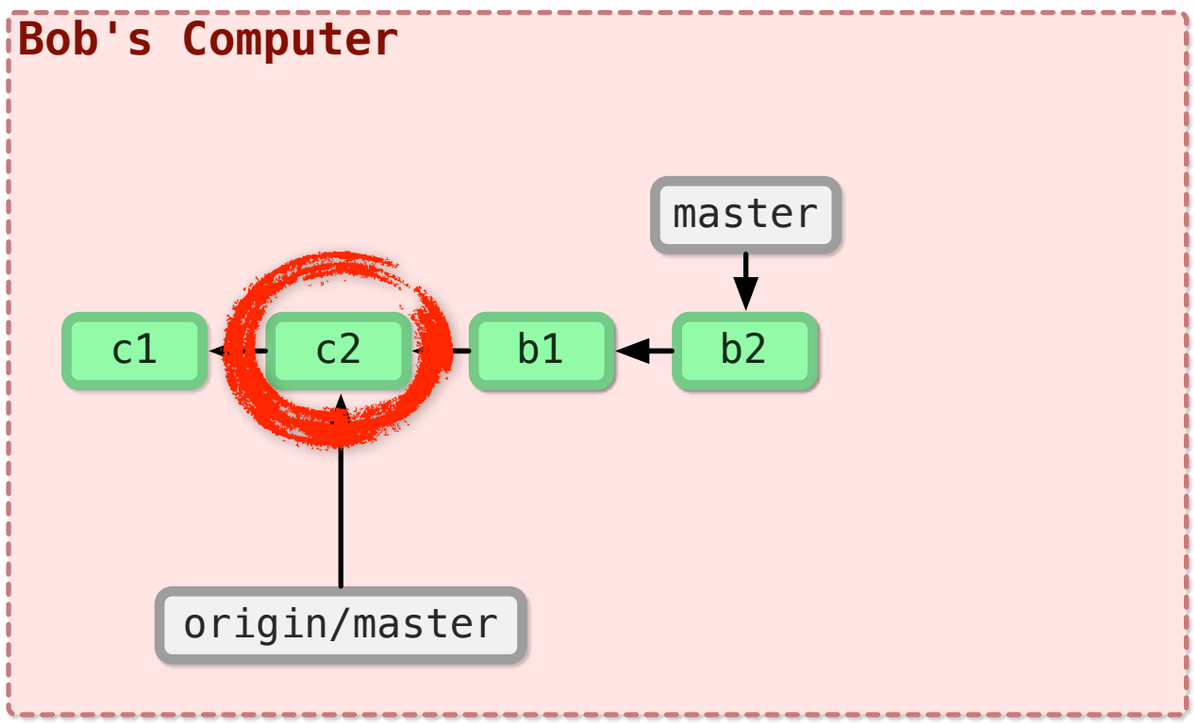




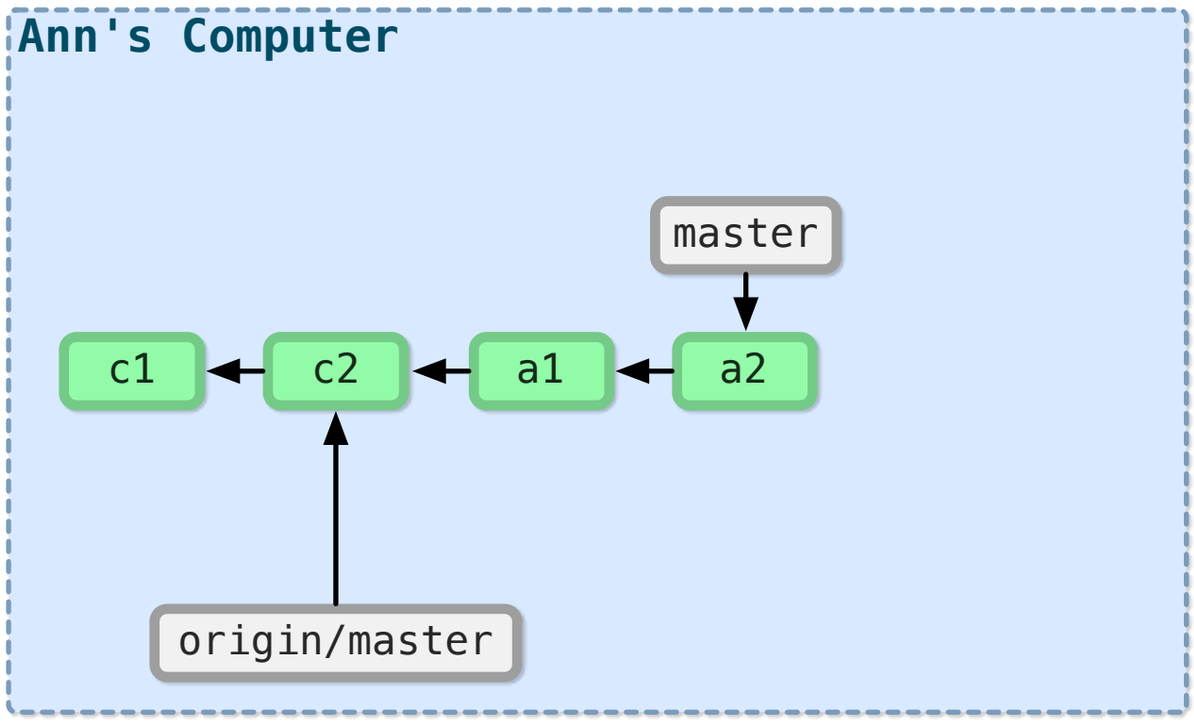

```
git commit -a -m "Bob's new feature"
```

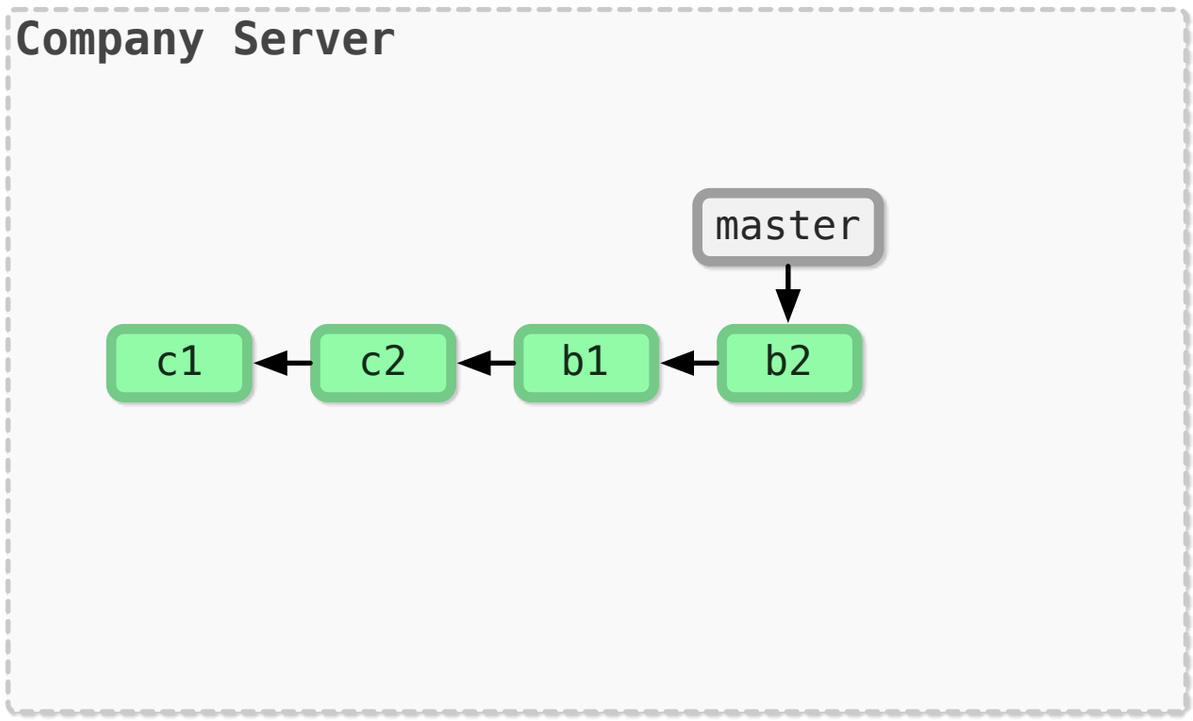


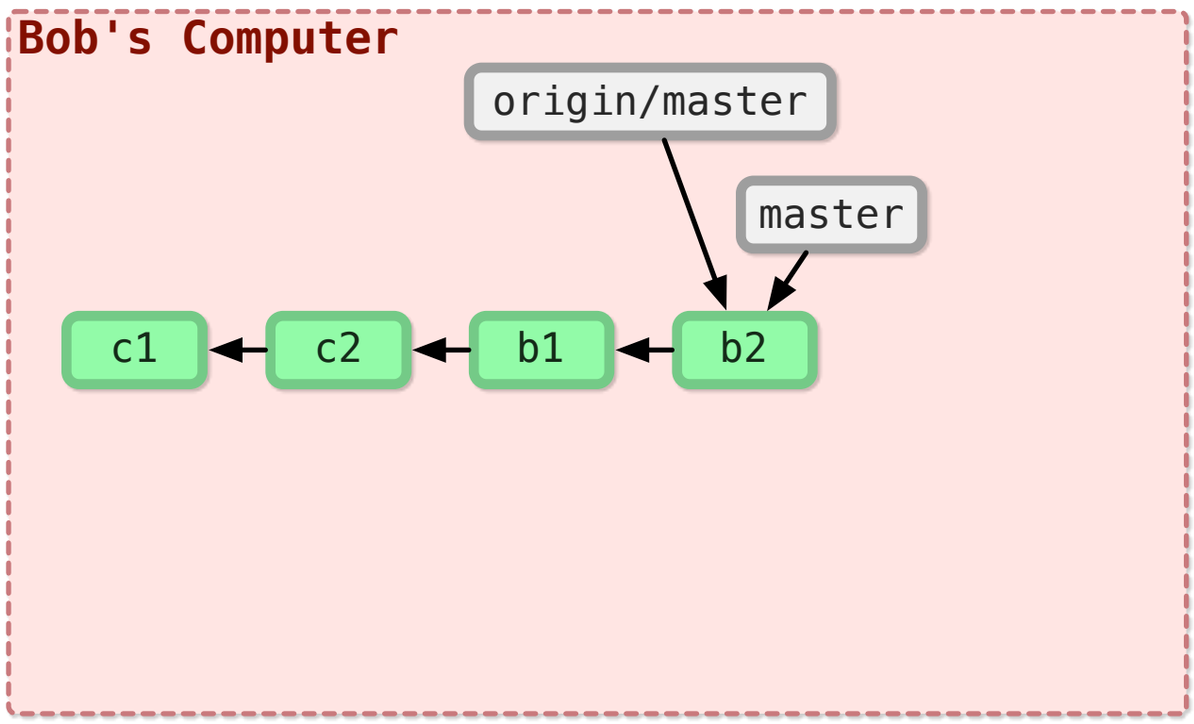




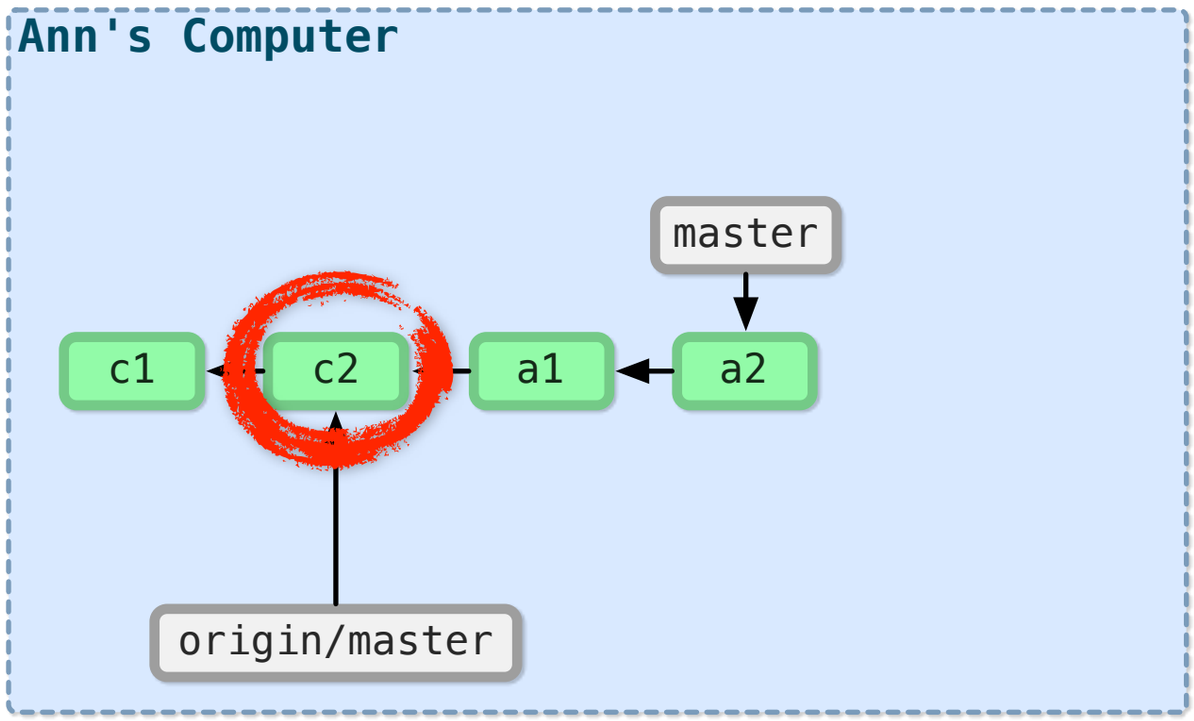

```
git push origin master
```

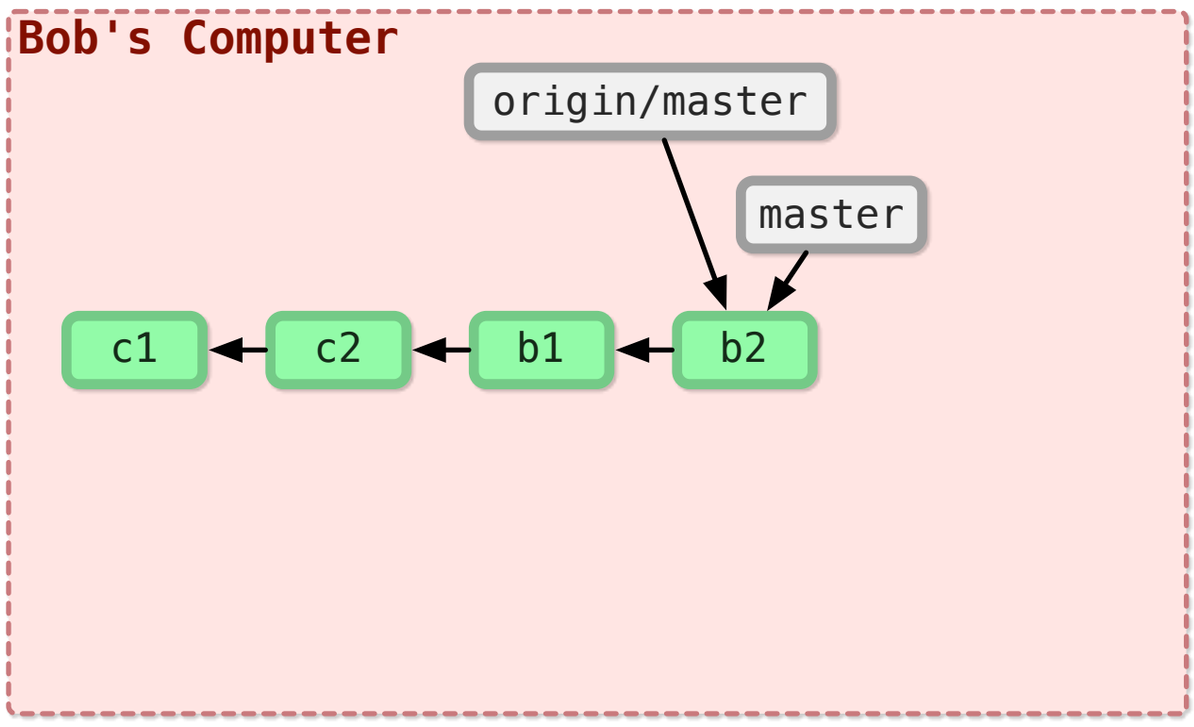




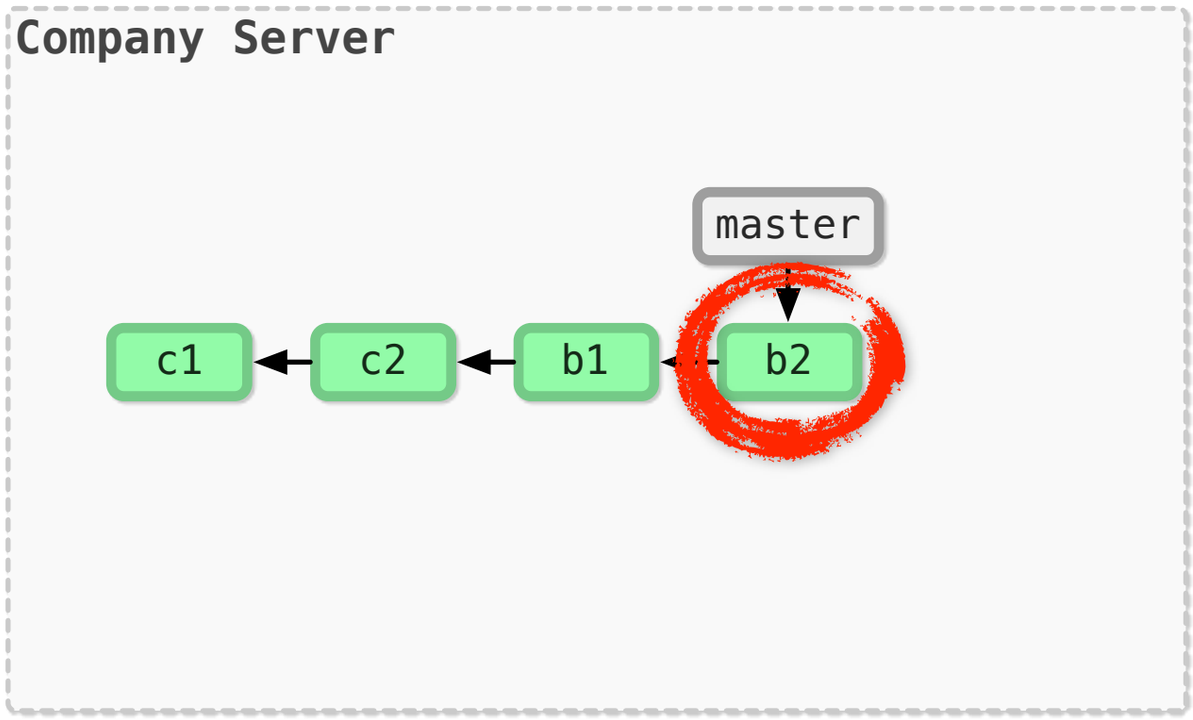



```
git push origin master
```

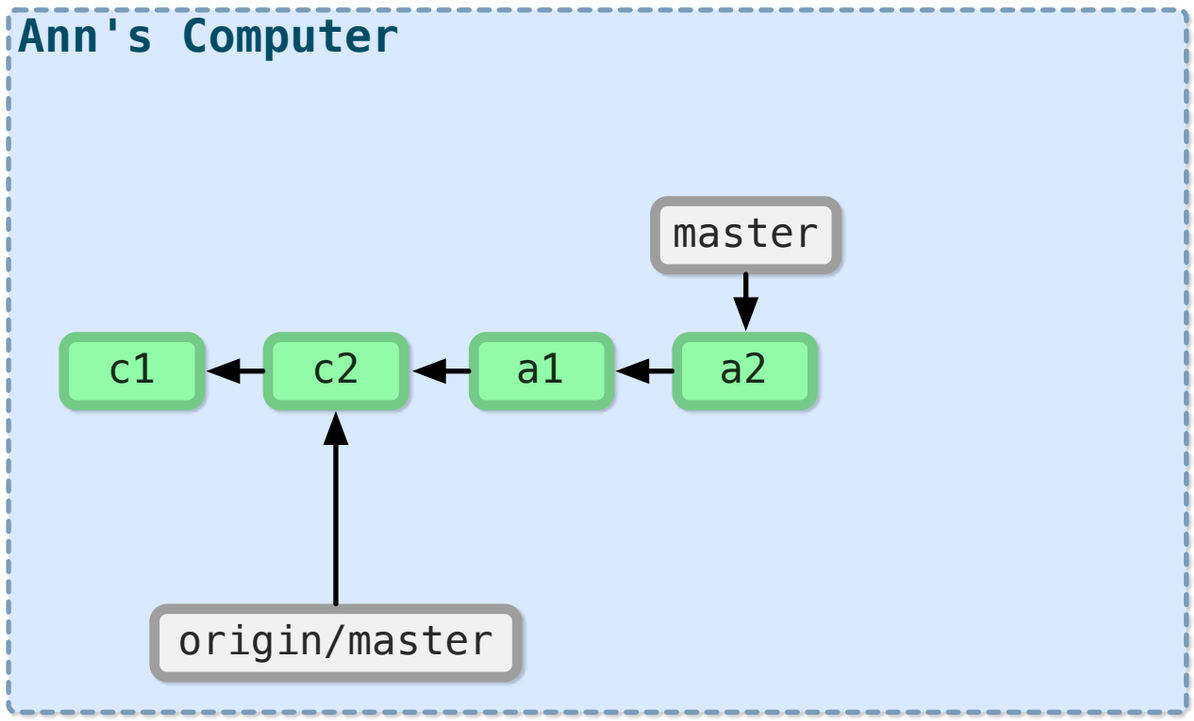


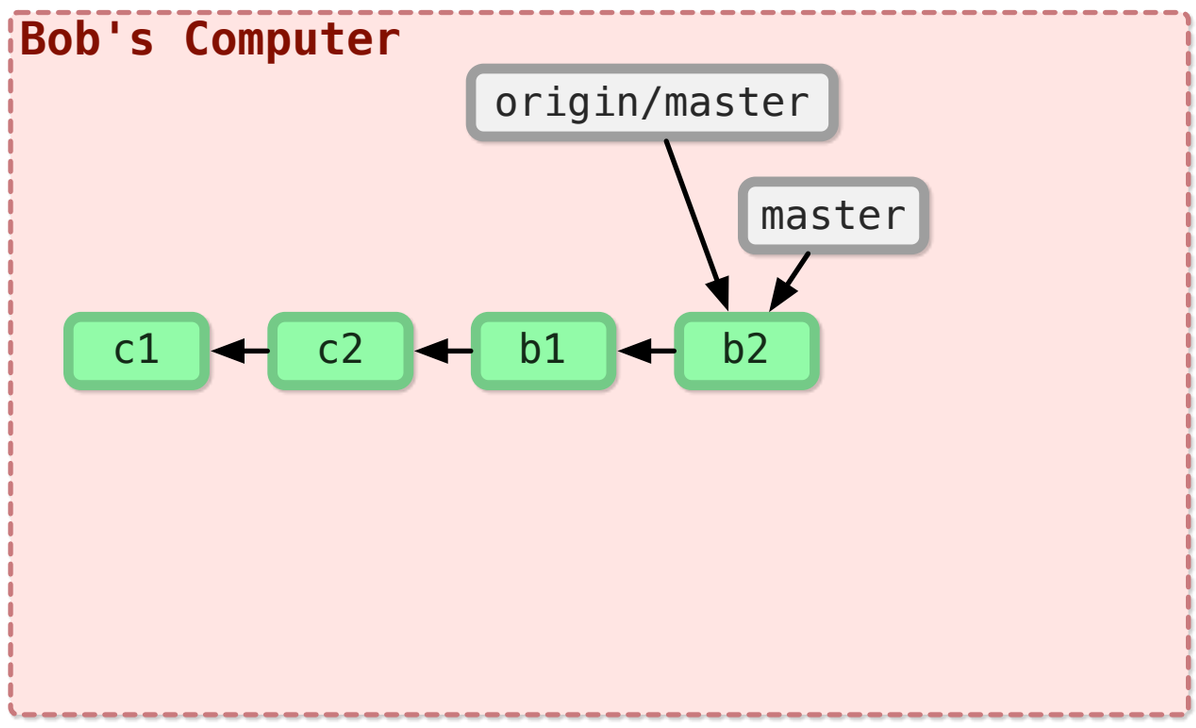



```
git push origin master
```

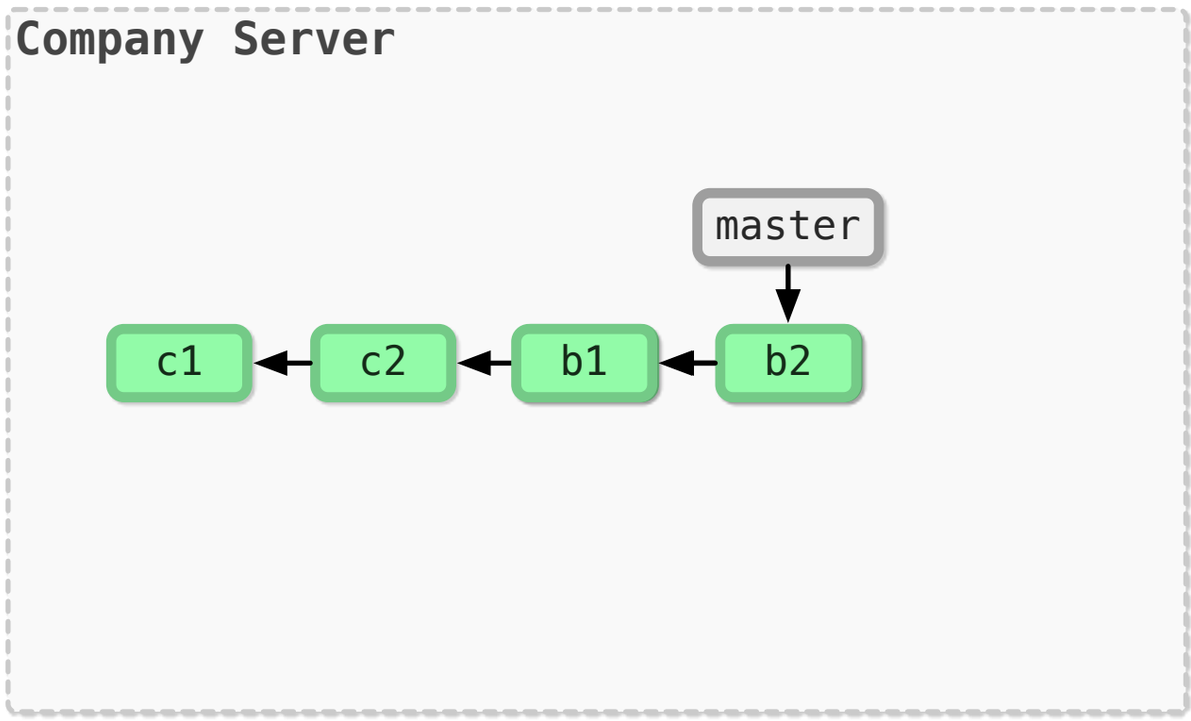


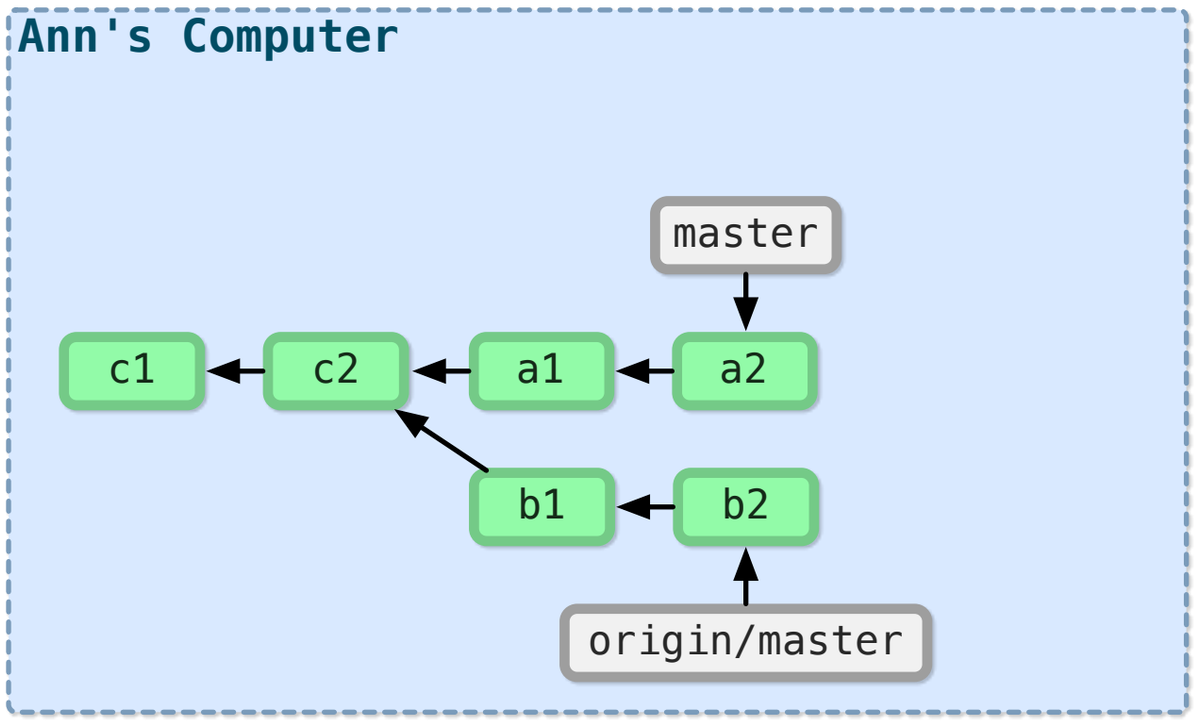

```

```

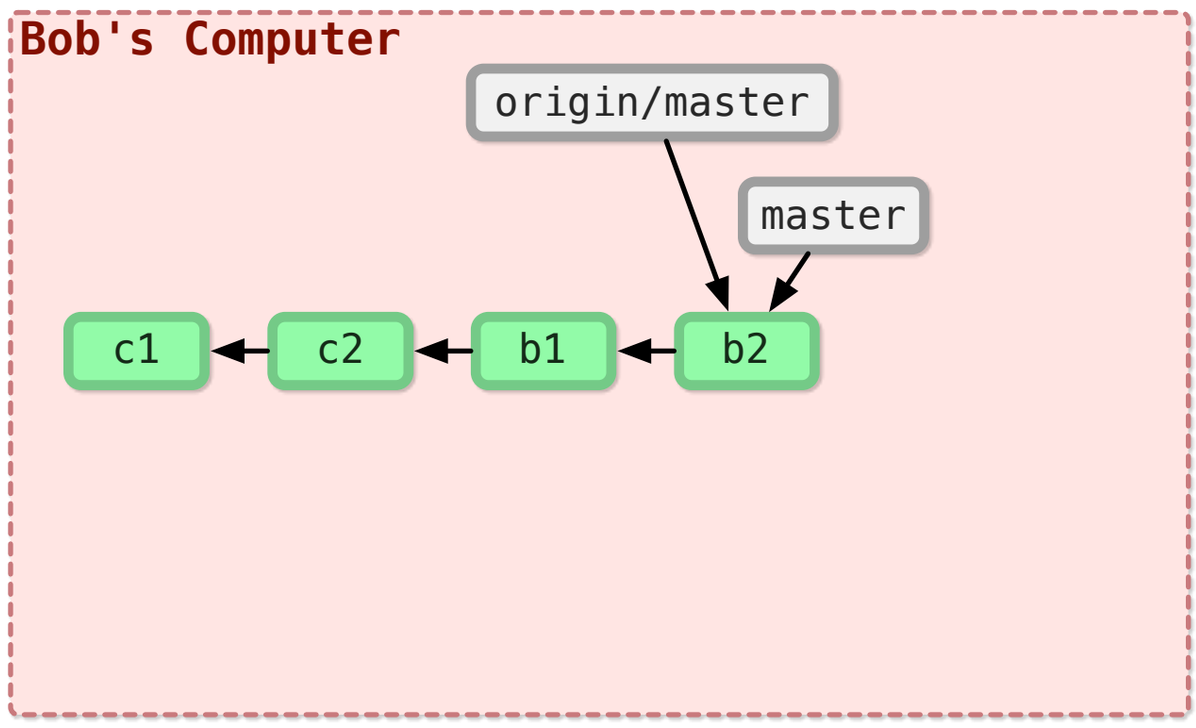

Ann's Computer
master
c1
c2
a1
a2
origin/master


Bob's Computer
origin/master
master
c1
c2
b1
b2

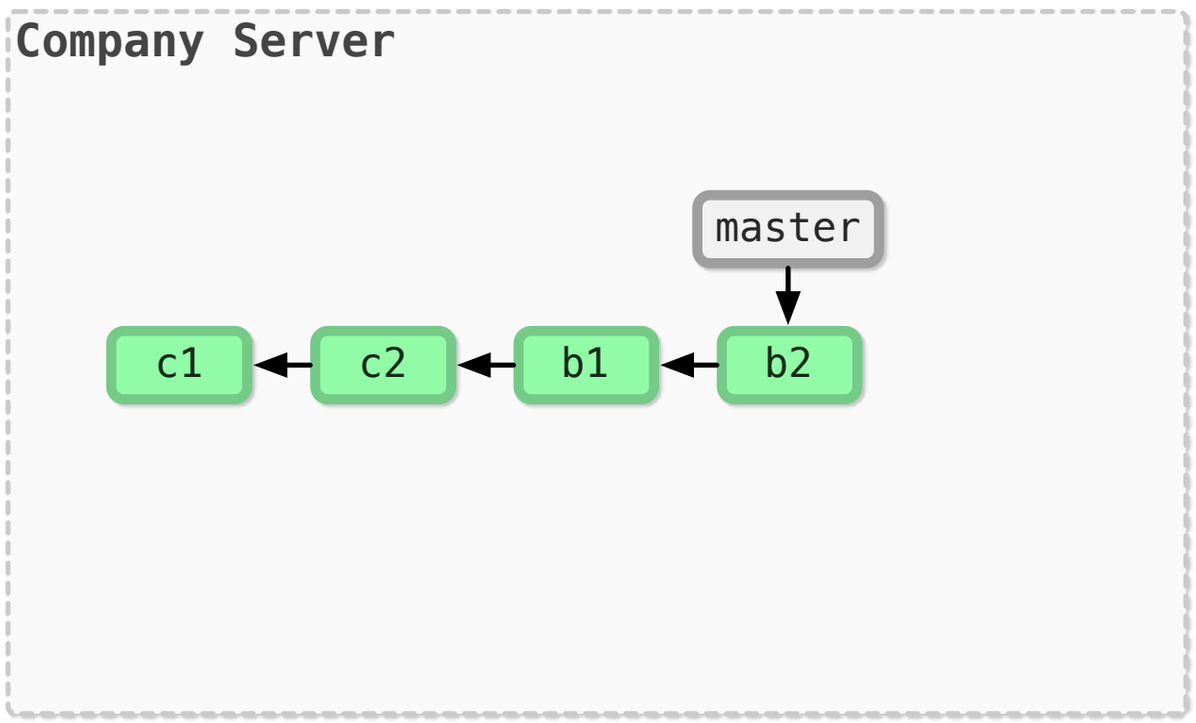
```
git fetch
```

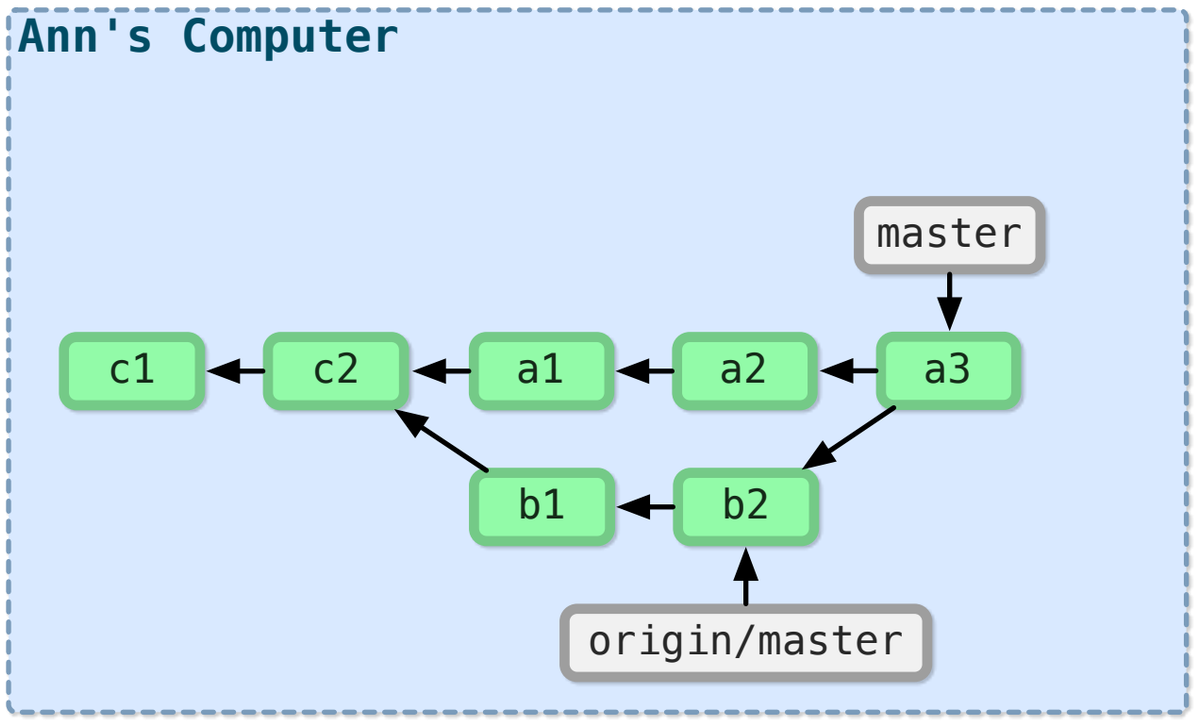

Company Server
master
c1
c2
b1
b2

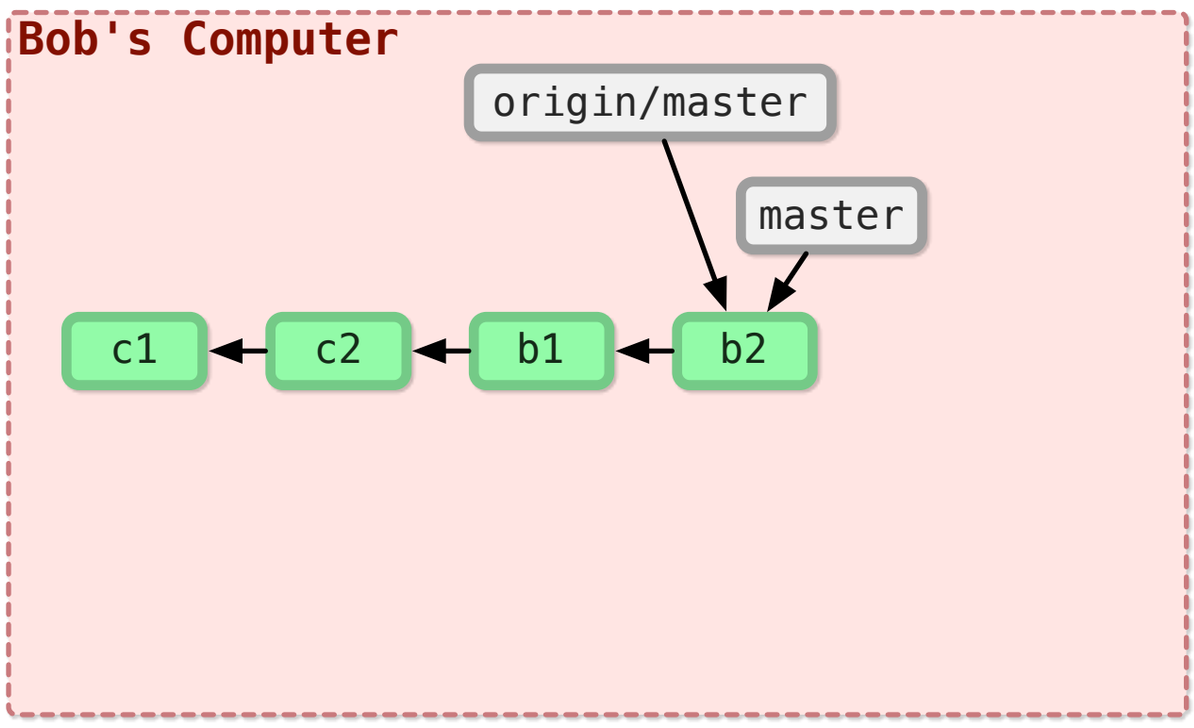

Ann's Computer
master
c1
c2
a1
a2
b1
b2
origin/master


Bob's Computer
origin/master
master
c1
c2
b1
b2

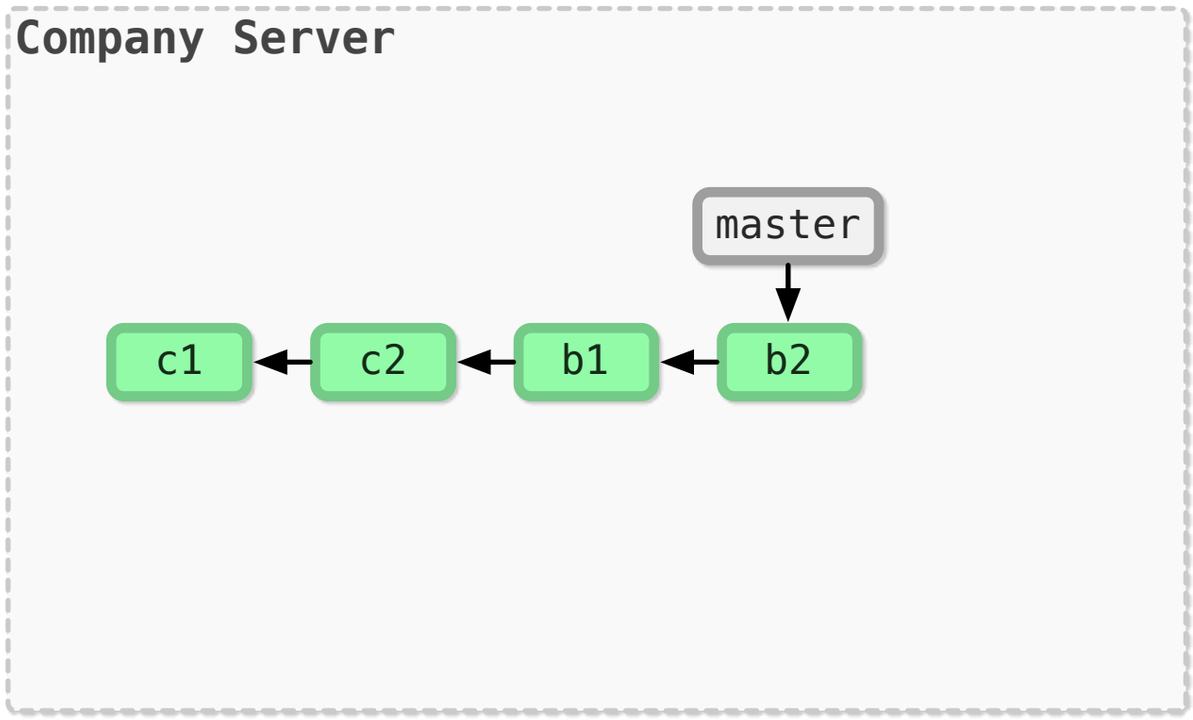
```
git fetch
```

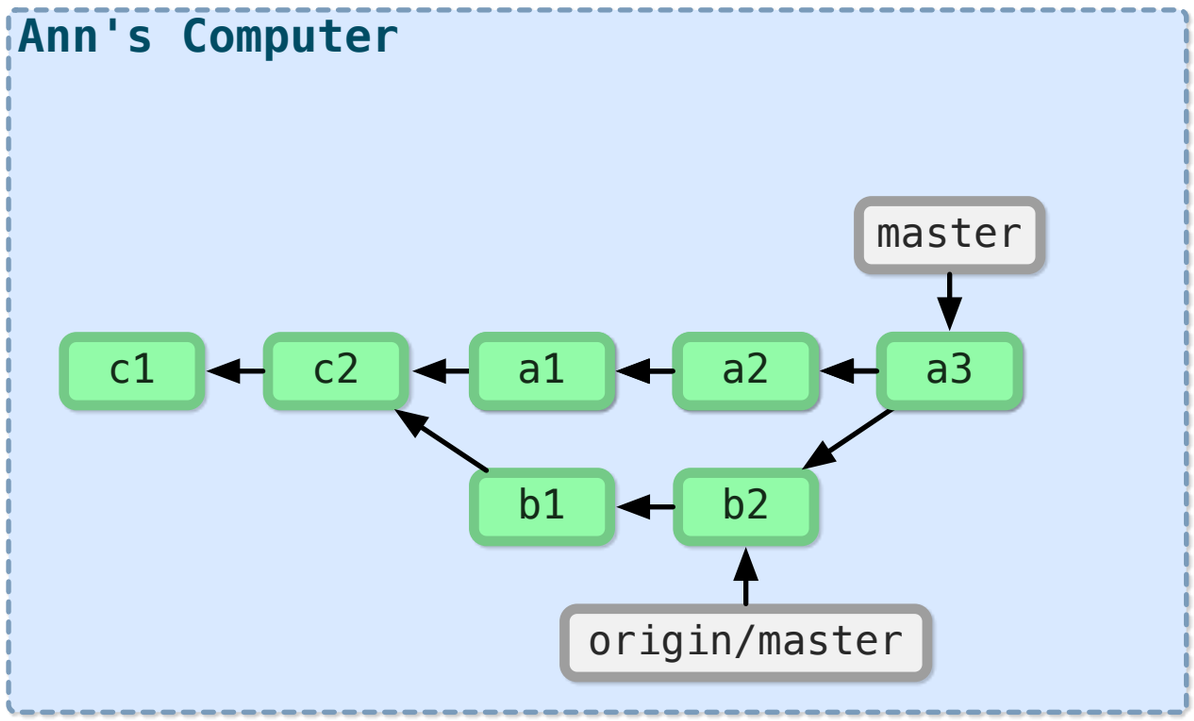

Company Server
master
c1
c2
b1
b2

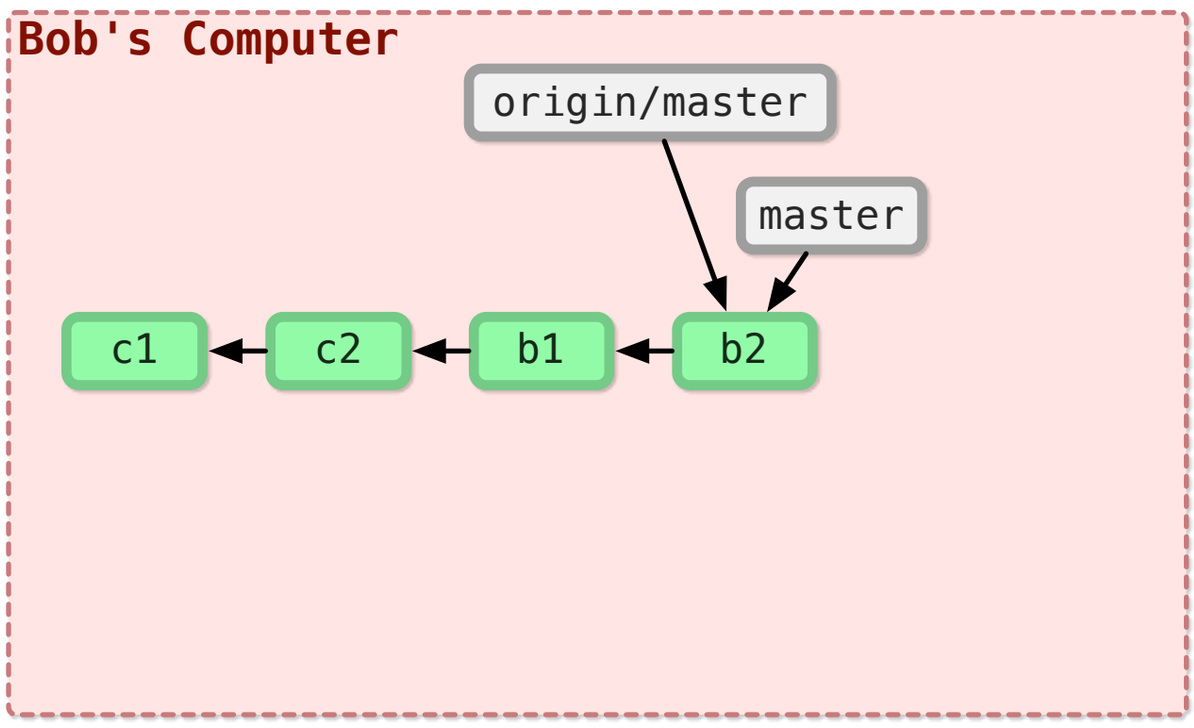




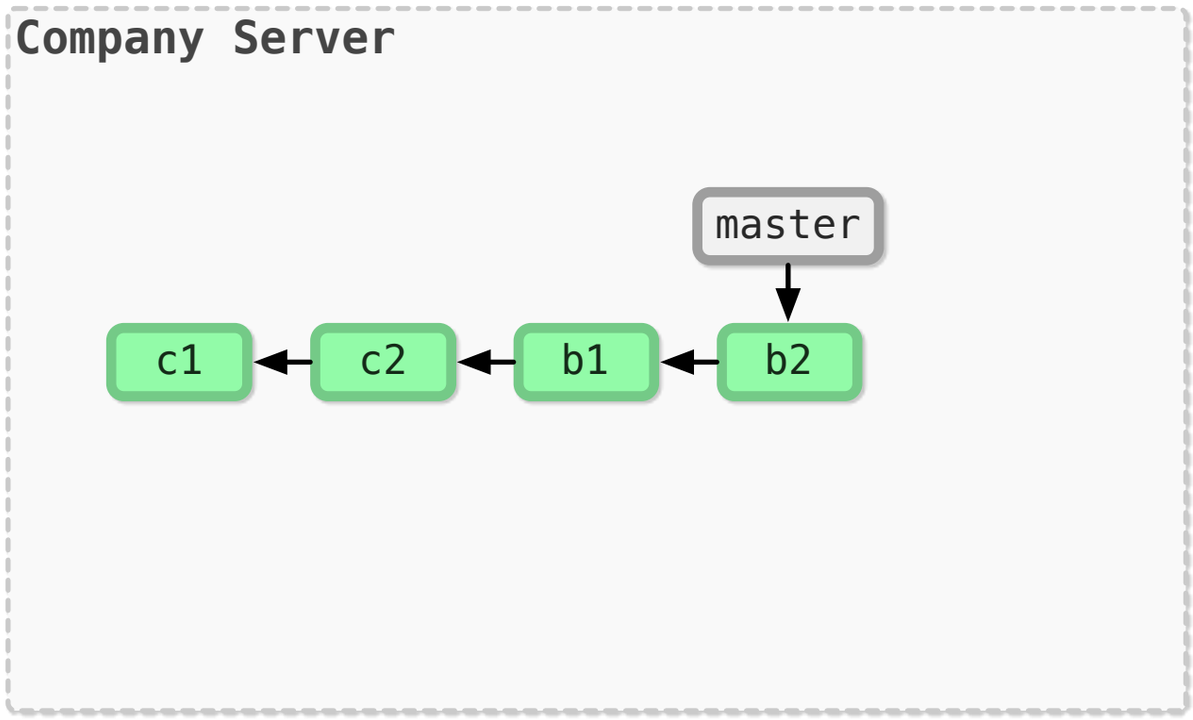

```
git merge origin/master
```

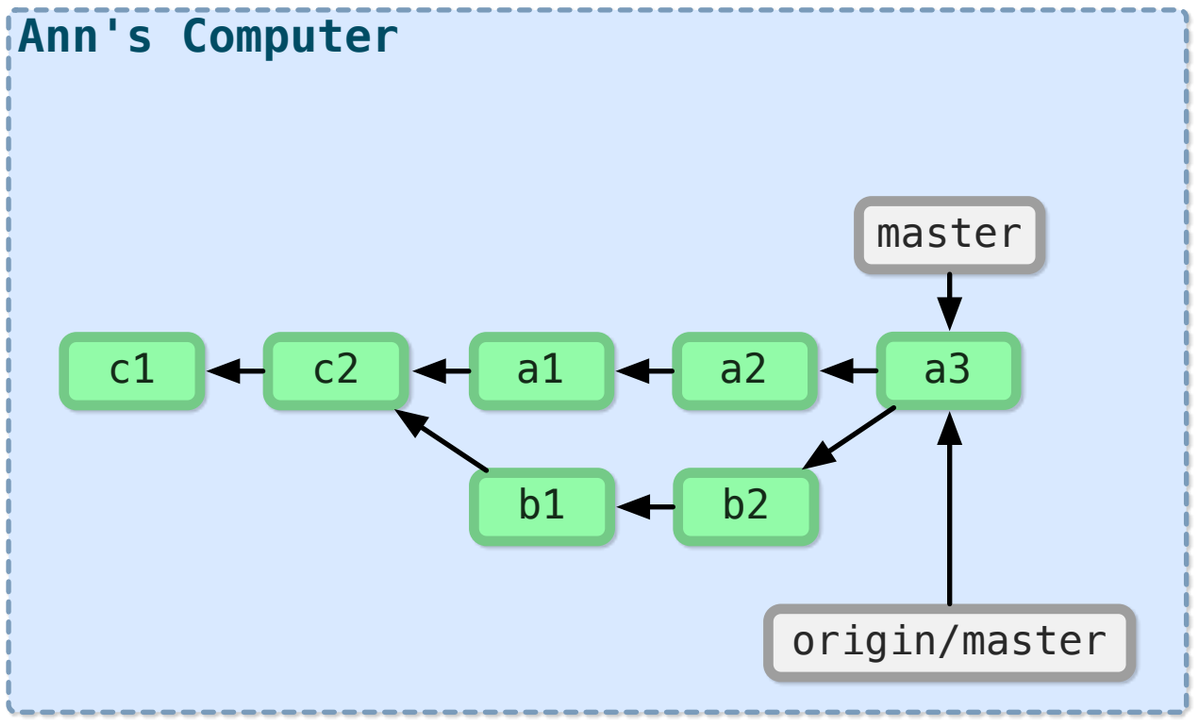


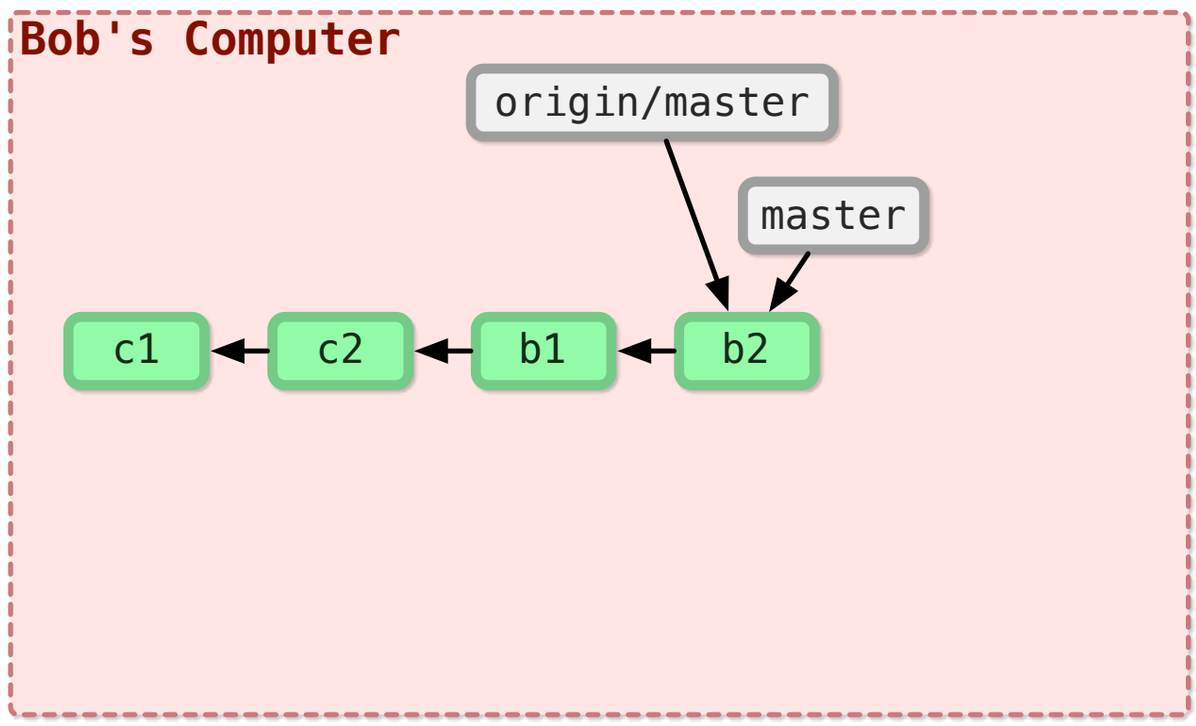




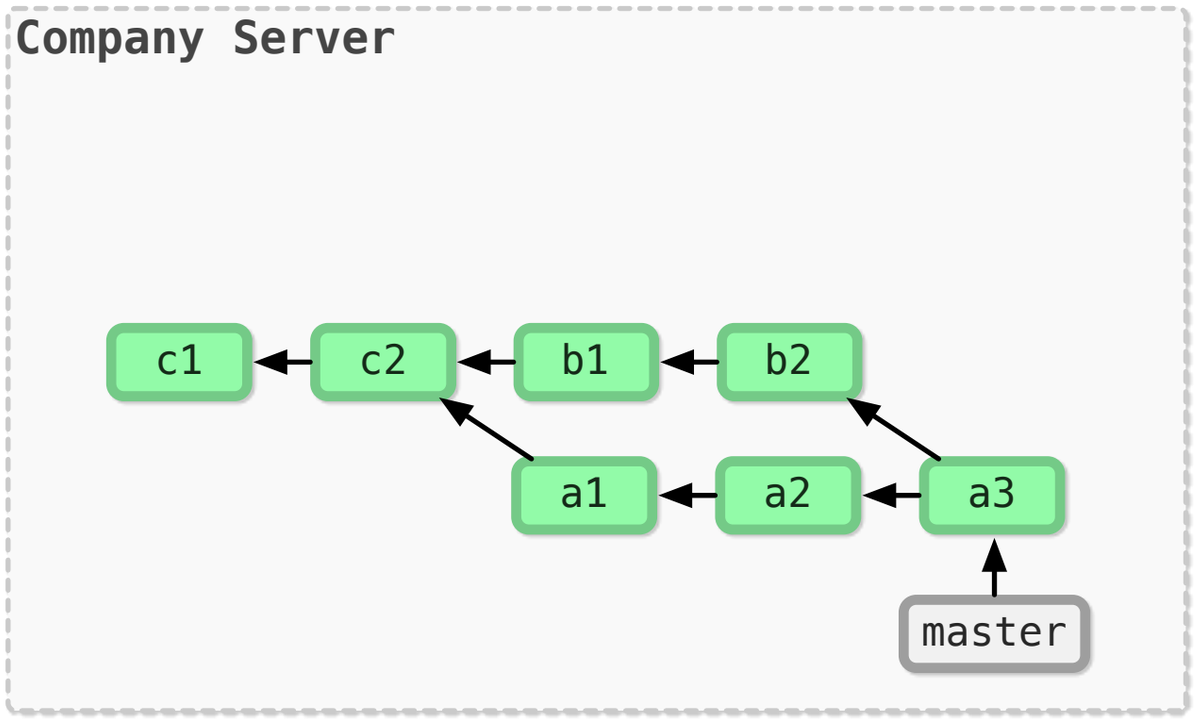

```
git push origin master
```

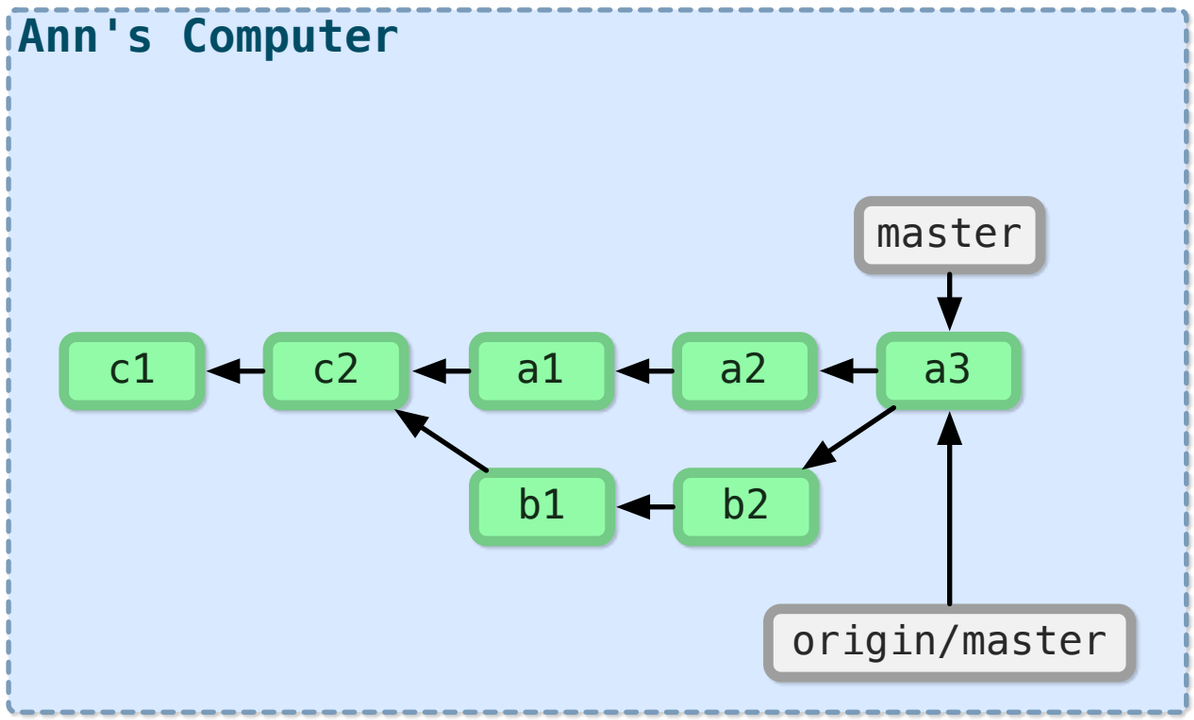


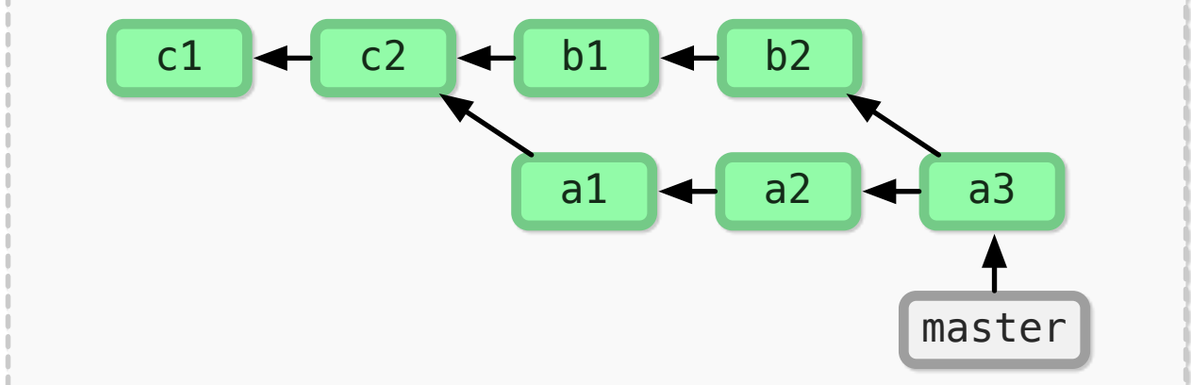


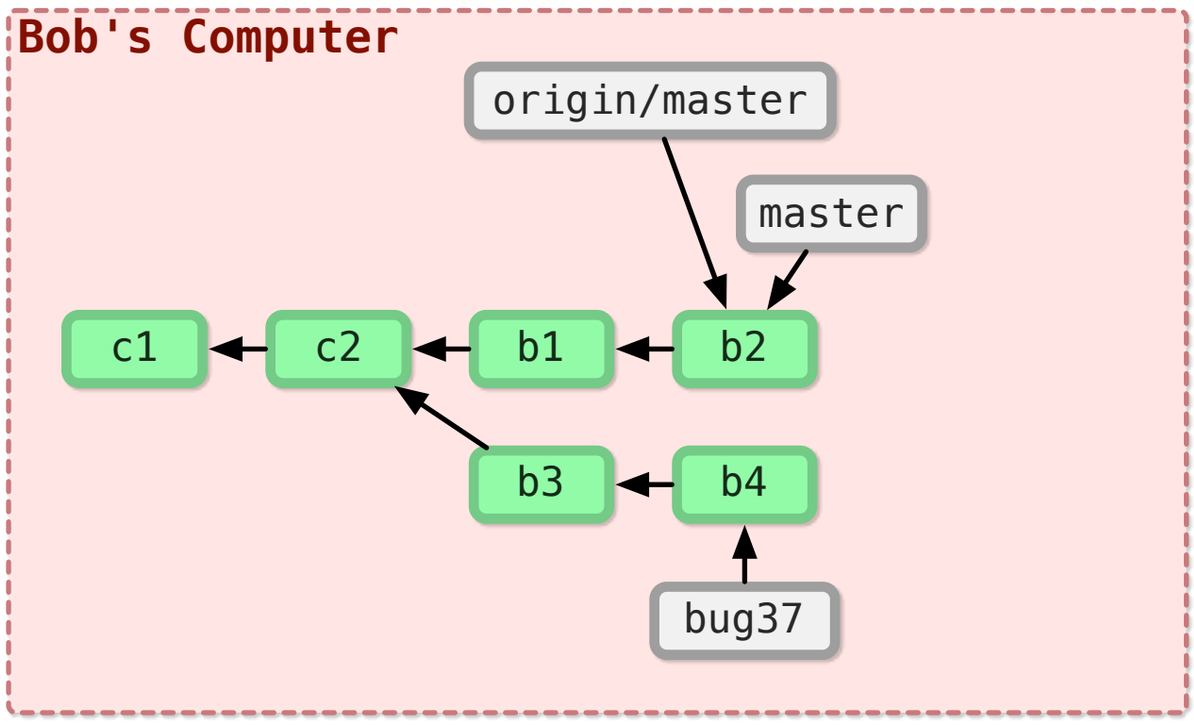



```
git push origin master
```

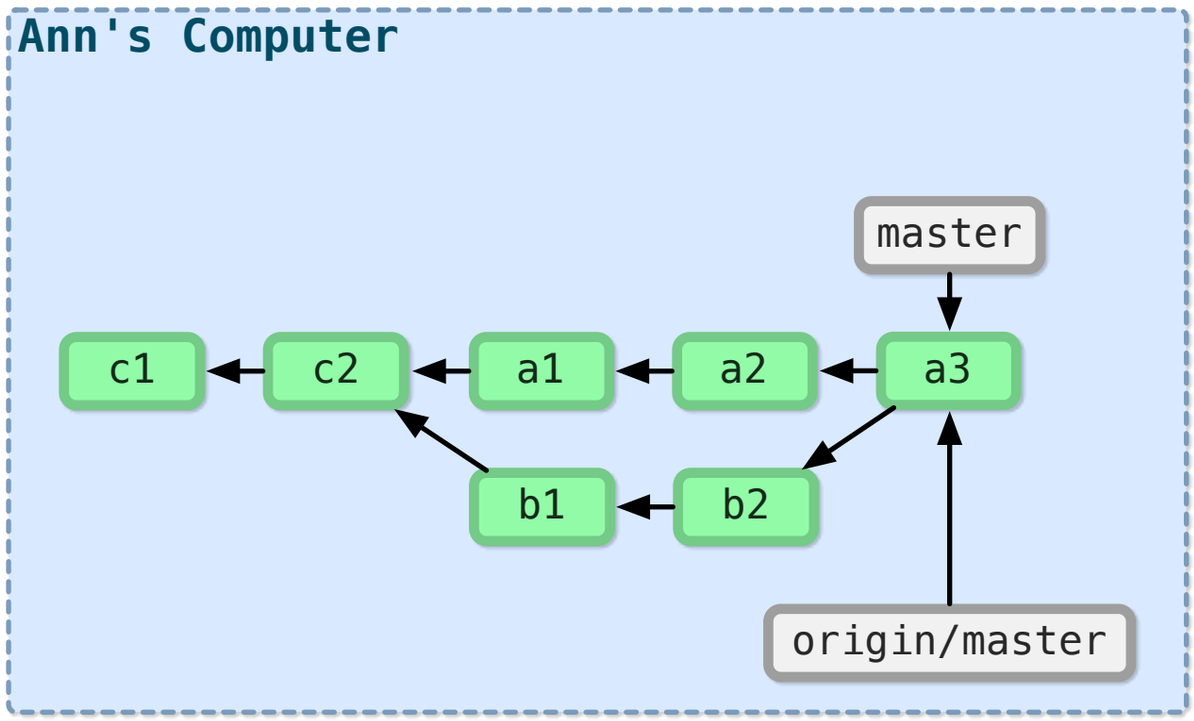


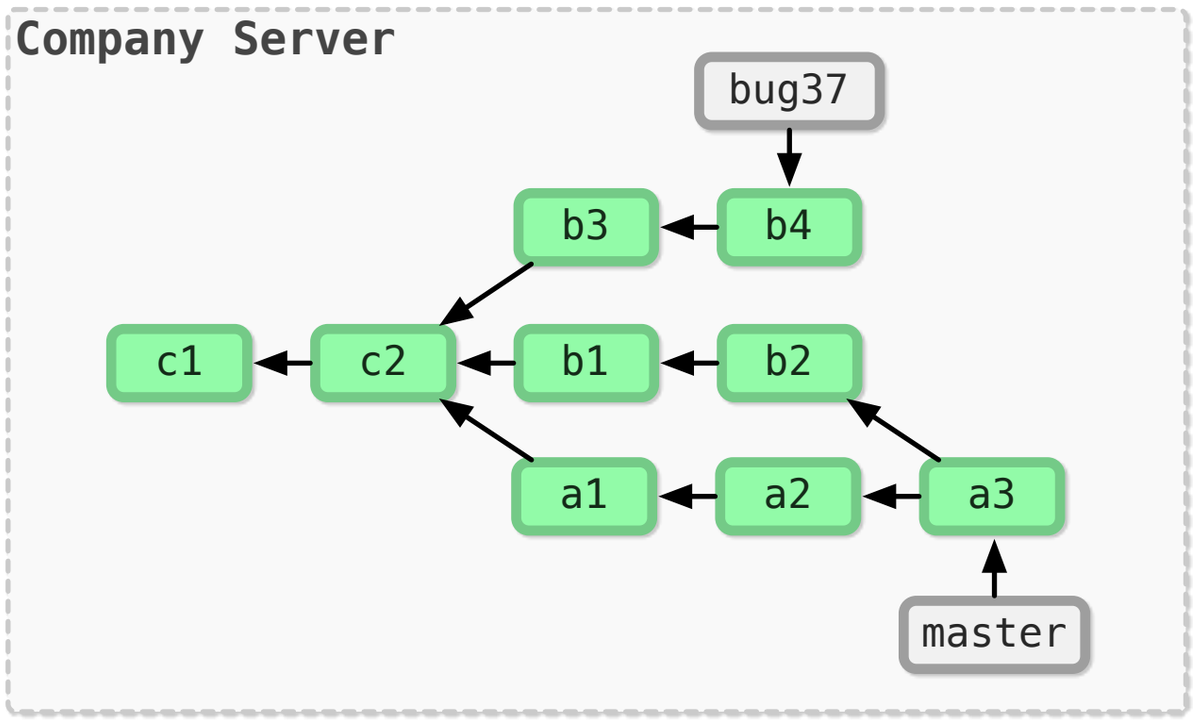


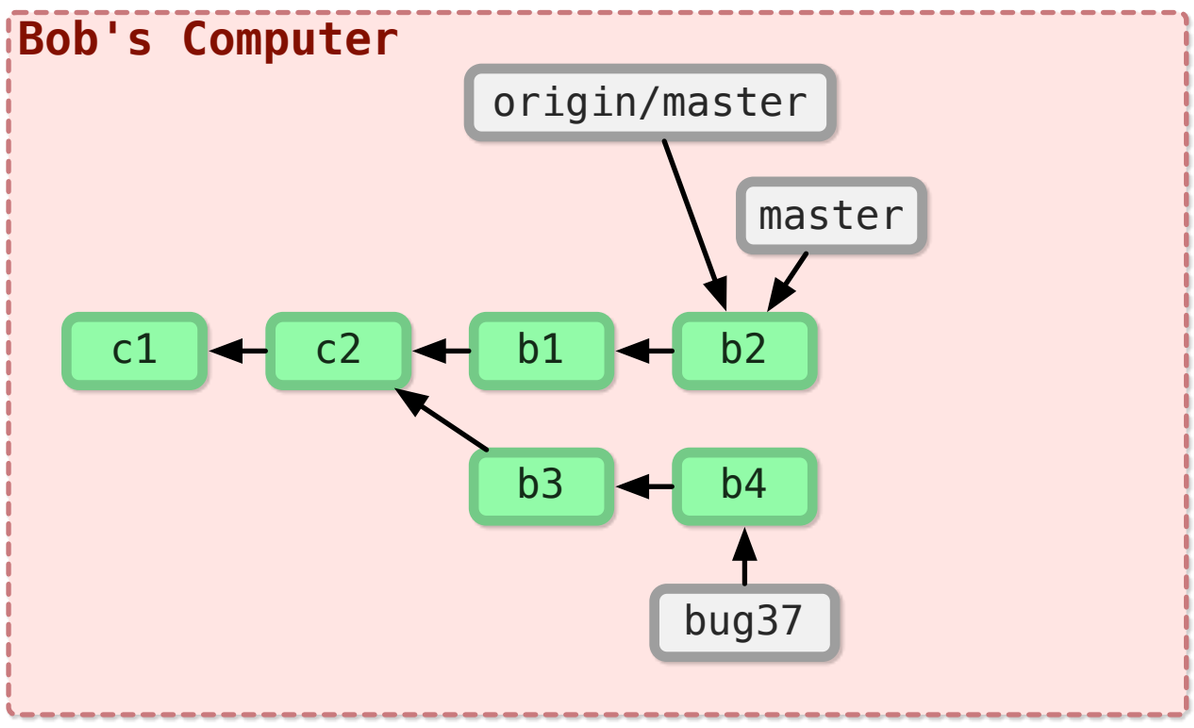




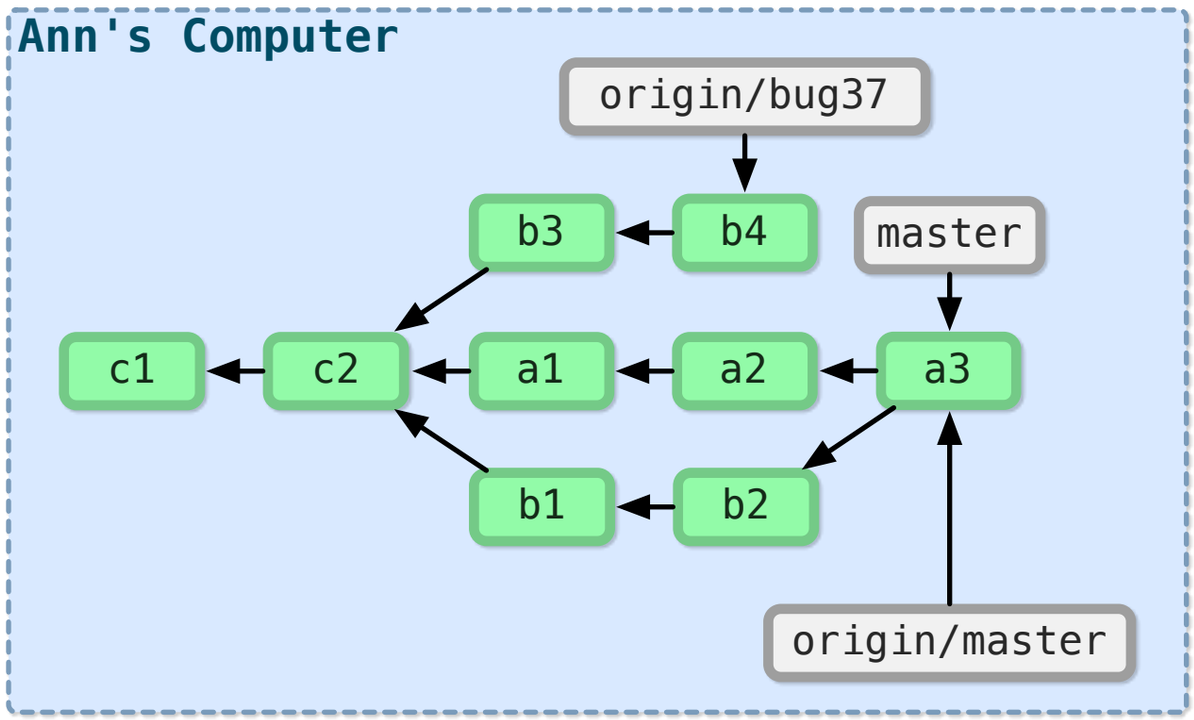

```
git checkout -b c2 bug37
git commit
git commit
```



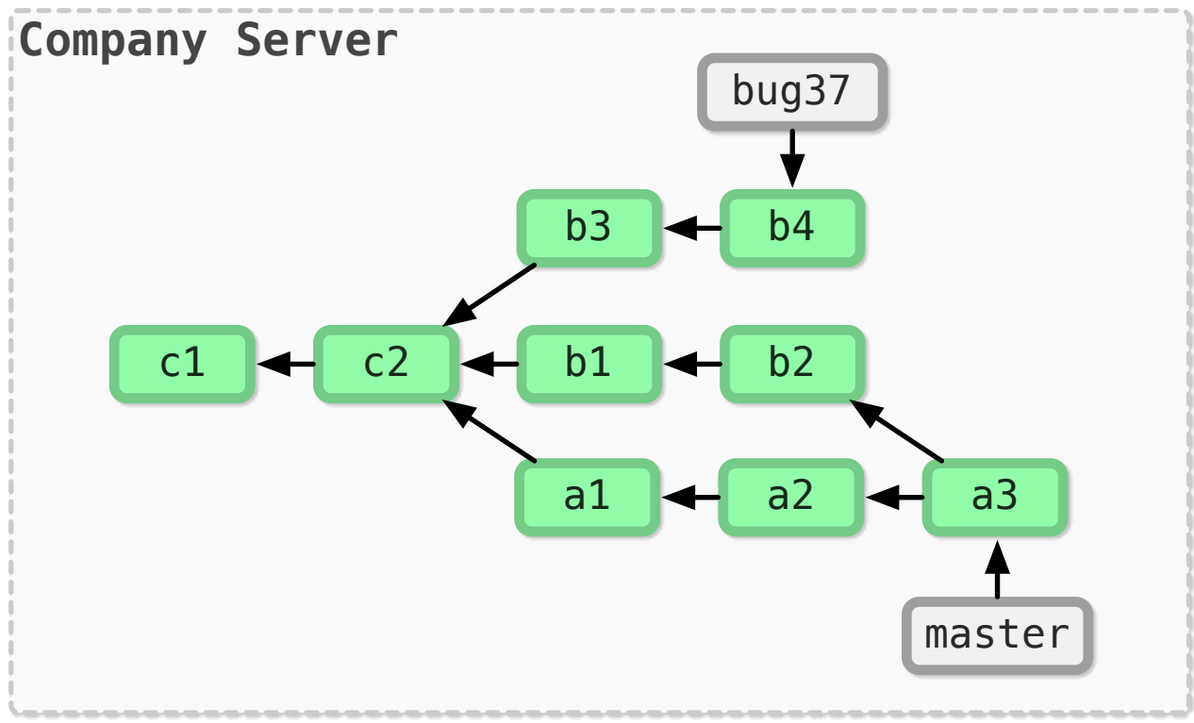


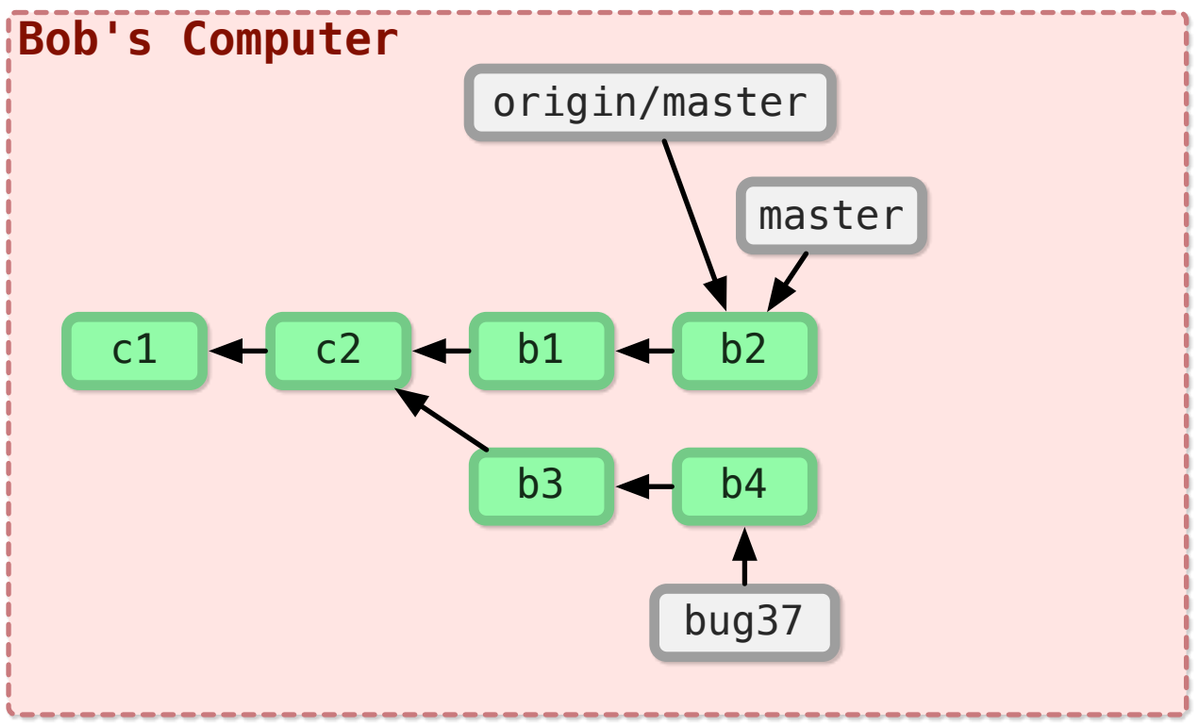



```
git push origin bug37
```


Ann's Computer
origin/bug37
b3
b4
master
c1
c2
a1
a2
a3
b1
b2
origin/master


git fetch


Company Server
bug37
b3
b4
c1
c2
b1
b2
a1
a2
a3
master


Bob's Computer
origin/master
master
c1
c2
b1
b2
b3
b4
bug37

git pull

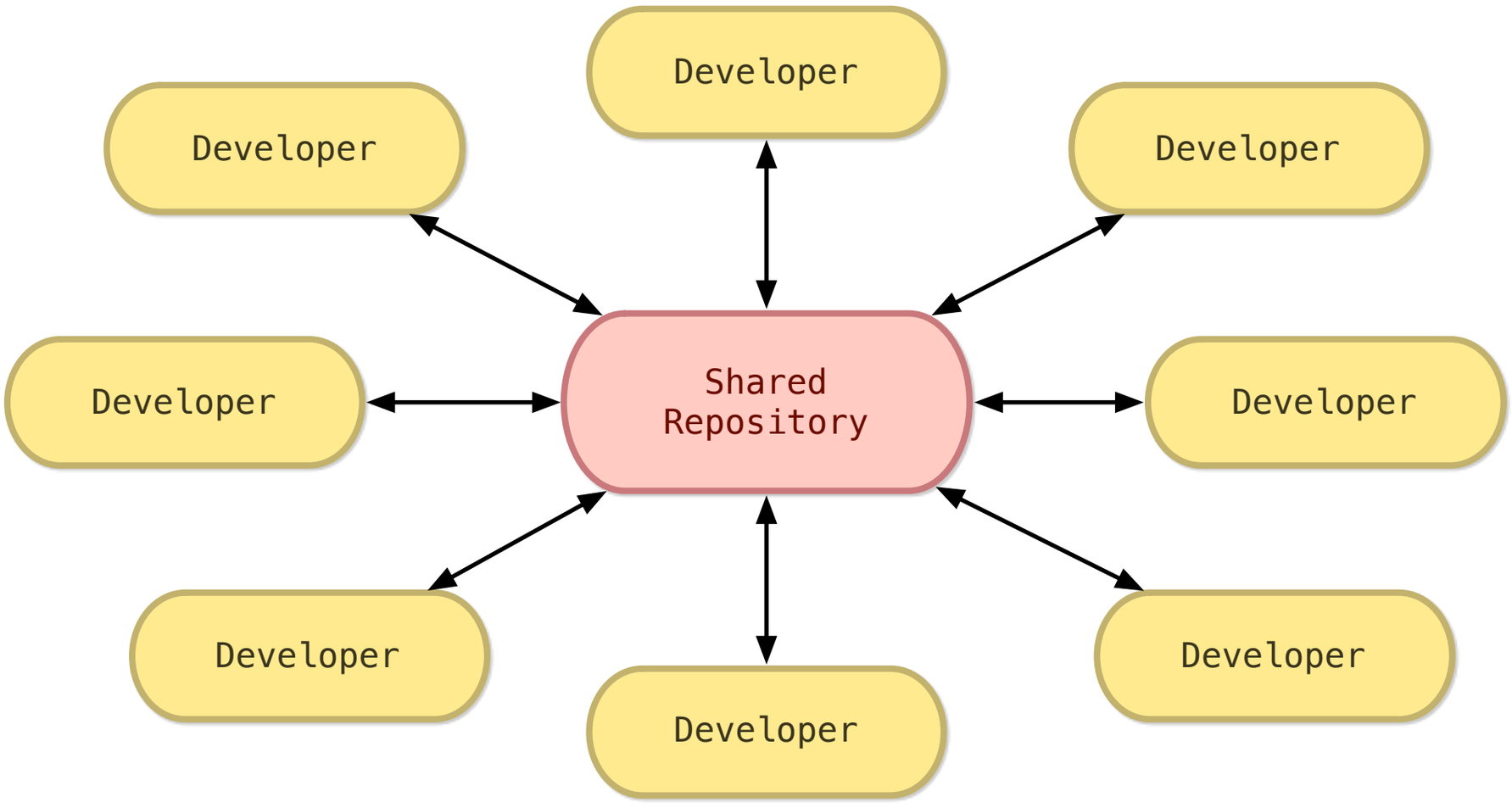
==

fetch + merge

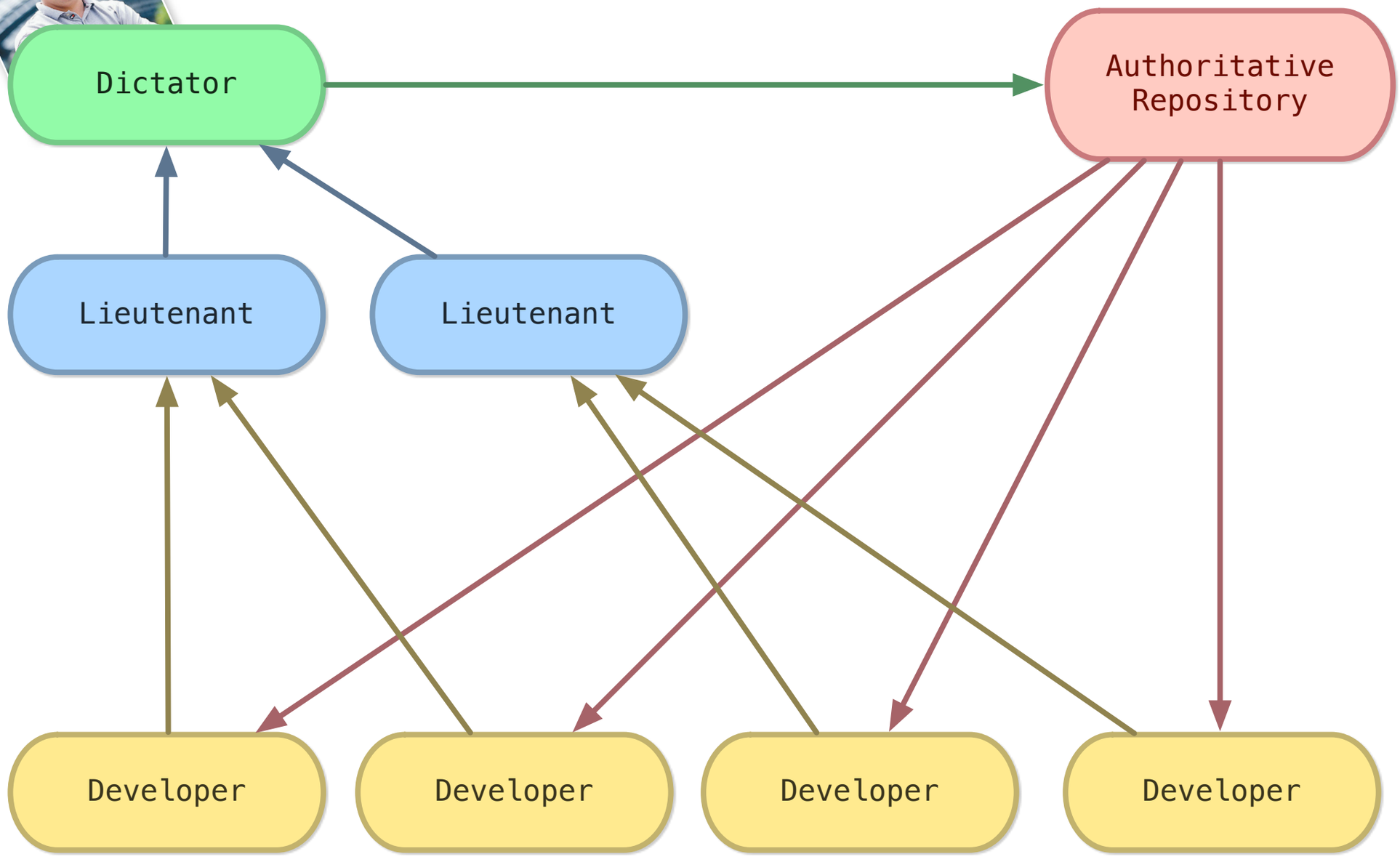
Can sometimes explode on you... safer to fetch then merge manually.

Collaboration

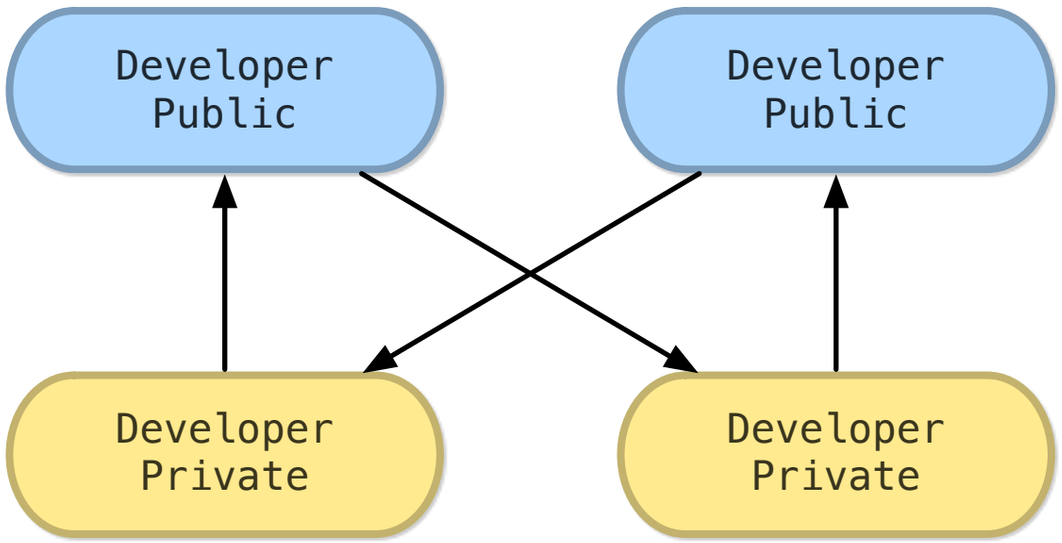
Centralized Model



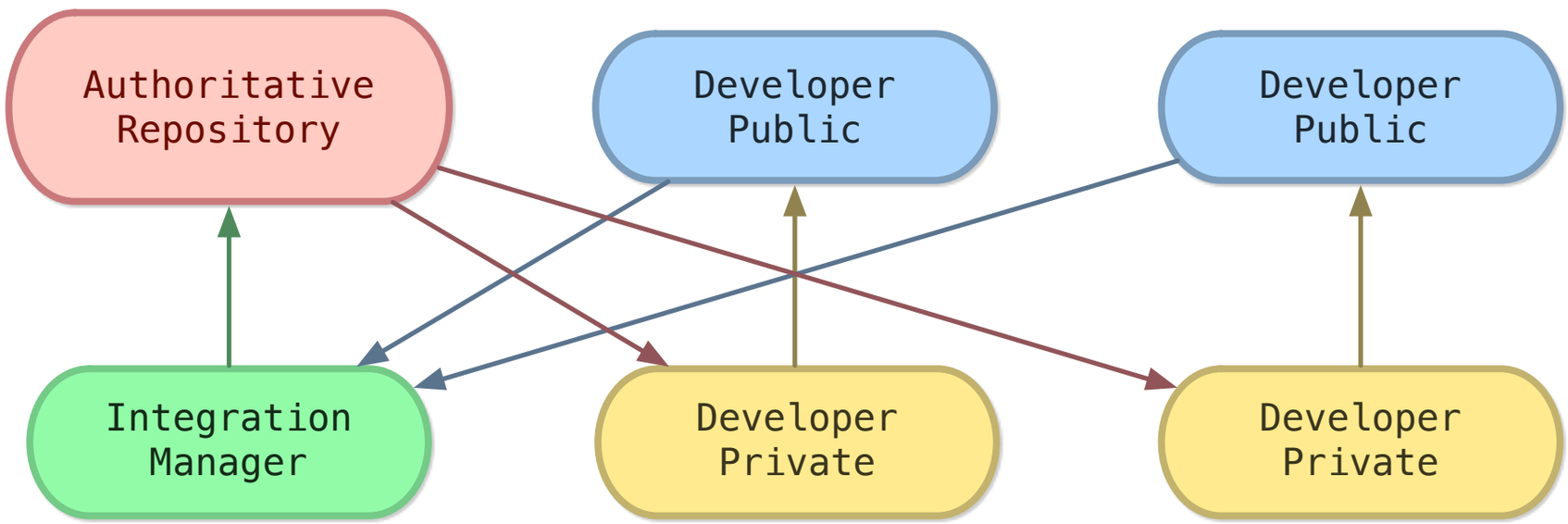
Dictator Model



Peer to Peer Model



Integration Model



Bottom Line

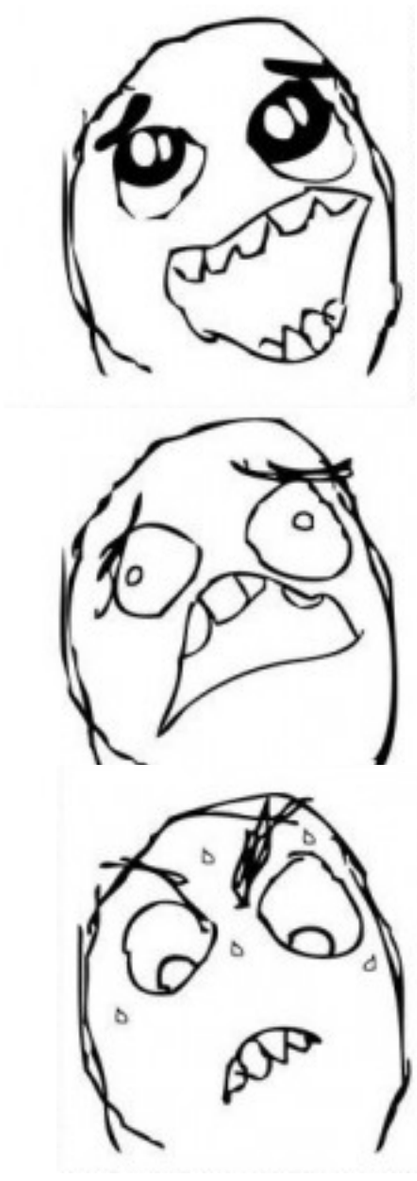
Many good collaboration models

Customize for **your** project / team

Distributed VCS gives you **options**

Workflow

Non Workflow



Make Changes

More Changes

Break Codebase

RAGE!!!

Fix Codebase

Repeat

Bad Workflow

Create Changes

Commit Changes

Basic Workflow

Create Changes

Stage Changes

Review Changes

Commit Changes

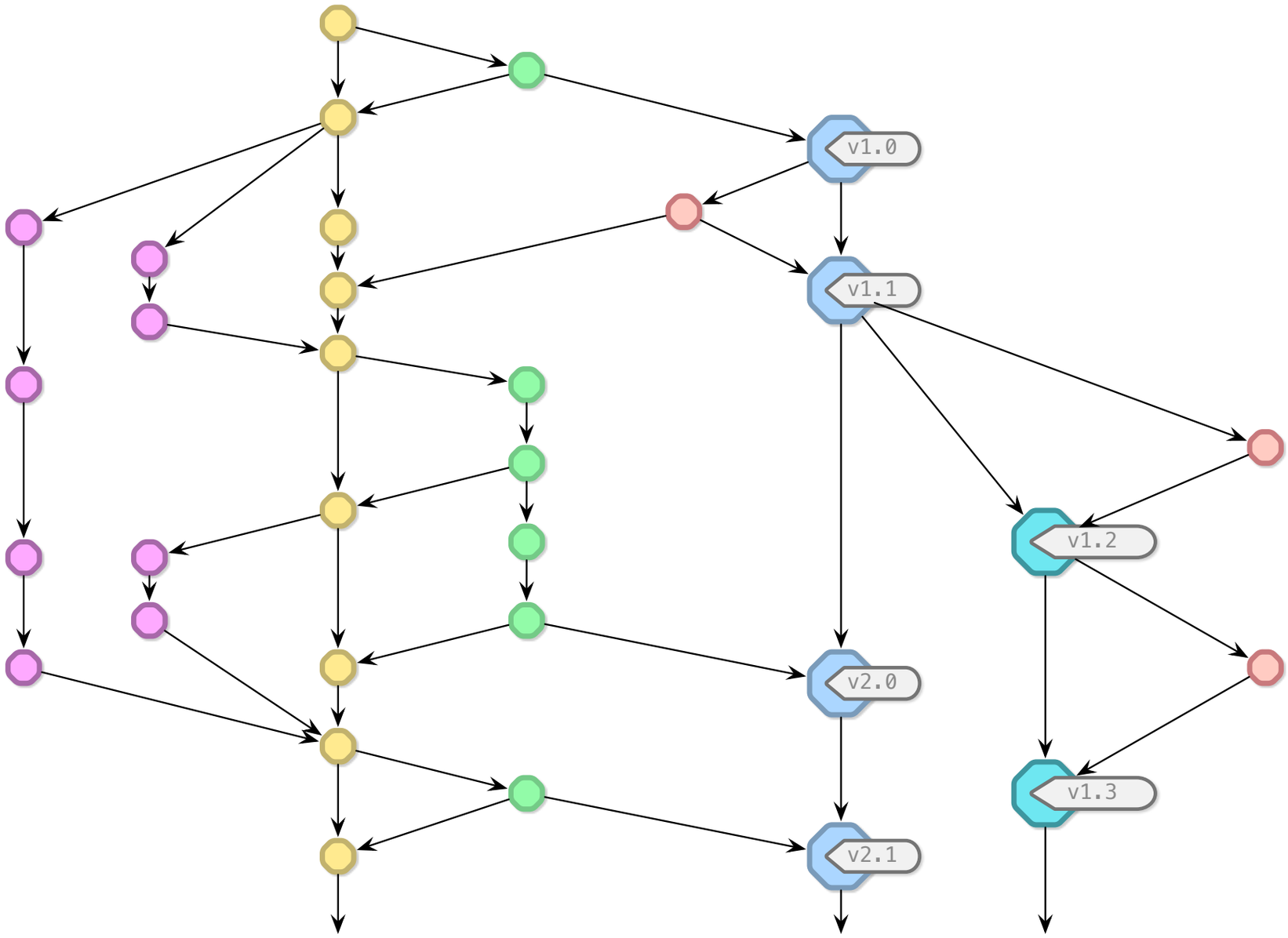
Workflow Pro Tip

Commit **Early**

Commit **Often**

Commit **Units**

Good Workflow



Credit: **Vincent Driessen**, http://nvie.com/posts/a-successful-git-branching-model

DON’T PANIC!

Primary Branches

Master Branch

Always reflects PRODUCTION-READY state.

Always exists.

Develop Branch

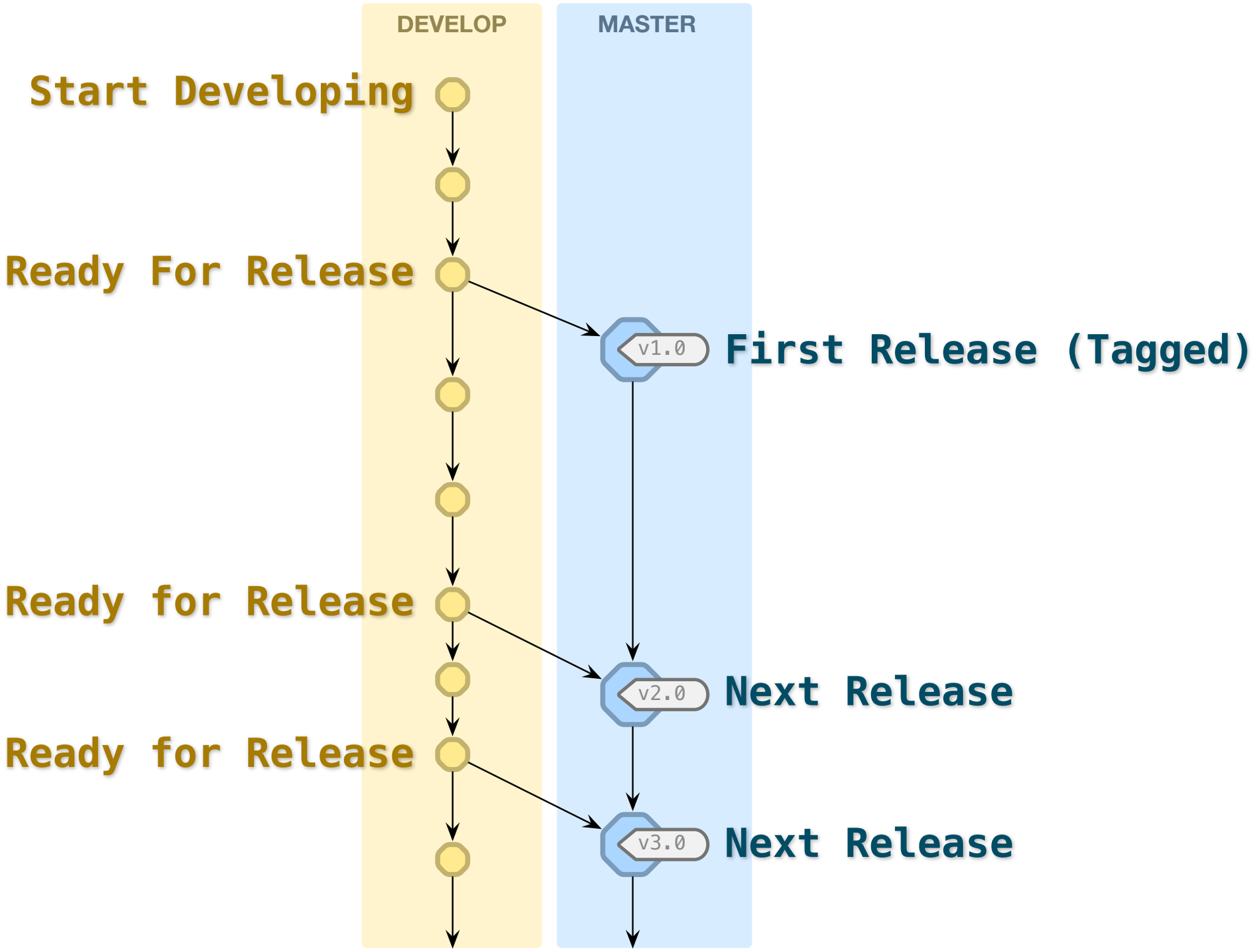
Latest DEVELOPMENT state for next release.

Base your continuous integration on this.

Ultimately ends up in MASTER.

Always exists.

Primary Branches



But DEVELOP does not really reflect a STABLE state yet.

Secondary Branches

Feature Branches

Fine-grained work-in-progress for future release.

Branches off latest DEVELOP.

Merges back into DEVELOP then discard.

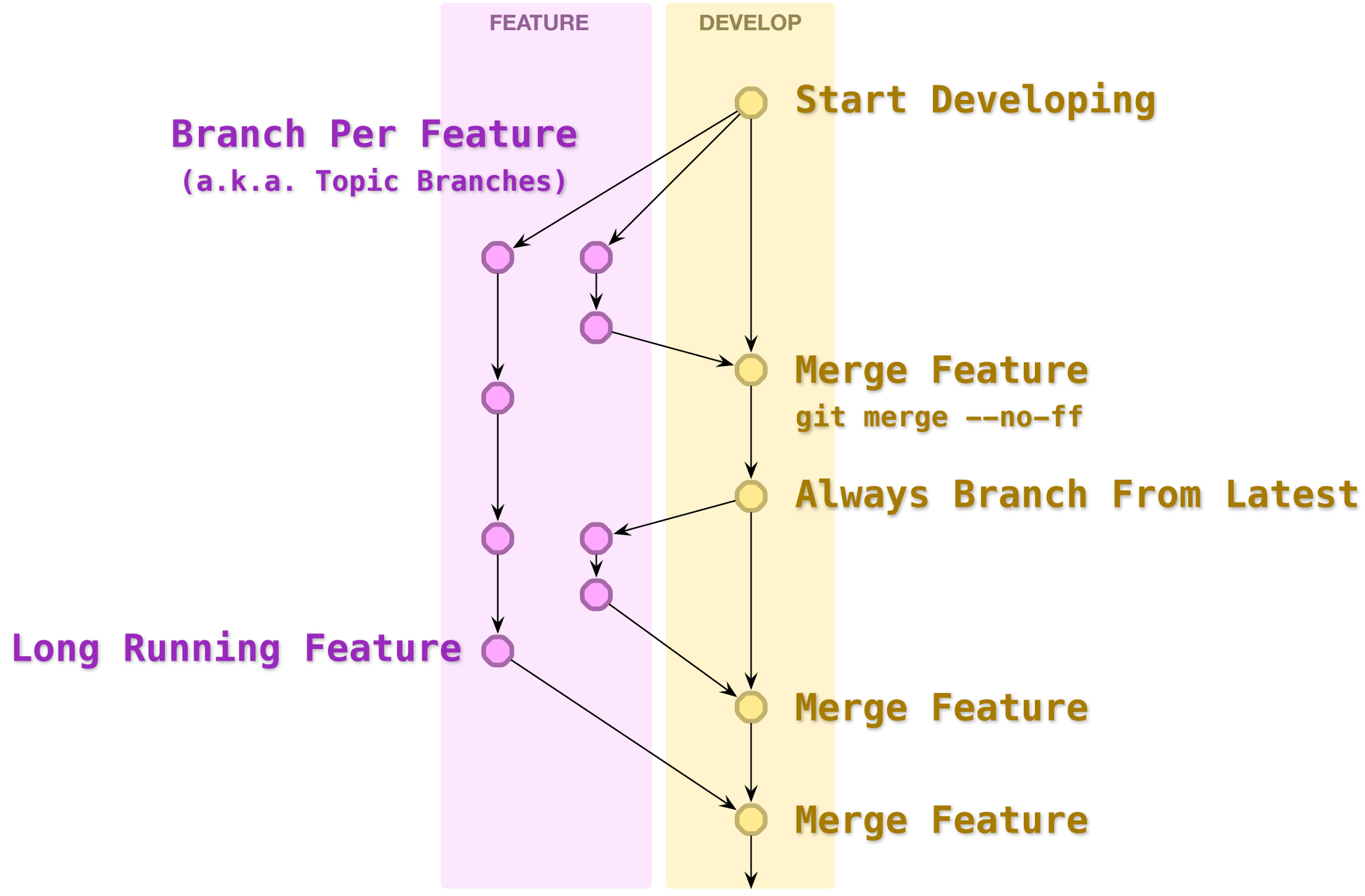
Or just discard (failed experiments).

Short or long running.

Typically in developer repositories only.

Naming convention: feature / cool-new-feature

Feature Branches



Secondary Branches

Release Branches

Latest RELEASE CANDIDATE state.

Preparatory work for release.

Last minute QA, testing & bug fixes happens here.

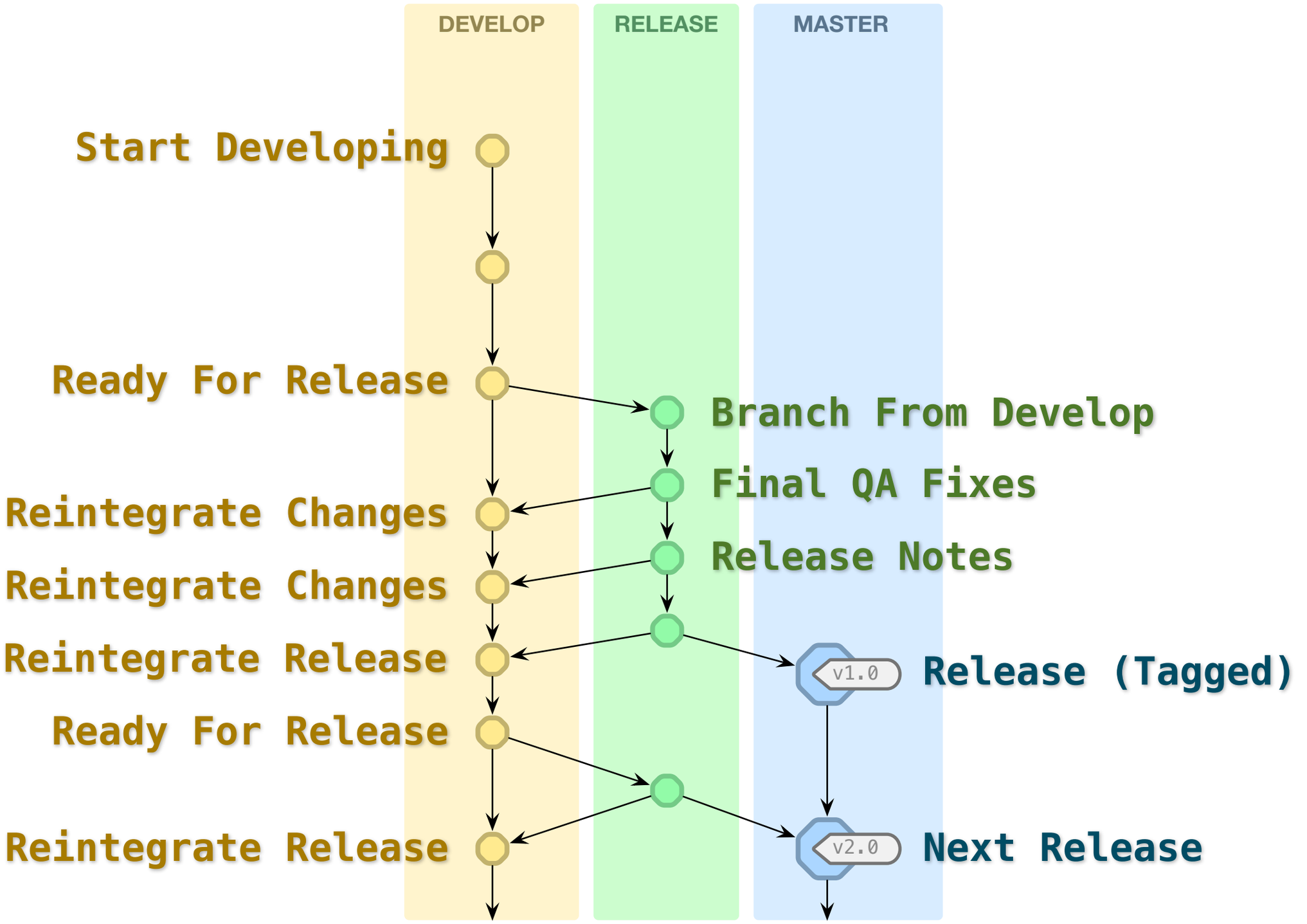
Sits between DEVELOP and MASTER.

Branch from DEVELOP.

Merge back into both MASTER and DEVELOP.

Discard after merging.

Release Branches



Secondary Branches

HotFix Branches

Like **RELEASE**, preparing for new release.

Resolve **emergency problems** with **existing** production release.

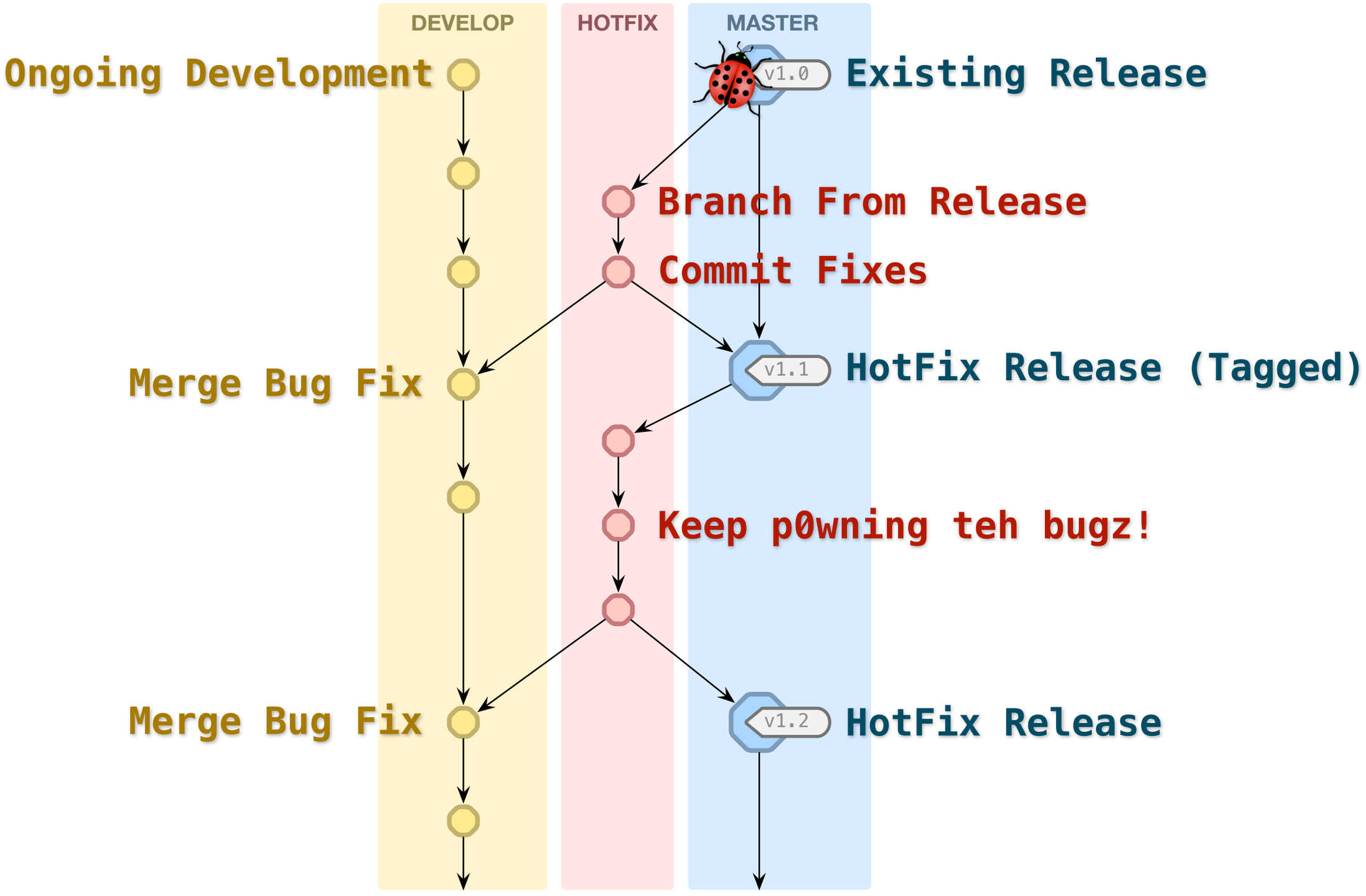
Branch from **MASTER**.

Merge back into both **MASTER** and **DEVELOP**.

Discard after merging.

Naming convention: **hotfix / bug-157**

HotFix Branches



Secondary Branches

Support Branches

Similar to **MASTER** + **HOTFIX** for **legacy releases.**

Branches off from **earlier** tagged **MASTER**.

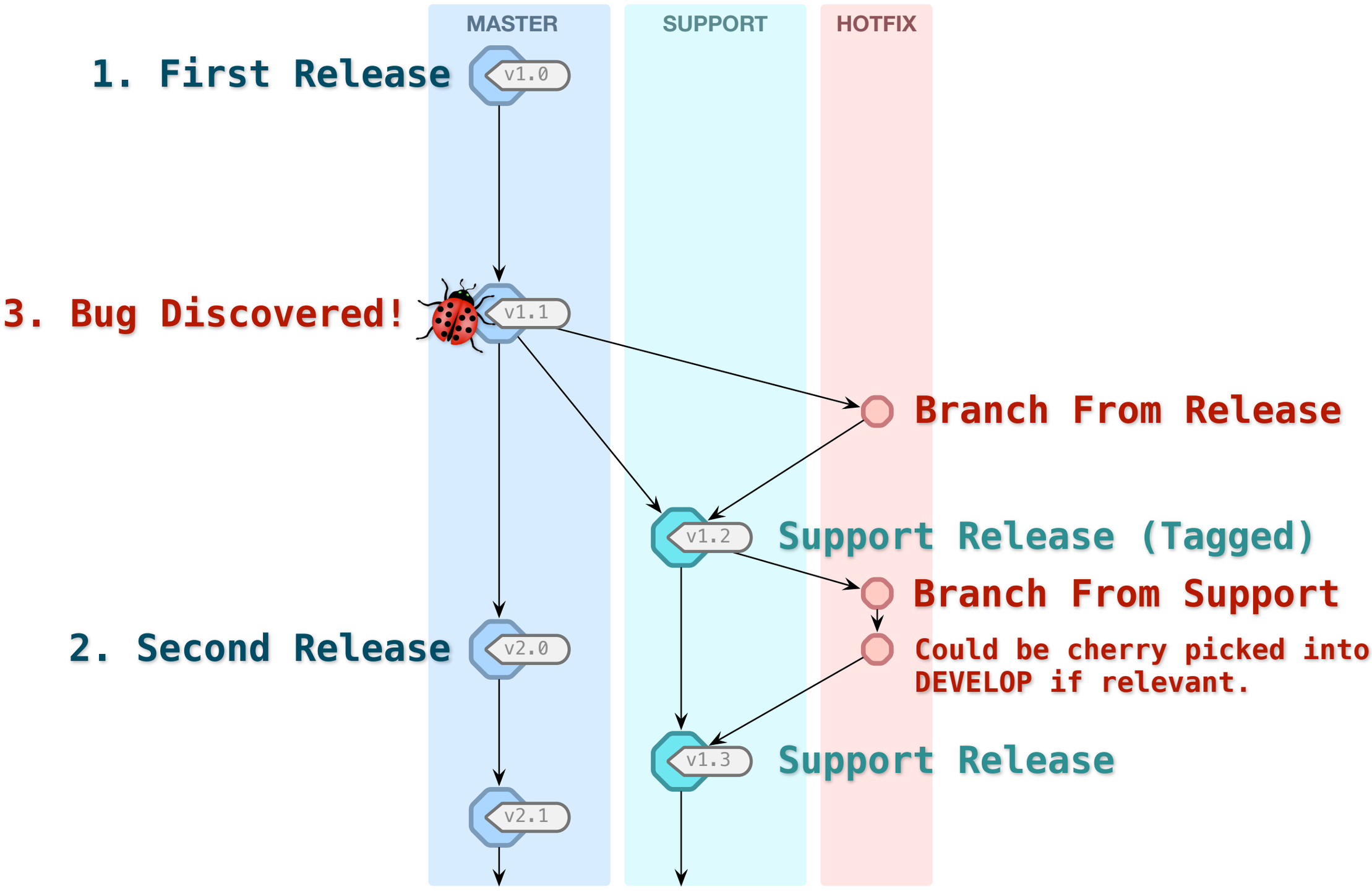
Does not merge back into anything.

Always exists once created.

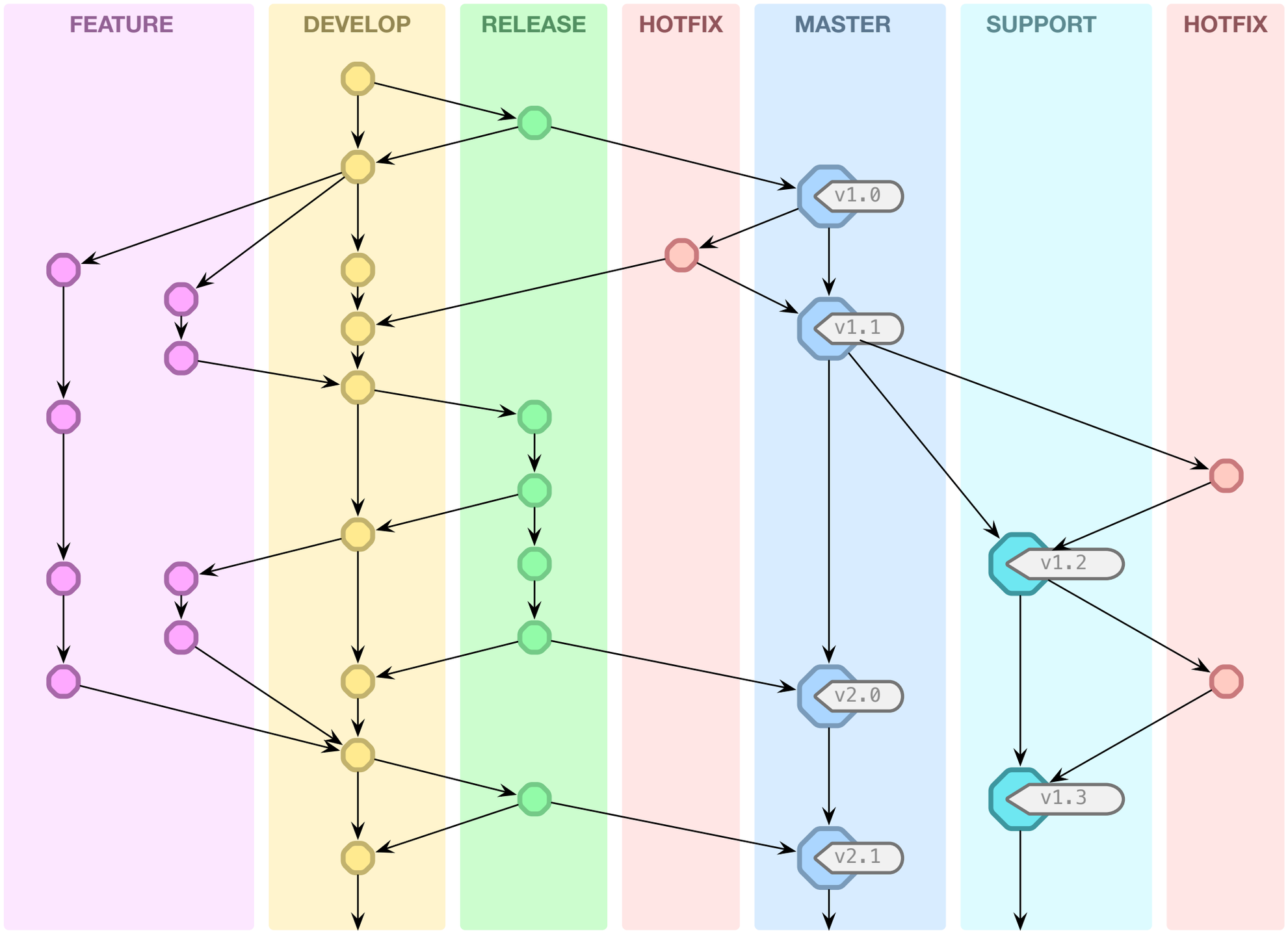
Continuing **parallel master branch** for a version series.

Naming convention: **support / version-1**

Support Branches



All Together Now



Public-Private Workflow

Public Branches

Authoritative history of the project.

Commits are **succinct** and well-documented.

As **linear** as possible.

Immutable.

Credit: **Benjamin Sandofsky**, http://sandofsky.com/blog/git-workflow.html

Public-Private Workflow

Private Branches

Disposable and **malleable.**

Kept in **local** repositories.

Never merge directly into **public**.

First **clean up** (reset, rebase, squash, and amend)

Then merge a **pristine, single commit** into **public**.

Credit: **Benjamin Sandofsky**, http://sandofsky.com/blog/git-workflow.html

Public-Private Workflow

1. Create a **private** branch off a **public** branch.
2. Regularly commit your work to this **private** branch.
3. Once your code is perfect, clean up its history.
4. Merge the cleaned-up branch back into the **public** branch.

Credit: **Benjamin Sandofsky**, http://sandofsky.com/blog/git-workflow.html

Bottom Line

Every project is different.

Custom design your workflow.

Branches are your LEGO blocks.

Further Reading

http://nvie.com/posts/a-successful-git-branching-model

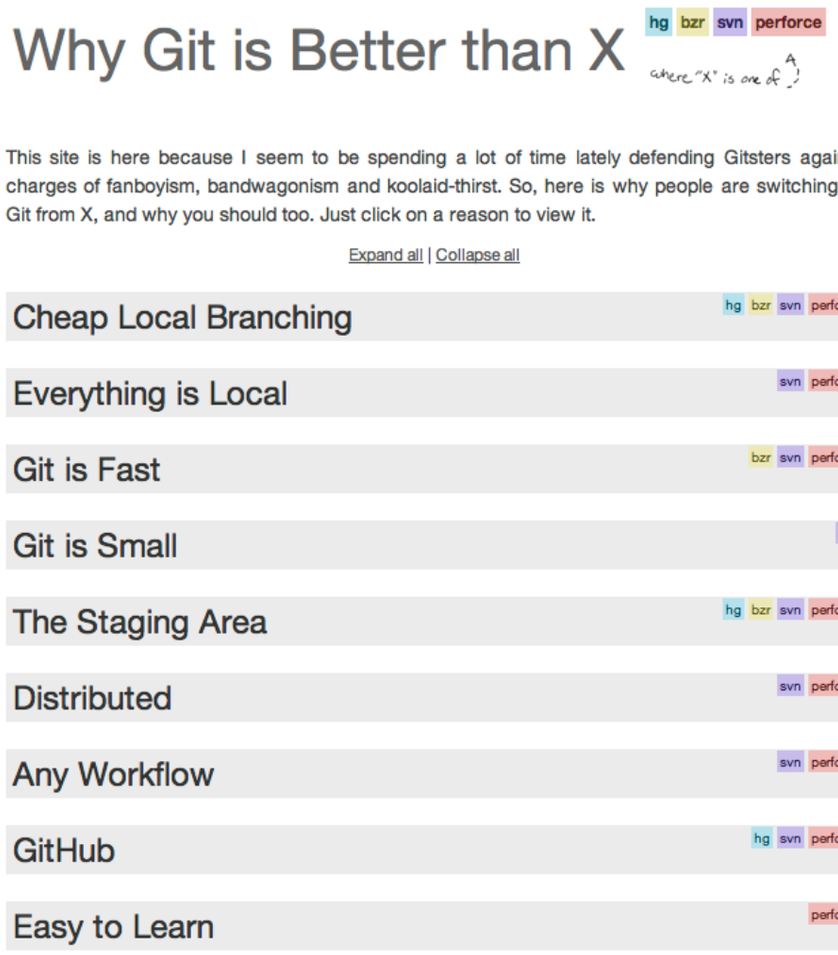
https://github.com/nvie/gitflow

http://sandofsky.com/blog/git-workflow.html

Resources & Tools

Stuff I recommend.

Choosing Git



http://whygitisbetterthanx.com

by Scott Chacon

Getting Git

Home | About Git | Documentation | Download | Tools & Hosting | Wiki

Git is...

Git is a **free & open source, distributed version control system** designed to handle everything from small to very large projects with speed and efficiency.

Every Git clone is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. **Branching and merging are fast** and easy to do.

Git is used for version control of files, much like tools such as Mercurial, Bazaar, Subversion, CVS, Perforce, and Team Foundation Server.

Projects using Git

- Git
- Linux Kernel
- Perl
- Eclipse
- Gnome
- KDE
- Qt
- Ruby on Rails
- Android
- PostgreSQL
- Debian
- X.org

Download Git

The latest stable Git release is

v1.7.6

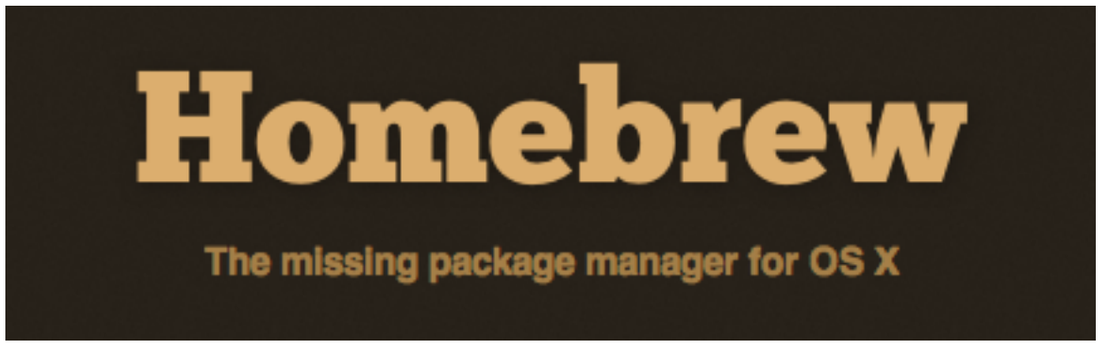
release notes (2011-06-26)

Windows | Mac OSX | Source

Other Download Options
Git Source Repository

http://git-scm.com

Getting Git



Install Homebrew

http://mxcl.github.com/homebrew/

```
$ brew install git
```

Not really Git related, but a great move overall and a fantastic way to keep Git up to date. Actually uses Git itself to keep itself up to date.

Commanding Git

Edit ~/.profile

```
git config --global user.name "Patrick Hogan"
git config --global user.email pbhogan@gmail.com

git config --global core.autocrlf input
git config --global core.safecrlf true

git config --global color.ui true
```

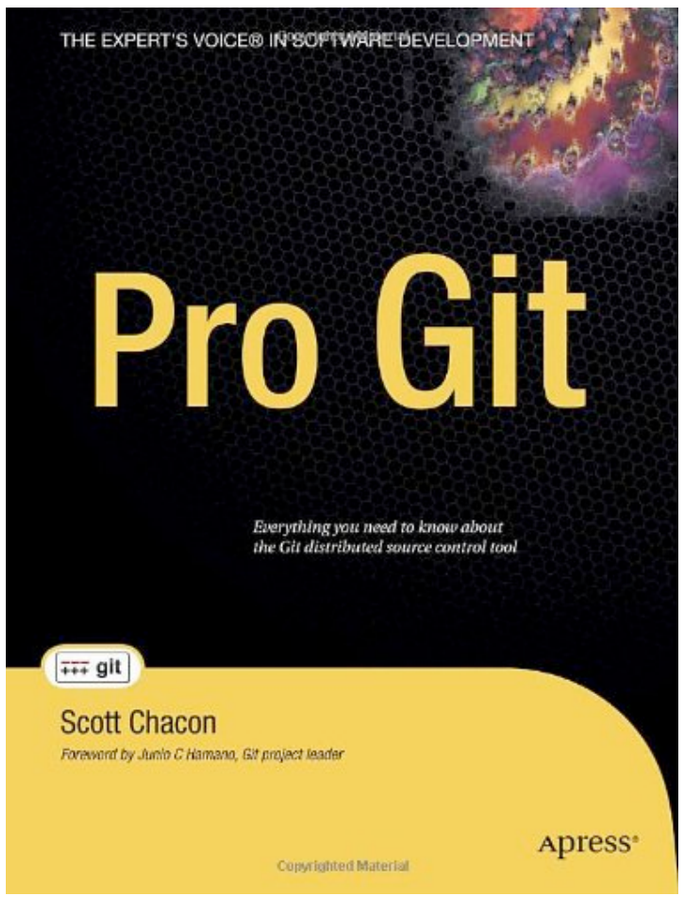
Commanding Git

Edit ~/.profile

```
source /usr/local/etc/bash_completion.d/git-completion.bash

RED="\[\033[0;31m\]"
YELLOW="\[\033[0;33m\]"
GREEN="\[\033[0;32m\]"
WHITE="\[\033[1;37m\]"
RESET="\[\033[1;0m\]"
GIT='$(__git_ps1 "[%s]")'
PS1="\n$WHITE\u$RESET: \W$YELLOW$GIT $GREEN\$$RESET "
```

```
pbhogan: Swivel[master] $
```

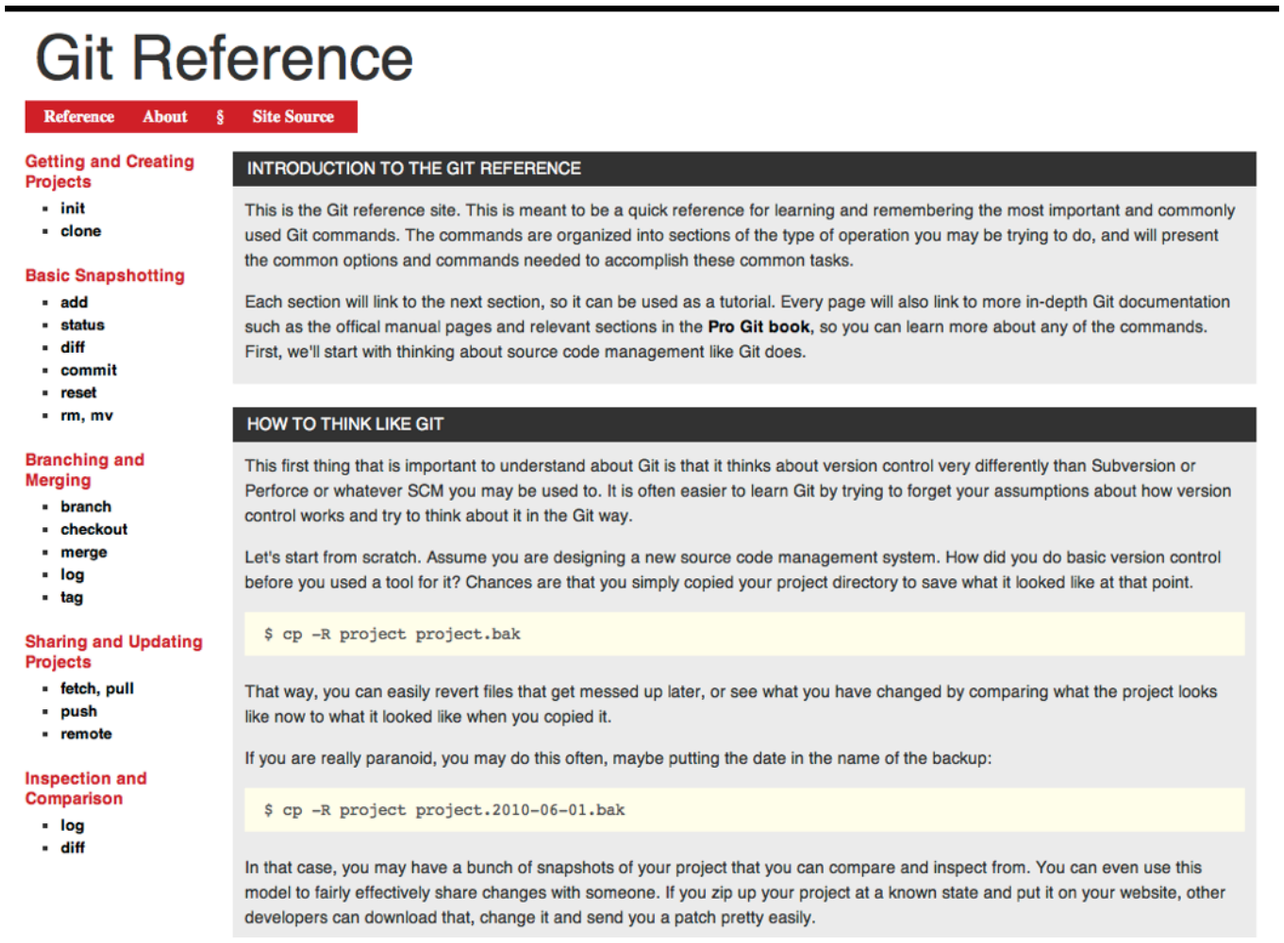
Learning Git



http://progit.org/book/
by Scott Chacon

It's free, and available in e-book form (but buy it to support this Scott!)

Learning Git



Git Reference

Reference About § Site Source

Getting and Creating Projects
- init
- clone

Basic Snapshotting
- add
- status
- diff
- commit
- reset
- rm, mv

Branching and Merging
- branch
- checkout
- merge
- log
- tag

Sharing and Updating Projects
- fetch, pull
- push
- remote

Inspection and Comparison
- log
- diff

INTRODUCTION TO THE GIT REFERENCE

This is the Git reference site. This is meant to be a quick reference for learning and remembering the most important and commonly used Git commands. The commands are organized into sections of the type of operation you may be trying to do, and will present the common options and commands needed to accomplish these common tasks.

Each section will link to the next section, so it can be used as a tutorial. Every page will also link to more in-depth Git documentation such as the offical manual pages and relevant sections in the **Pro Git book**, so you can learn more about any of the commands. First, we'll start with thinking about source code management like Git does.

HOW TO THINK LIKE GIT

This first thing that is important to understand about Git is that it thinks about version control very differently than Subversion or Perforce or whatever SCM you may be used to. It is often easier to learn Git by trying to forget your assumptions about how version control works and try to think about it in the Git way.

Let's start from scratch. Assume you are designing a new source code management system. How did you do basic version control before you used a tool for it? Chances are that you simply copied your project directory to save what it looked like at that point.

```
$ cp -R project project.bak
```

That way, you can easily revert files that get messed up later, or see what you have changed by comparing what the project looks like now to what it looked like when you copied it.

If you are really paranoid, you may do this often, maybe putting the date in the name of the backup:

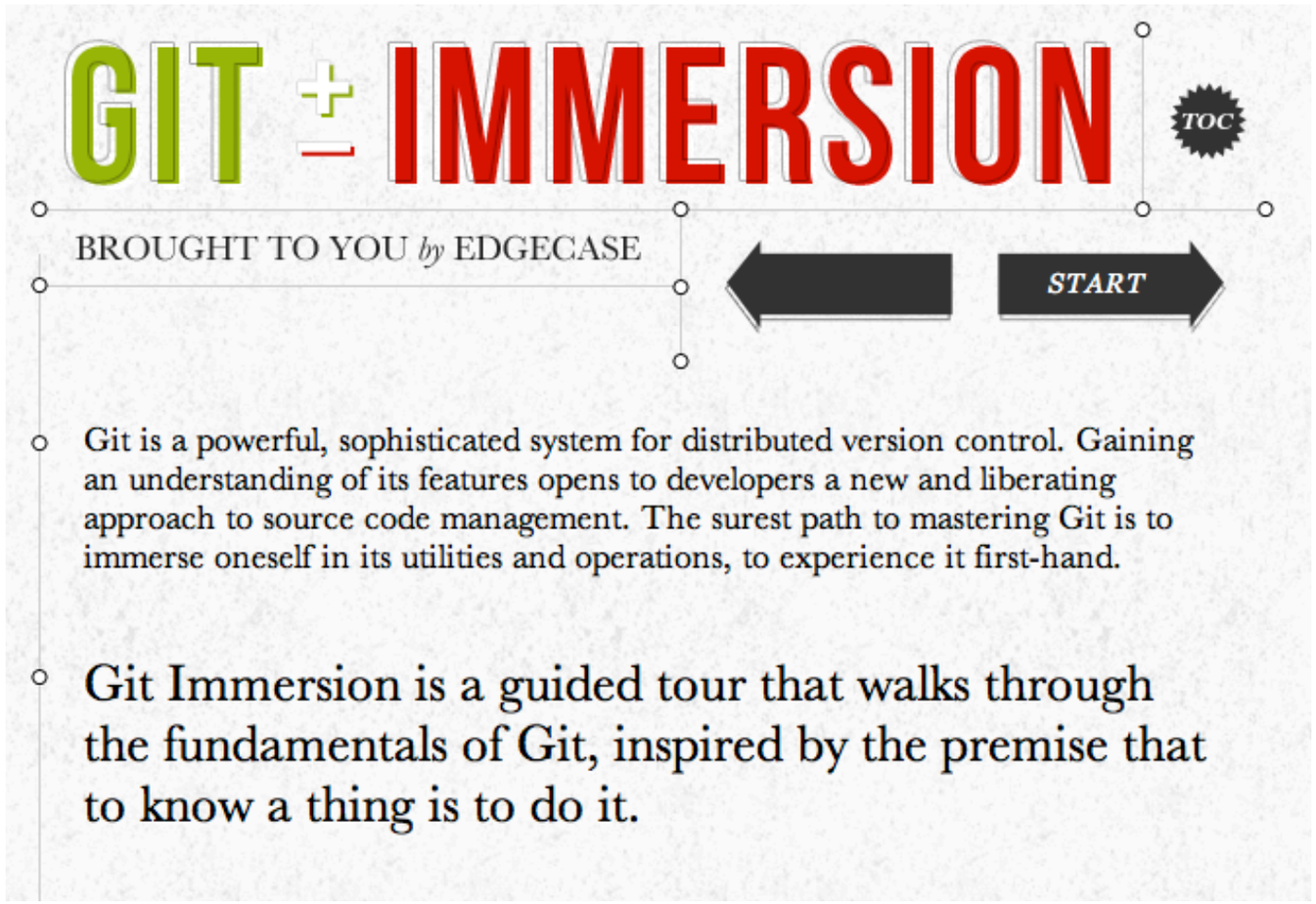
```
$ cp -R project project.2010-06-01.bak
```

In that case, you may have a bunch of snapshots of your project that you can compare and inspect from. You can even use this model to fairly effectively share changes with someone. If you zip up your project at a known state and put it on your website, other developers can download that, change it and send you a patch pretty easily.

http://gitref.org

by the GitHub team

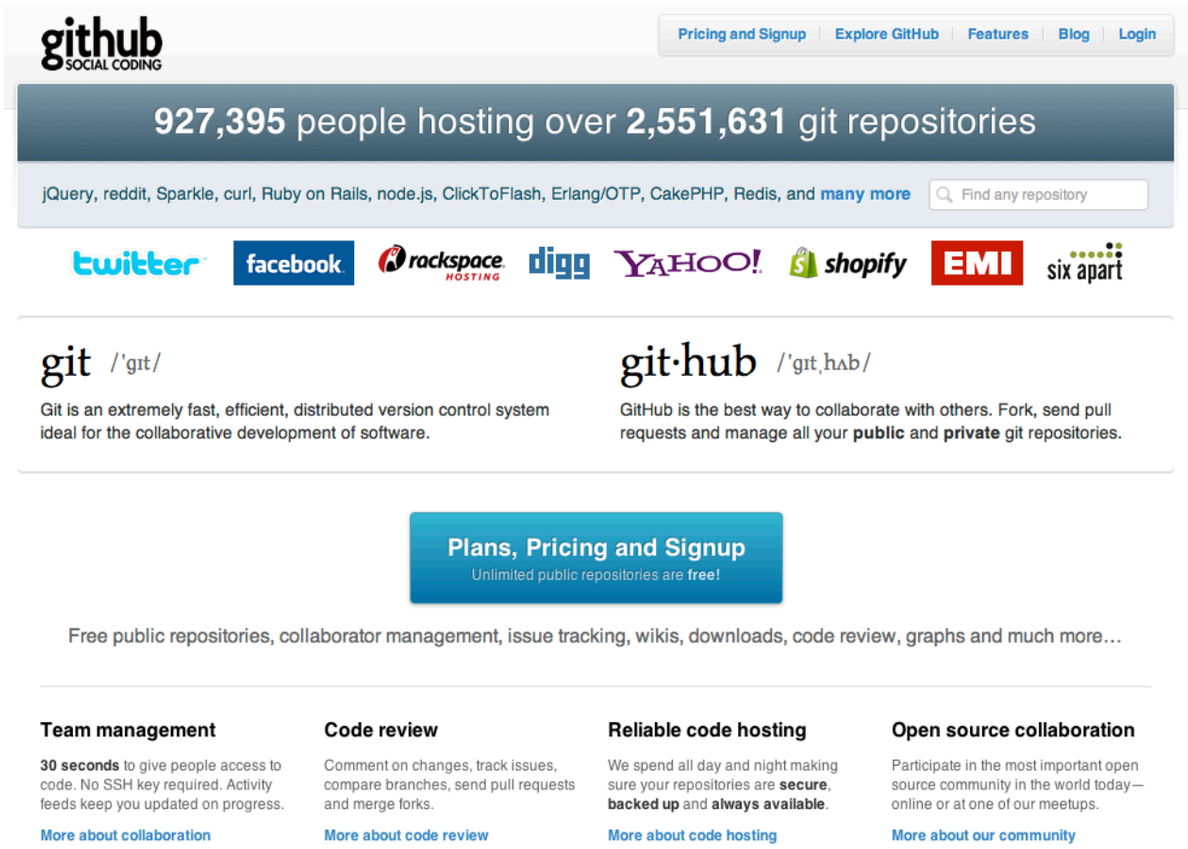
Learning Git



http://gitimmersion.com
by the EdgeCase team

Learn by doing approach.

Hosting Git



http://github.com

Free public repositories.

Free for public repositories — it's where I host my open source stuff.
$7 for 5 private repositories.

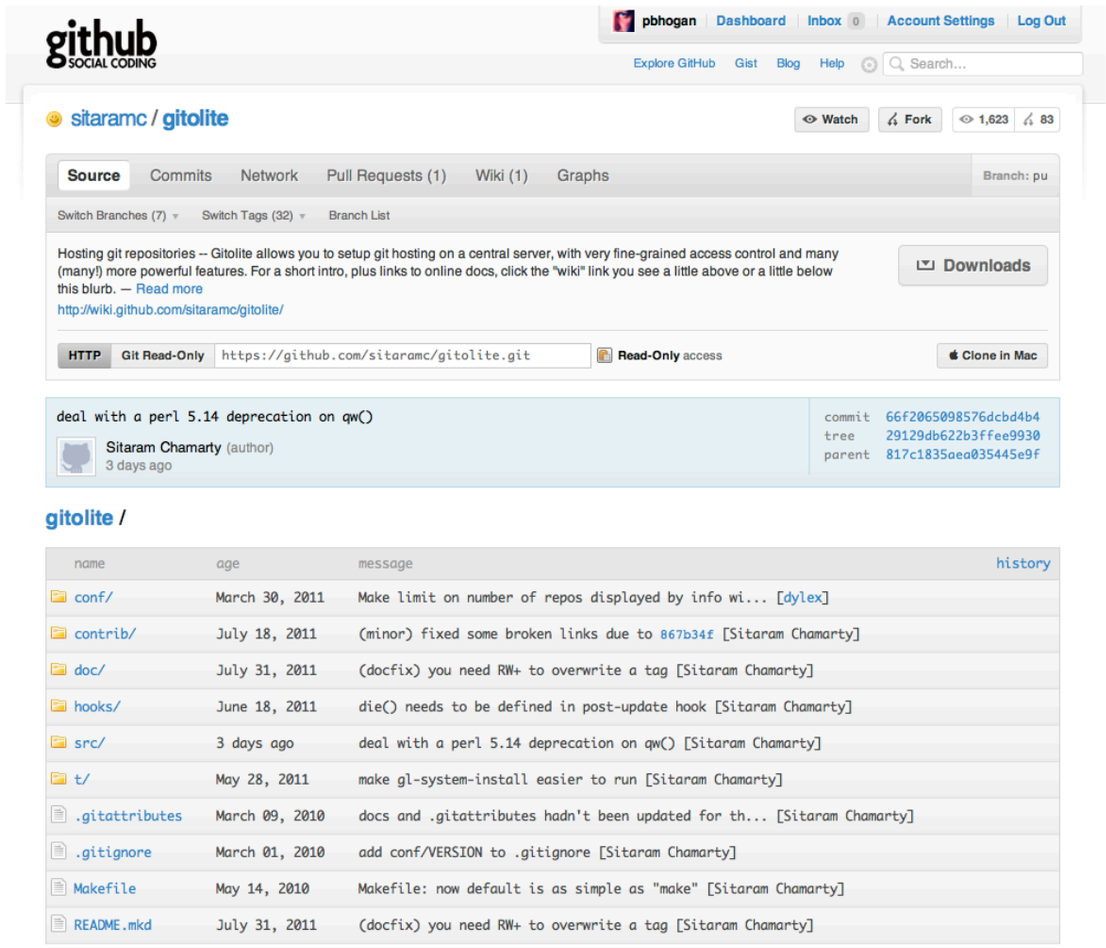
GitHub Promo Code

360iDEVph

http://github.com
Free public repositories.

Free for public repositories — it's where I host my open source stuff.
$7 for 5 private repositories.

Hosting Git



https://github.com/sitaramc/gitolite

DIY team repositories hosting.

Gitolite allows you to setup git hosting on a central server, with very fine-grained access control.

Hosting Git

http://dropbox.com
DIY single user hosting :-)

Yes! This is how I host my private projects currently.

Hosting Git


```
$ mkdir -p /Users/pbhogan/Dropbox/Repos/Swivel.git
$ cd /Users/pbhogan/Dropbox/Repos/Swivel.git
$ git init --bare
$ cd /Users/pbhogan/Projects/Swivel
$ git remote add dropbox file:///Users/pbhogan/Dropbox/Repos/Swivel.git
```

http://dropbox.com
DIY single user hosting :-)

Here's how. Basically just setting up a file:// remote to a location in your Dropbox. Dropbox takes care of the rest.
SINGLE USER ONLY!!! Bad things will happen if you try this in a shared folder.

Using Git

http://www.git-tower.com

The single best Git client — bar none.

This client is awesome and super polished. I use it every day at work.
Has GitHub and Beanstalk integration.

Tower Promo Code

IDEV2011

http://www.git-tower.com
The single best Git client — bar none.

This client is awesome and super polished. I use it every day at work.
Has GitHub and Beanstalk integration.

Using Git

http://mac.github.com

Free. Works with GitHub only.

Nice looking client if you're exclusively a GitHub user. I prefer Git Tower myself.

Diffing Git

http://kaleidoscopeapp.com

Compare files. Integrates with Tower.

A very sexy app for file comparison. Even compares images. Integrates with lots of other tools including Git Tower and command line.

Merging Git

http://sourcegear.com/diffmerge

Decent, free file merging tool.

A bit clunky and sluggish, but quite effective tool for helping you merge files. I hope Kaleidoscope will add merge conflict resolving one day so I can stop using this. Free.

Tower Promo: IDEV2011
10% discount

GitHub Promo: 360iDEVph
1 month free micro account

Patrick Hogan

@pbhogan

patrick@gallantgames.com

www.gallantgames.com

Please rate this talk at: http://spkr8.com/t/7606