Principles of Software Construction: Objects, Design, and Concurrency

Git Workflows in Practice

Jonathan Aldrich **Bogdan Vasilescu**

Reopened: Lecture 19 (Modules) Quiz, on Canvas

Lecture 20 (Test Doubles) Quiz, on Canvas

Recall: Types of Test Doubles

Fakes: Fully functional class with simplified implementation

Stubs: Artificial class that returns pre-configured data

Mocks: Instrumented variant of real class with fine-grained control

- Tend to be used interchangeably in practice
  - Most frameworks/libraries that support this focus on *mocking* (e.g., Mockito, ts-mocks), but also enable stubbing.
  - Rule of thumb: with stubs, you just assert against **values returned**, while with mocks, you assert against the **actual (instrumented) object**

Administrative

- Midterm scores released
- HW6a: Framework design and submit design documents by Friday, Apr 7

Midterm Reflections

- Midterm 1 median (85%) / mean (83%) vs Midterm 2 median (81%) / mean (79%)
- Lowest scores:
  - Callbacks – we talked about it on Tuesday
  - Decorator pattern – more about it next Tuesday
  - Good to practice implementing a few common design patterns before the final
- General note: work on explanations!
  - Answers often echoed lines from slides (even verbatim), but didn't *explain* why they applied. Using the right keywords typically just gets you half the credit.
- Other thoughts/questions?

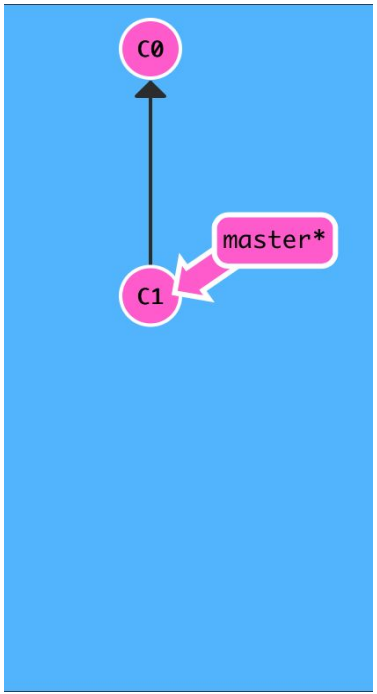
Today

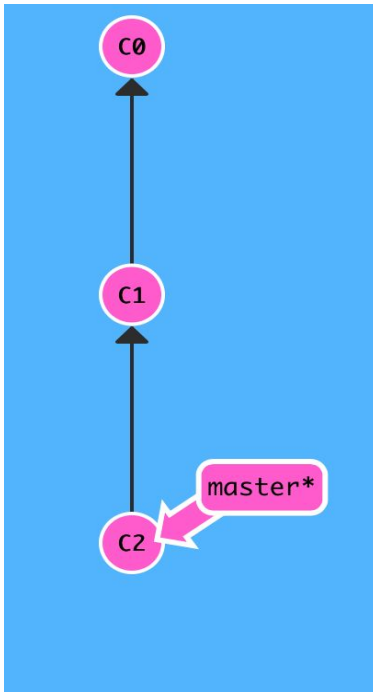
- Revisiting Git
  - Deeper dive into its internals
  - Branches & forks – modern developer workflows
- Software development at scale
  - How do Google/Meta do it?
  - Processes & tools
  - Discussion on Mono-repos

GIT BASICS

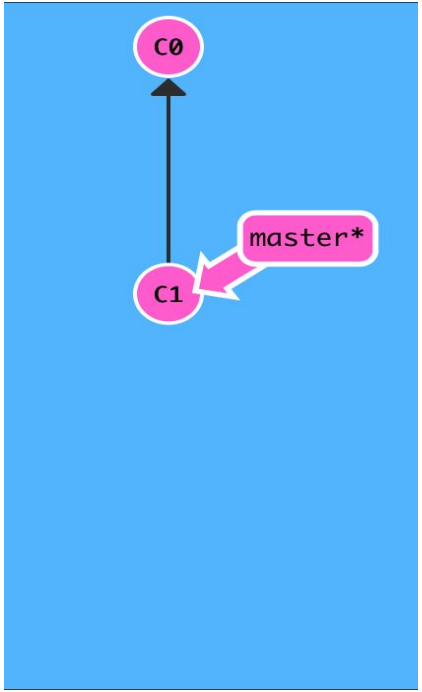
Graphics by https://learngitbranching.js.org
Note on outdated terminology:
https://www.theserverside.com/feature/Why-GitHub-renamed-its-master-branch-to-main

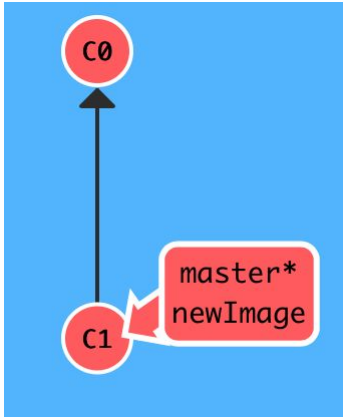
git commit



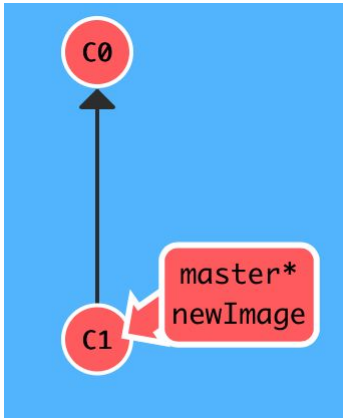


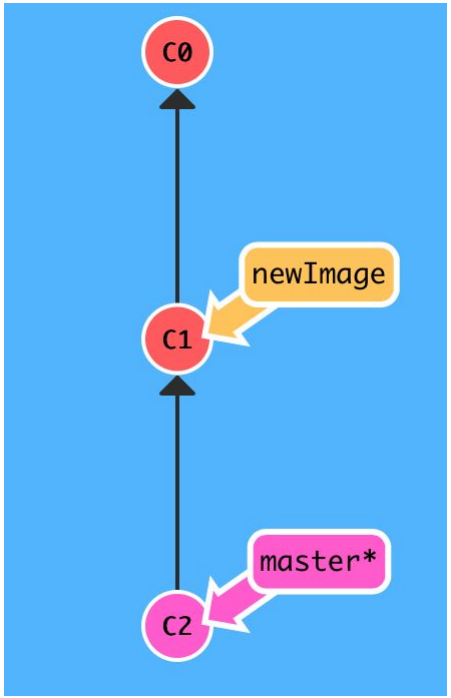
git branch newImage



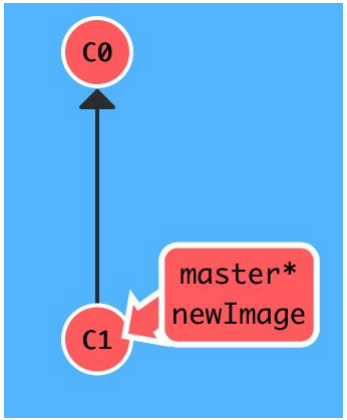


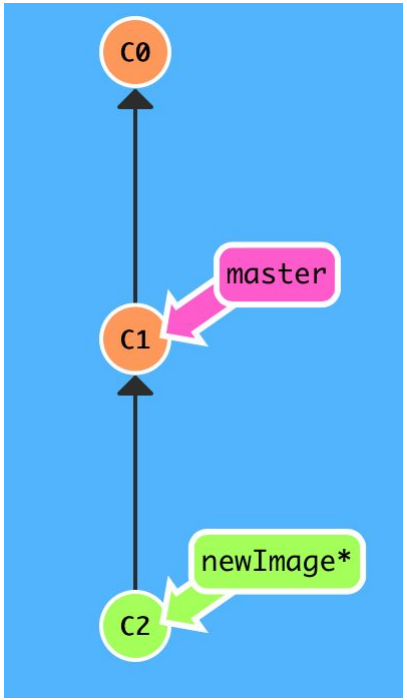
git commit





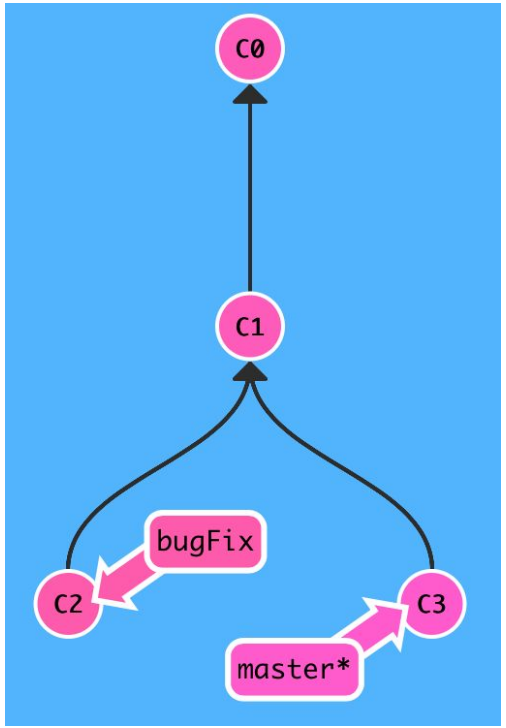
git checkout newImage; git commit

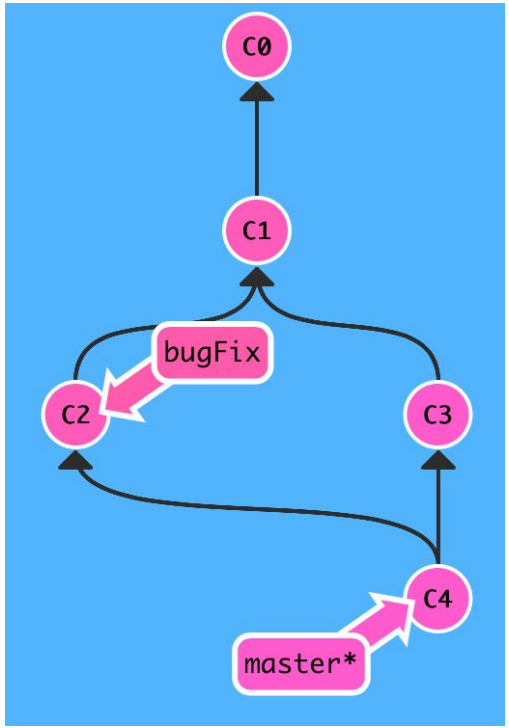




Three ways to move work around between branches

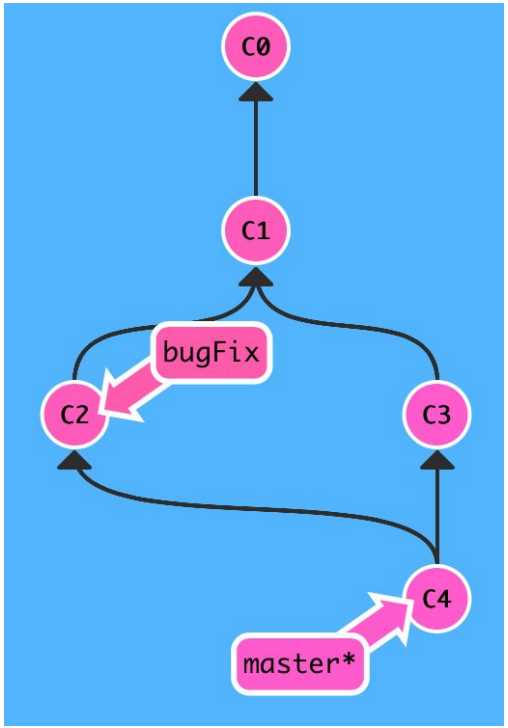
1) `git merge bugFix` (into `master`)

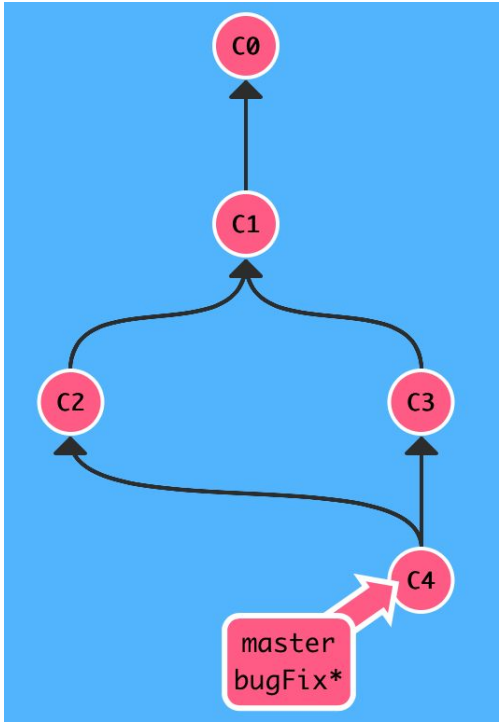




You can also merge master into bugFix:

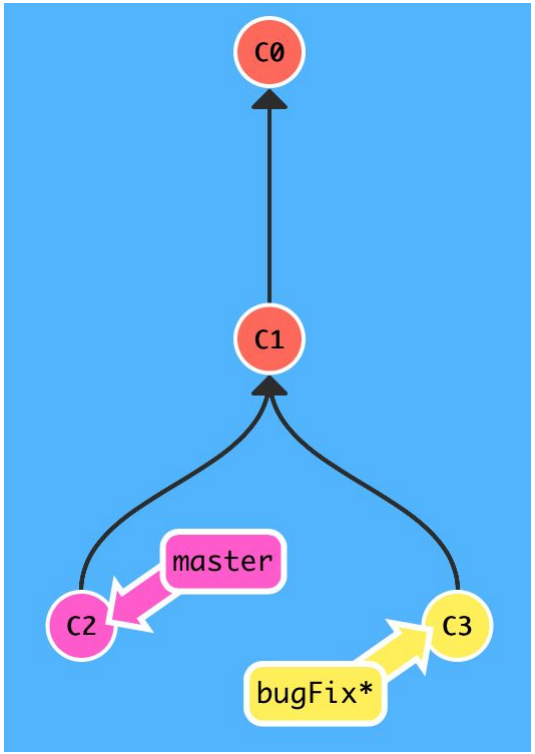
`git checkout bugfix; git merge master` (into bugFix)

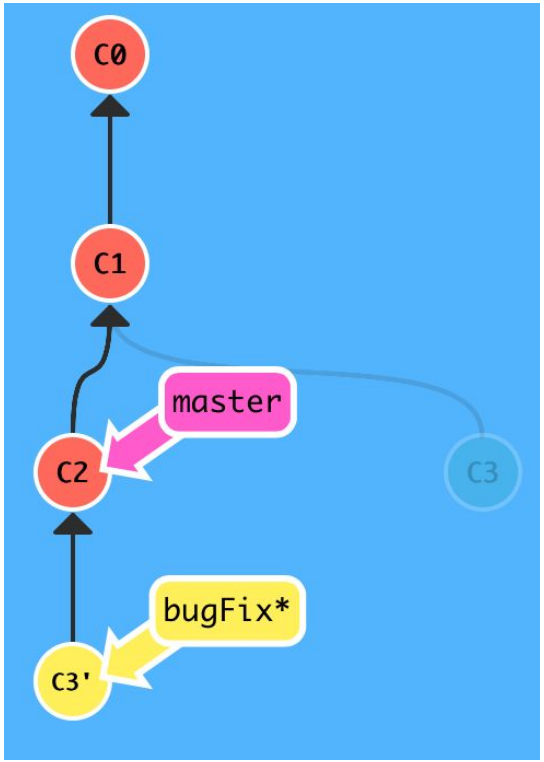




Move work from bugFix directly onto master

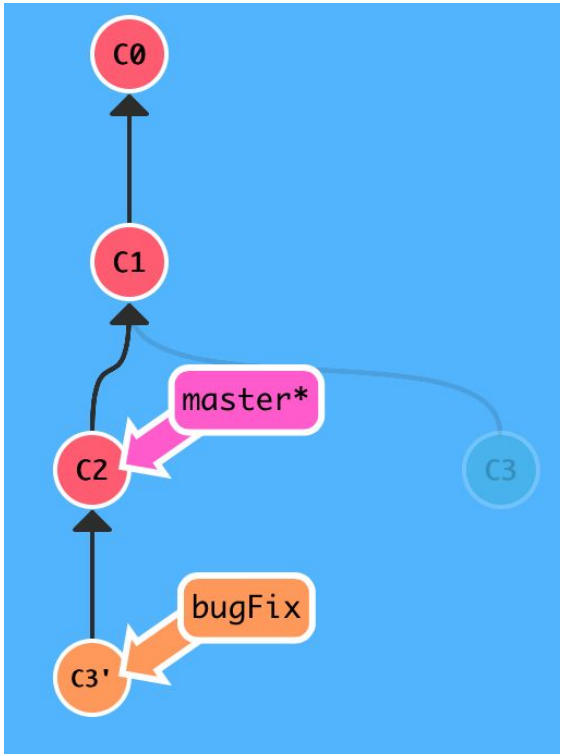
2) git rebase master

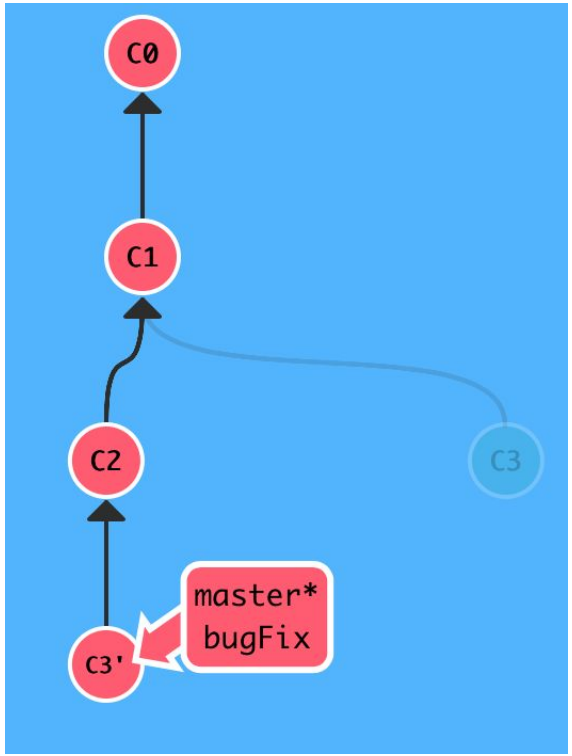




But master hasn't been updated, so:

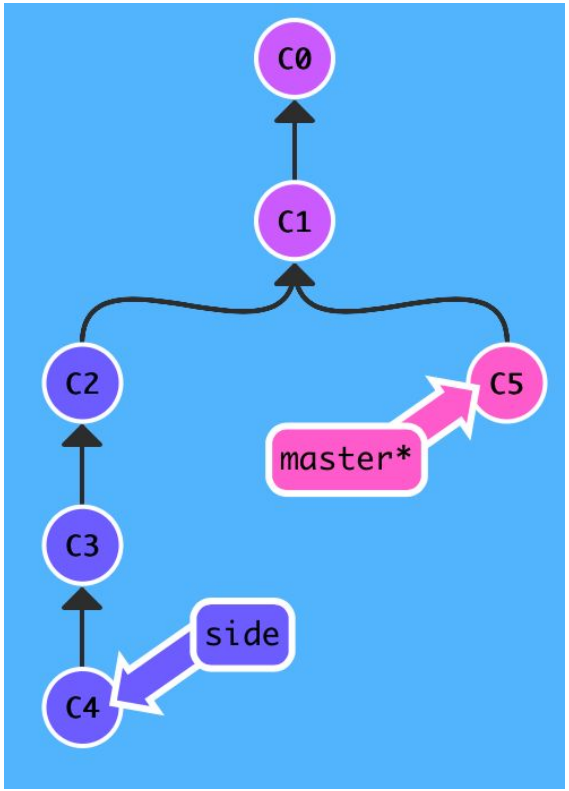
```
git checkout master; git rebase bugFix
```

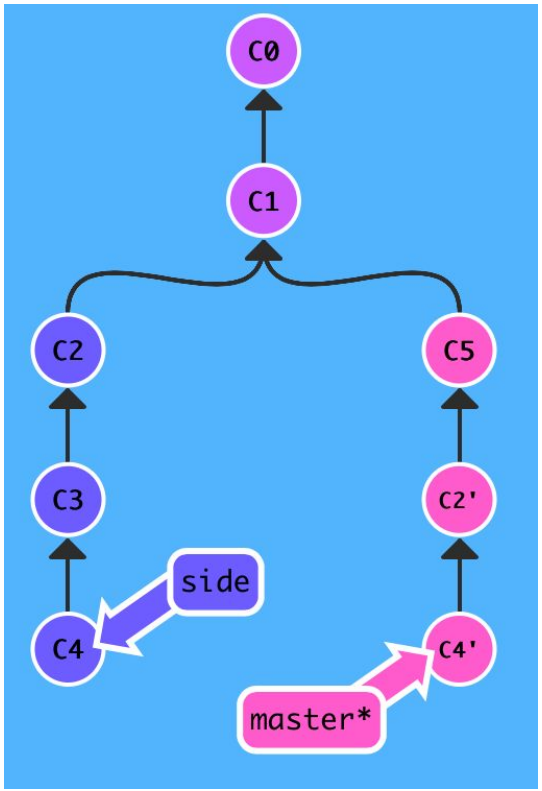




Copy a series of commits below current location

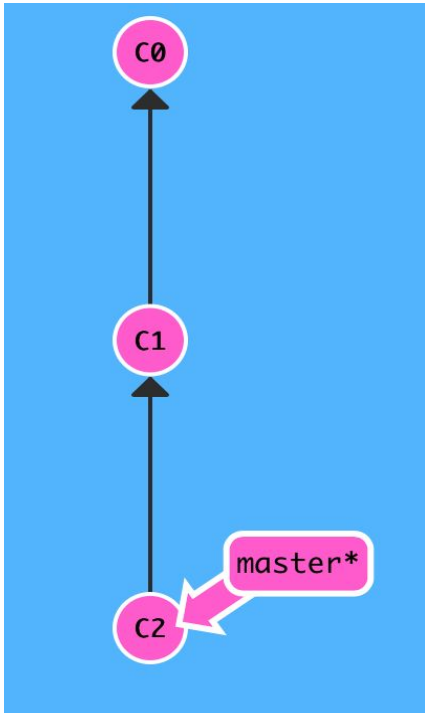
3) git cherry-pick C2 C4



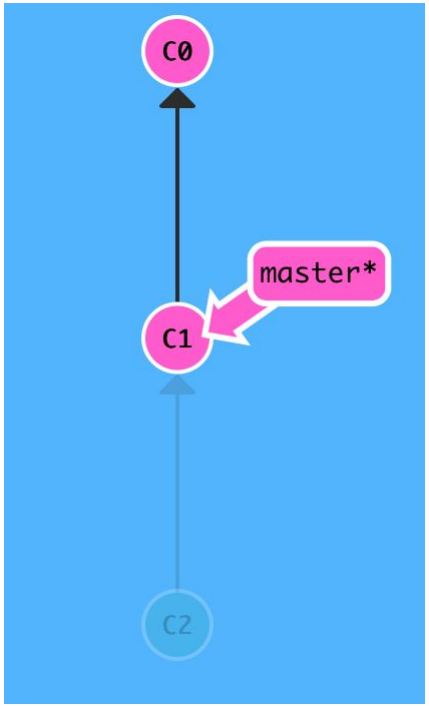


Ways to undo work (1)

```
git reset HEAD~1
```

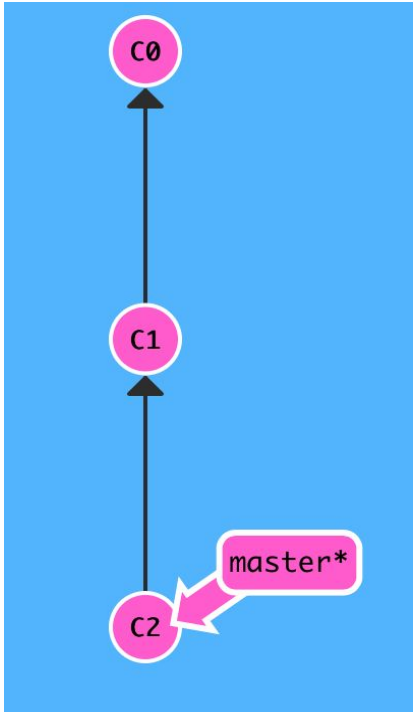


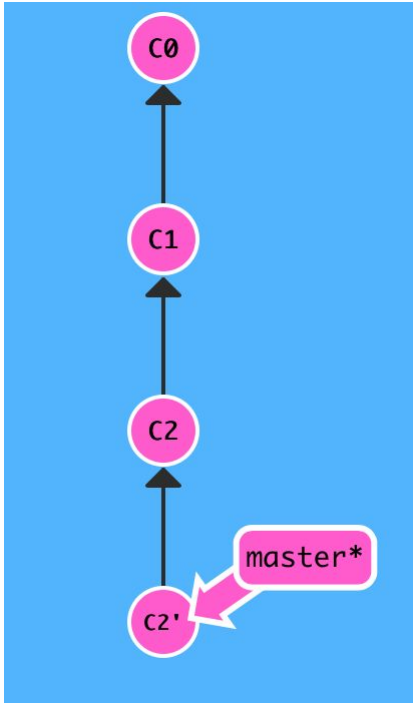
HEAD is the symbolic name for the currently checked out commit



Ways to undo work (2)

```
git revert HEAD
```

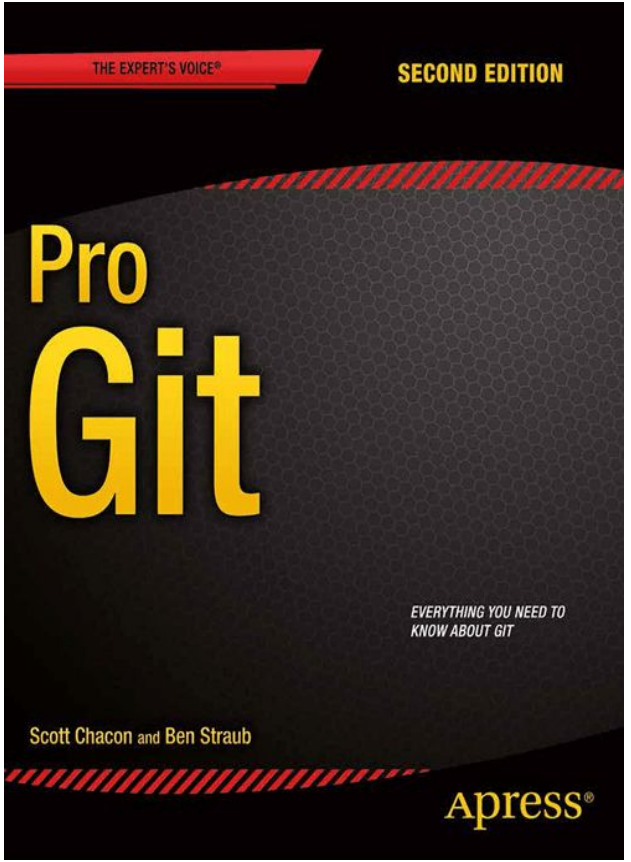




git reset does not work for remote branches

Highly Recommended

- (second?) Most useful life skill you will have learned in 214/514

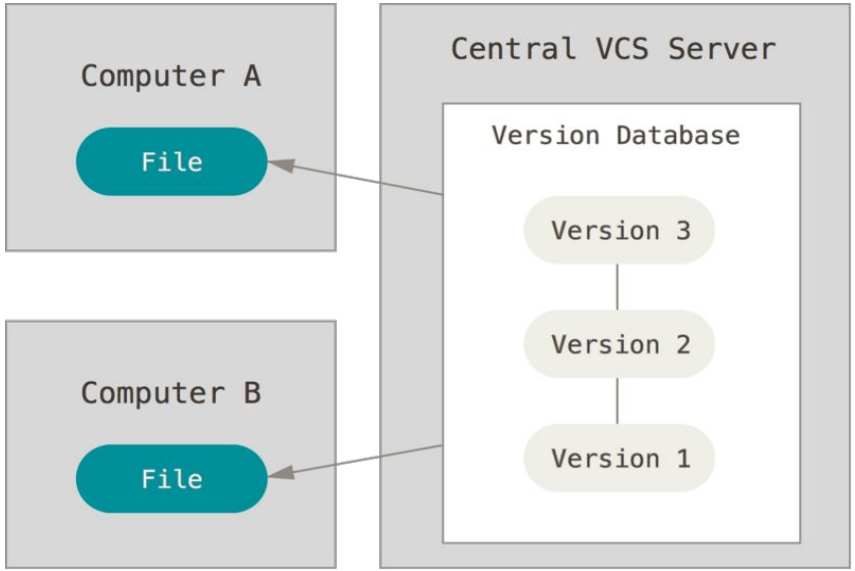


https://git-scm.com/book/en/v2

TYPES OF VERSION CONTROL

Centralized version control

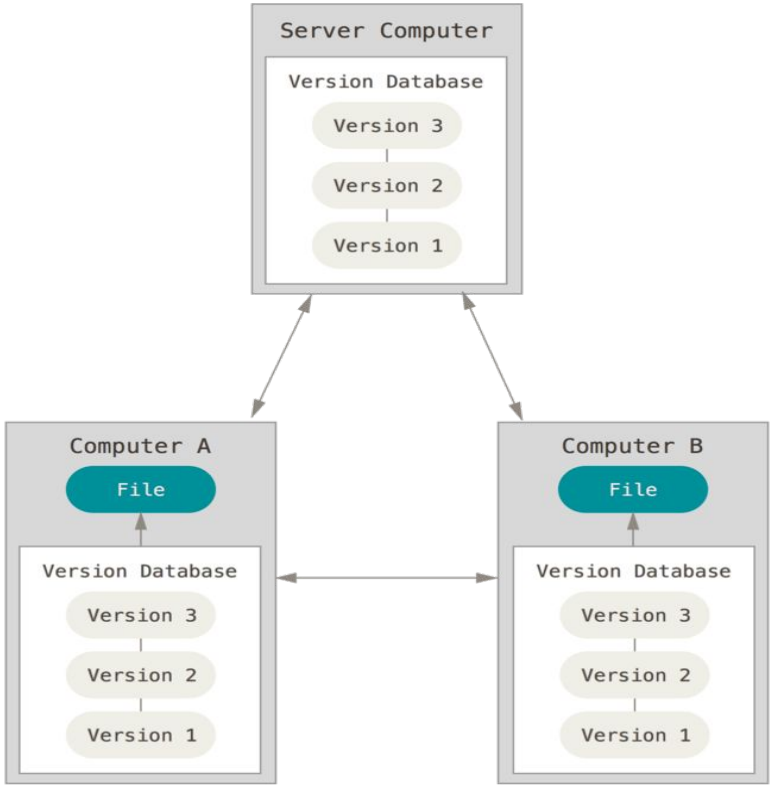
- Single server that contains all the versioned files
- Clients check out/in files from that central place
- E.g., CVS, SVN (Subversion), and Perforce



https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

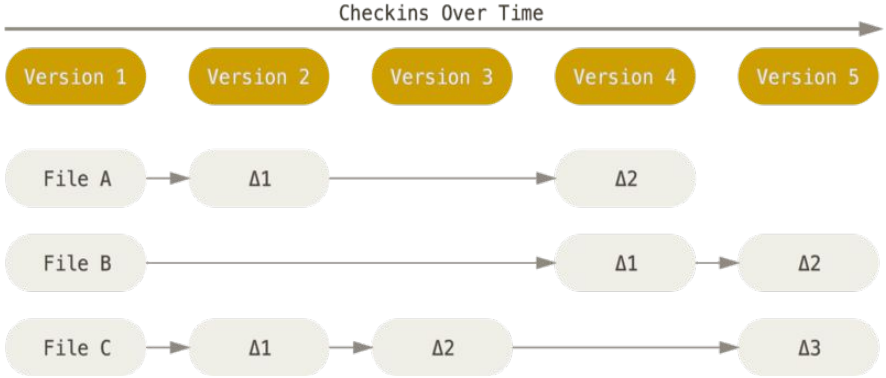
Distributed version control

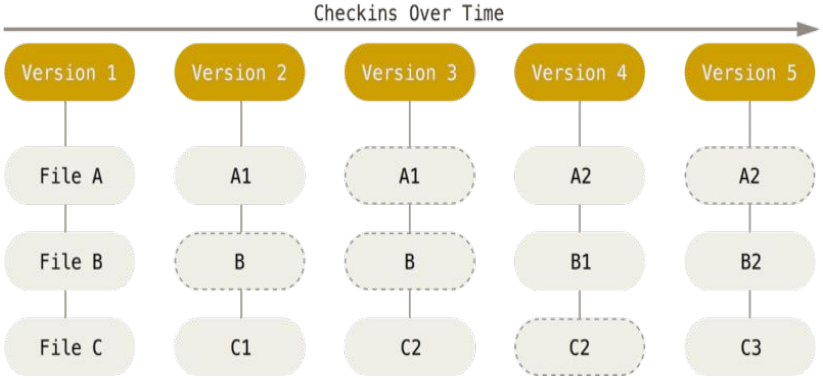
- Clients fully mirror the repository
  - Every clone is a full backup of ***all*** the data
- E.g., Git, Mercurial, Bazaar



https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

SVN (left) vs. Git (right)



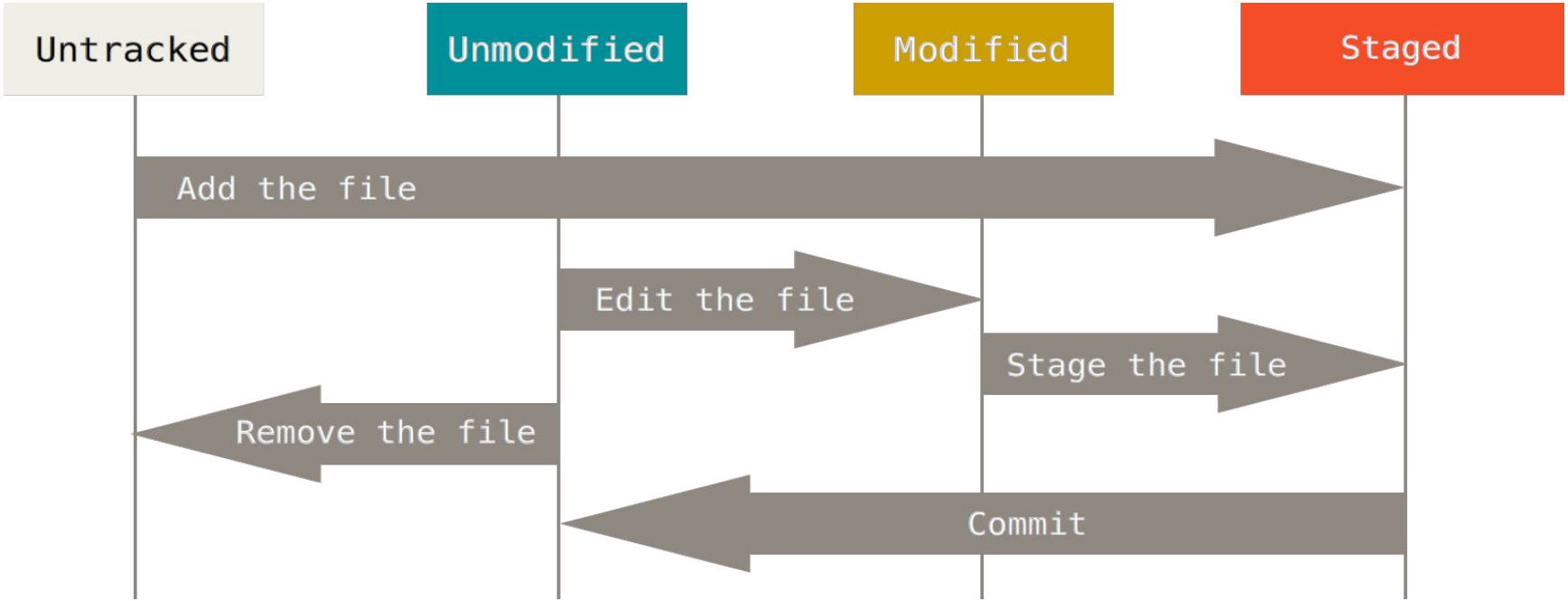


- SVN stores changes to a base version of each file
- Version numbers (1, 2, 3, …) are increased by one after each commit

- Git stores each version as a snapshot
- If files have not changed, only a link to the previous file is stored
- Each version is referred by the SHA-1 hash of the contents

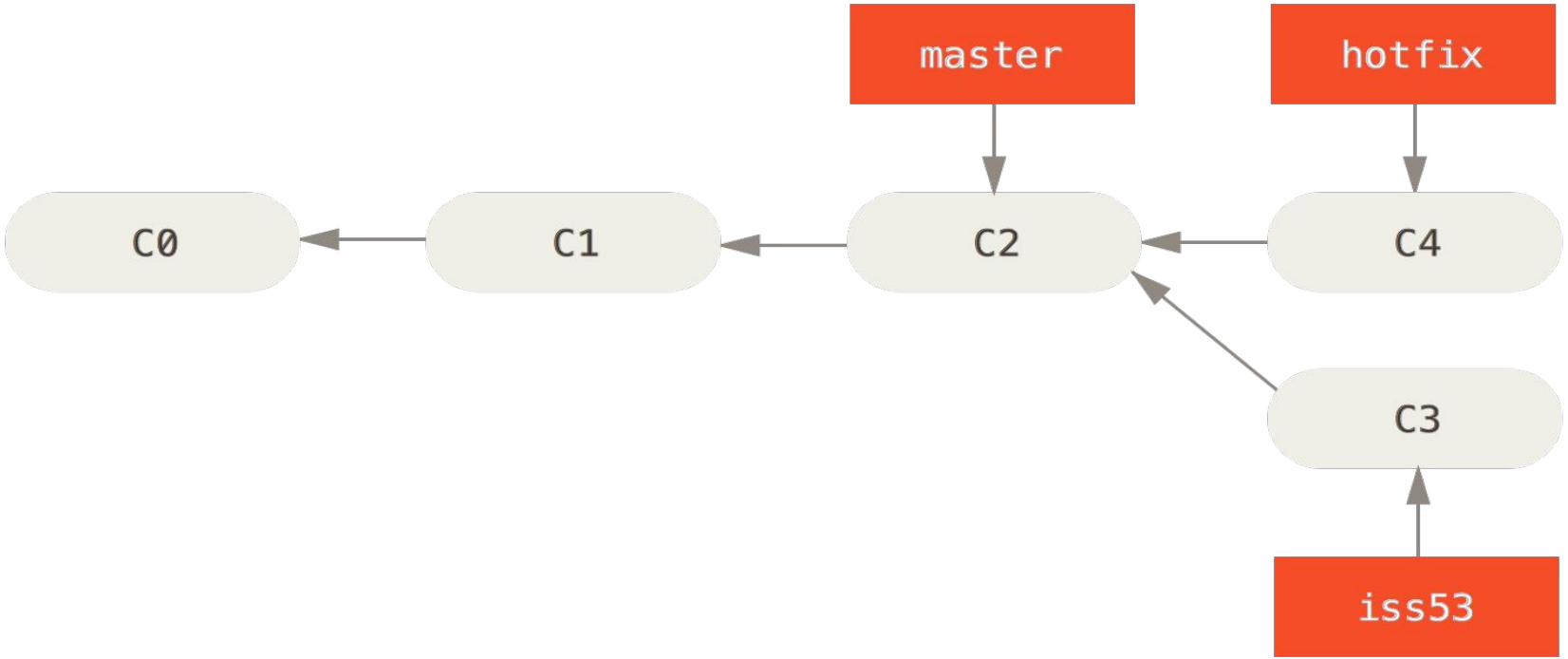
https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Aside: Git process



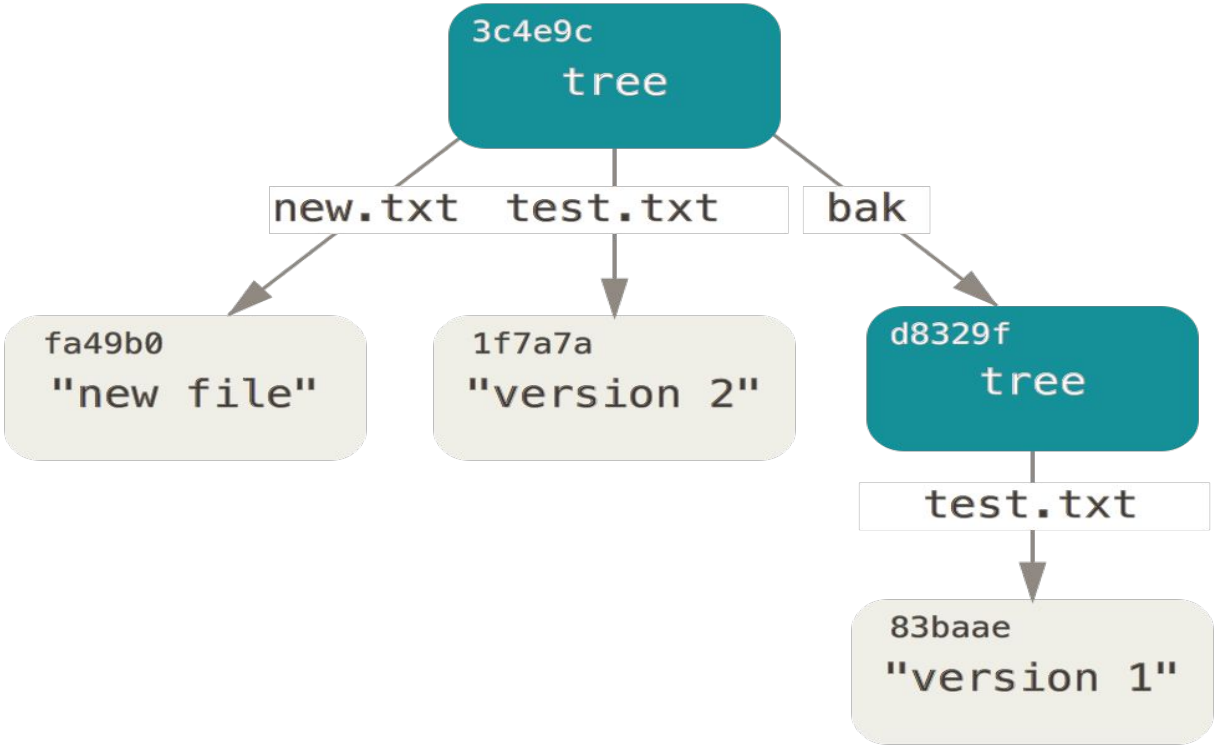
© Scott Chacon “Pro Git”

Git Internals



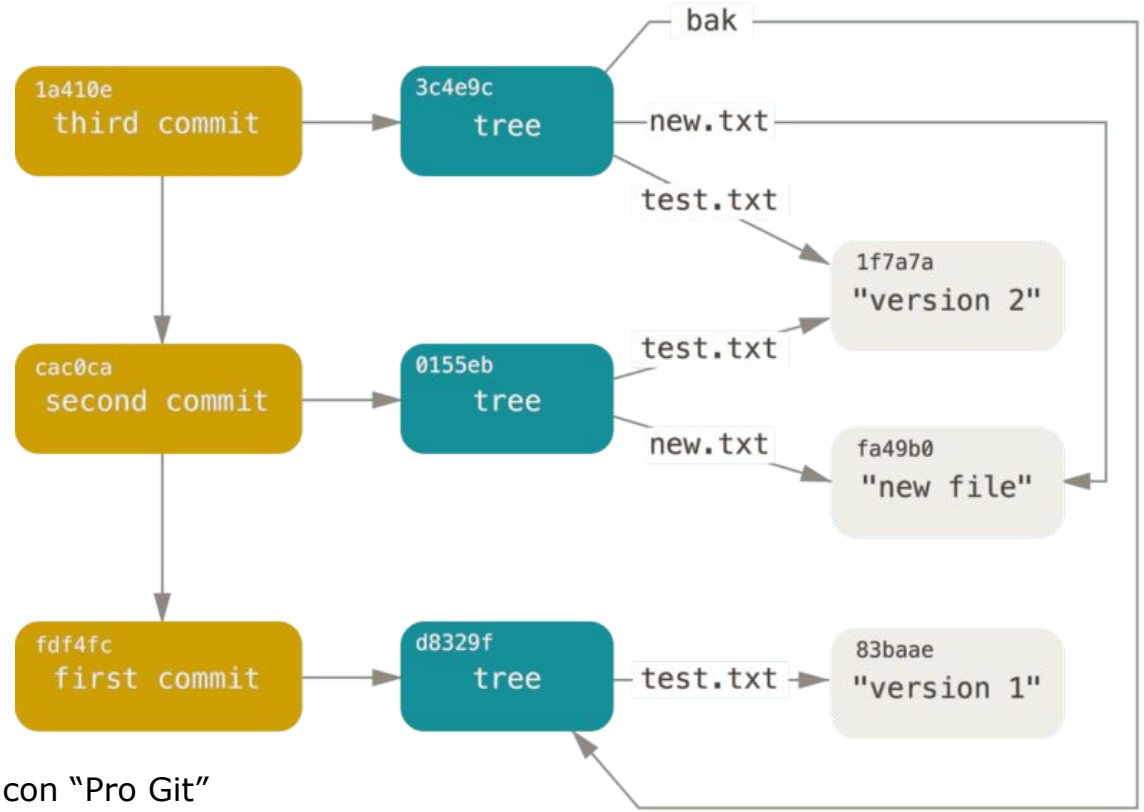
© Scott Chacon “Pro Git”

Git Internals



© Scott Chacon "Pro Git"

Aside: Git object graph



© Scott Chacon "Pro Git"

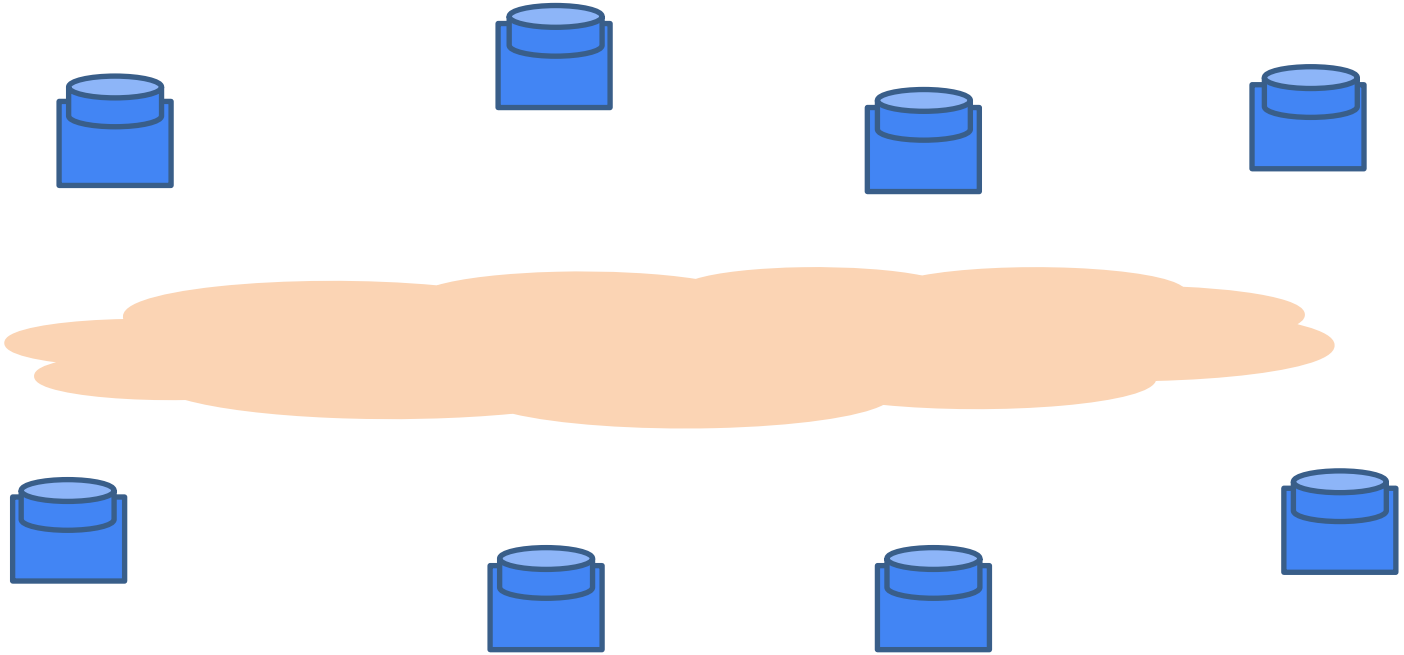
Aside: Which files to manage

- All code and noncode files
  - Java / JavaScript code
  - Build scripts
  - Documentation
- Exclude generated files (.class, node_modules, …)
- Most version control systems have a mechanism to exclude files (e.g., .gitignore)

SYNCING LOCAL ↔ REMOTE

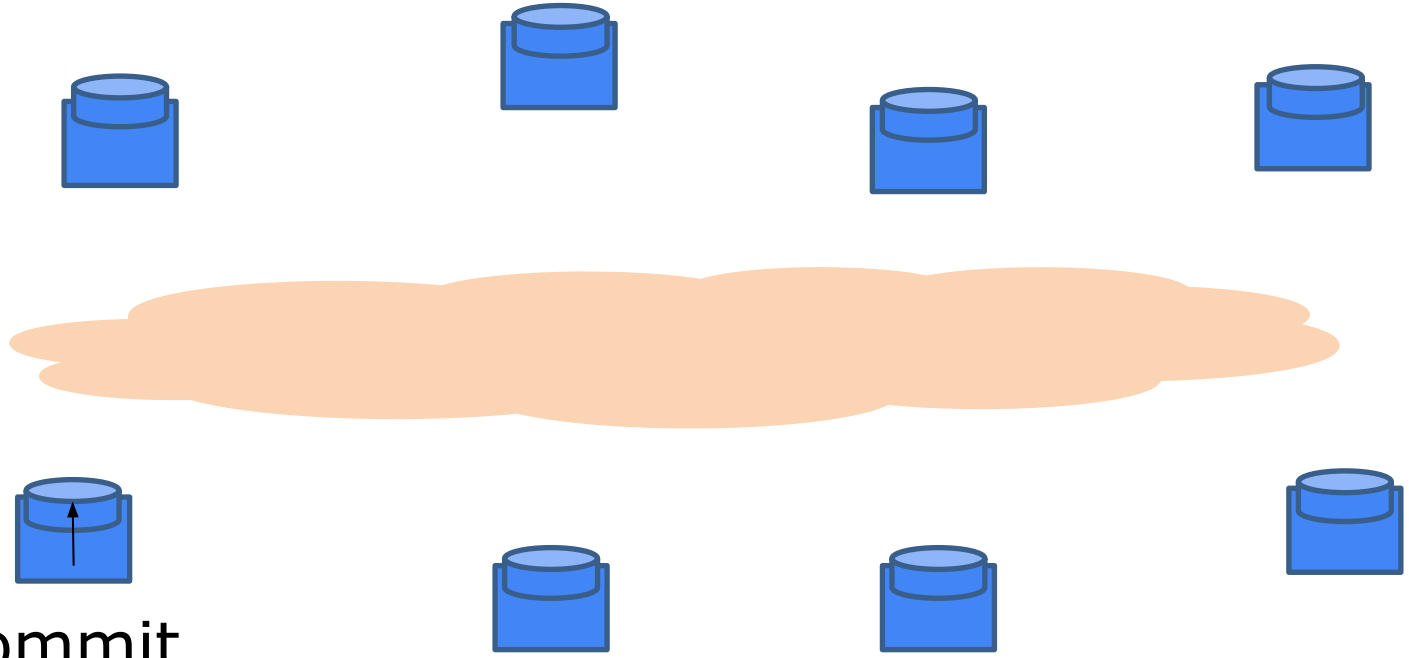
Git

Every computer is a server and version control happens locally.



Git

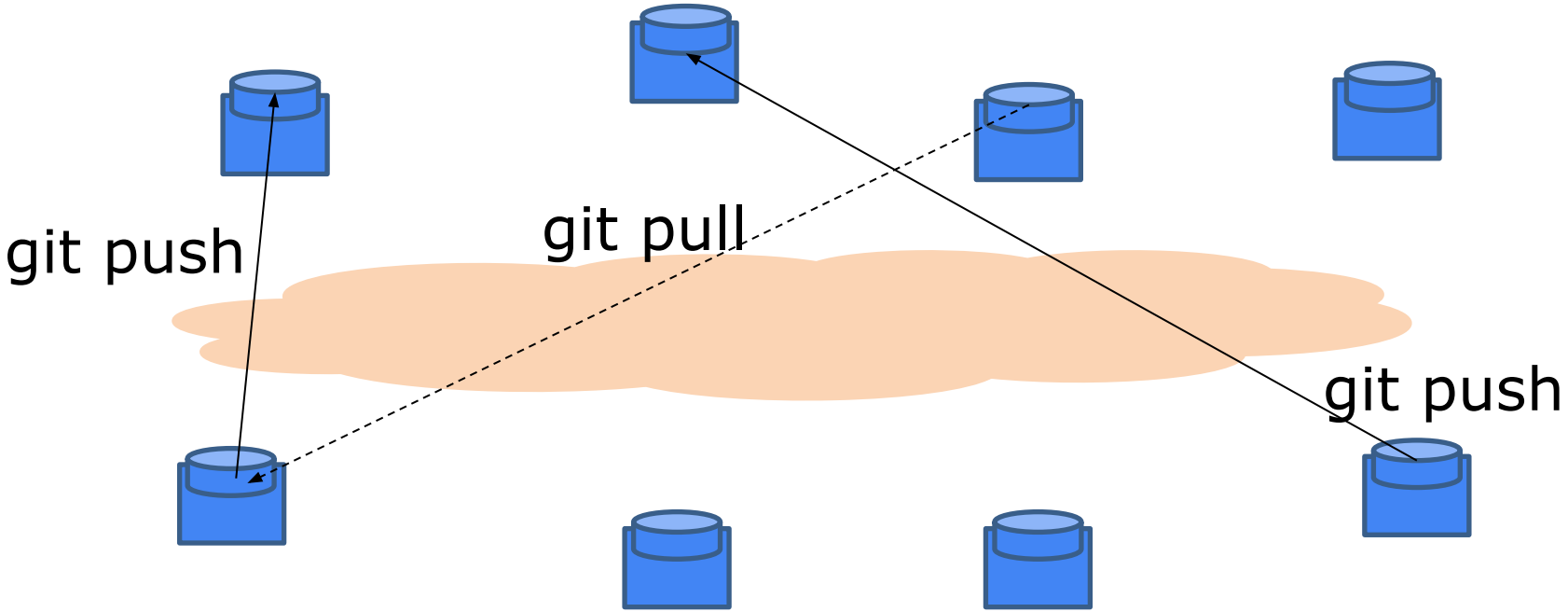
How do you share code with collaborators if commits are *local*?



git commit

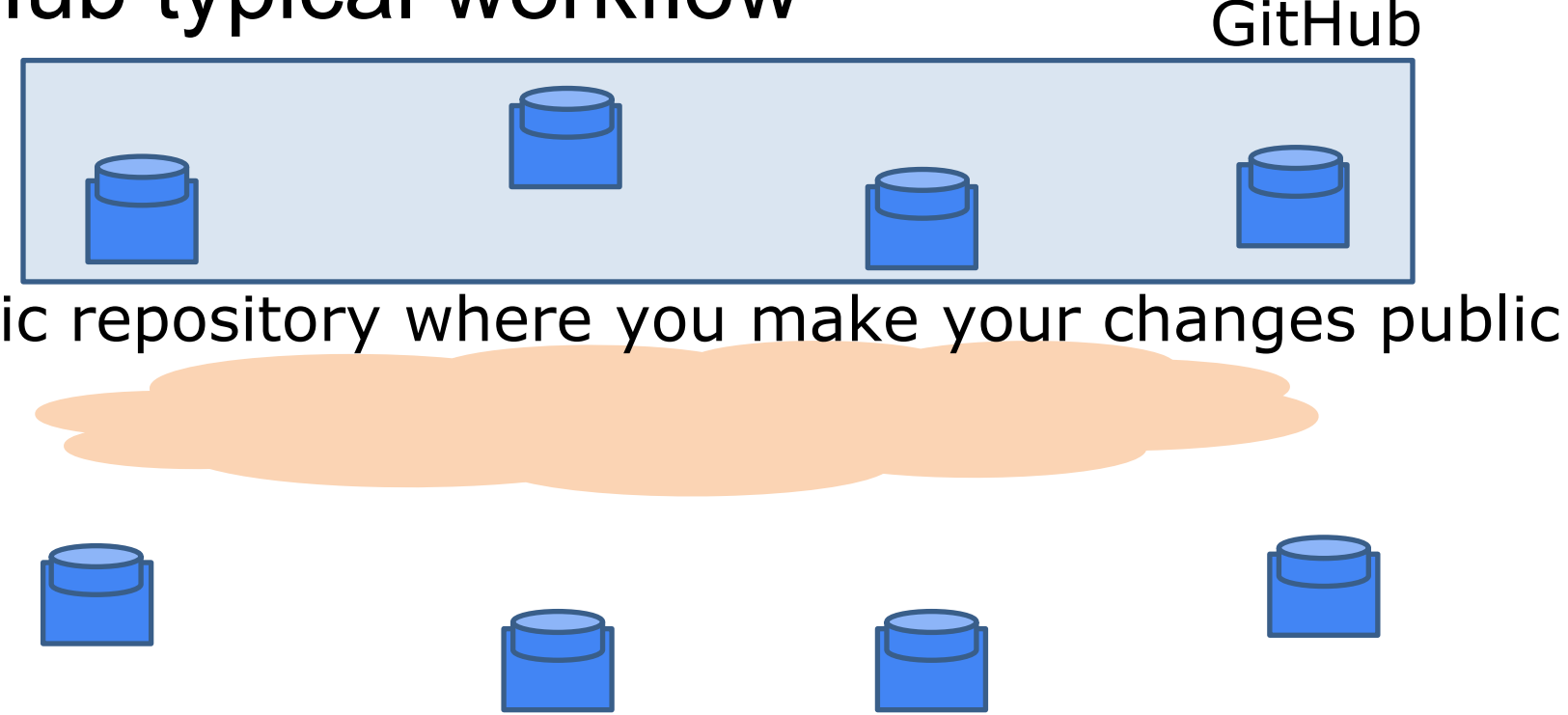
Git

You *push* your commits into their repositories /
They *pull* your commits into their repositories

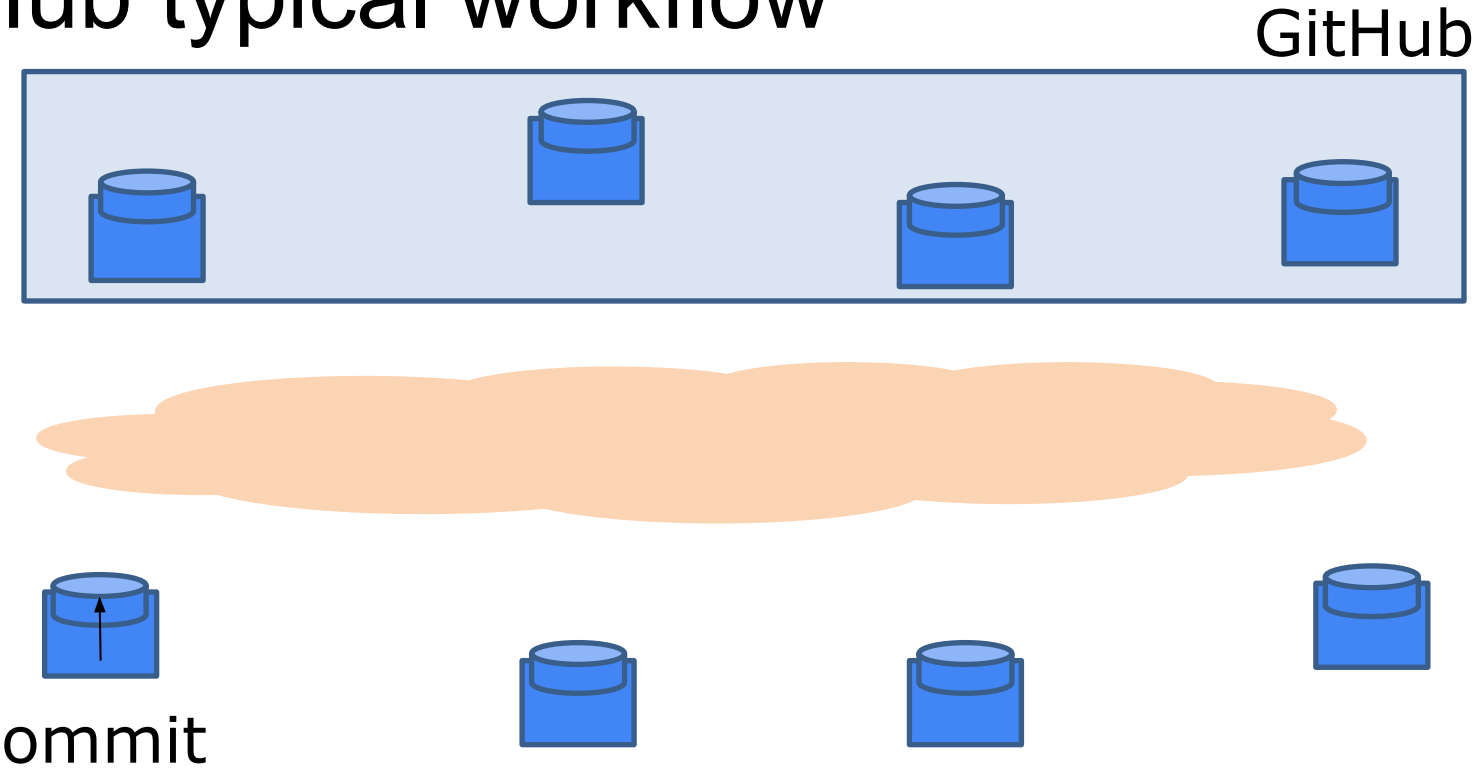


… But requires host names / IP addresses

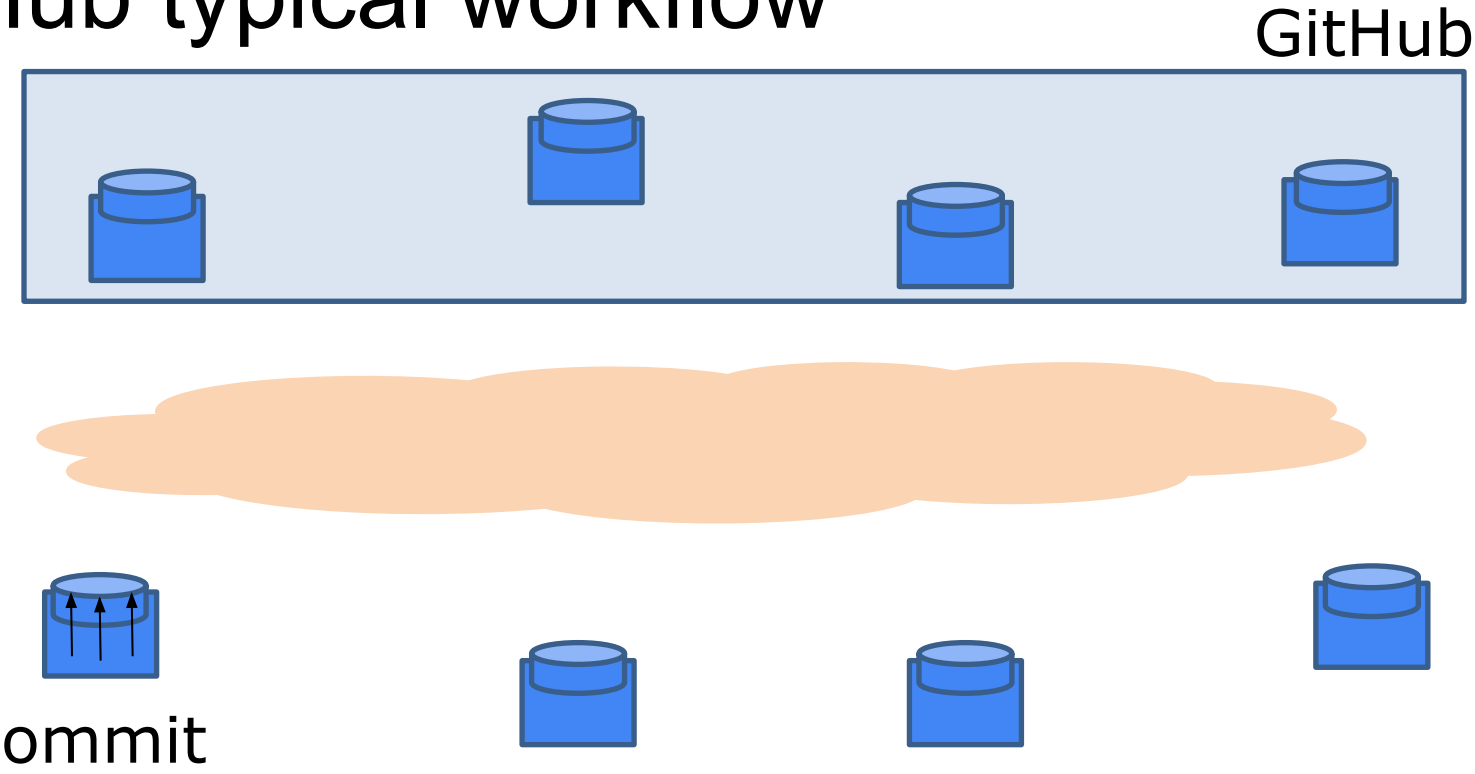
GitHub typical workflow



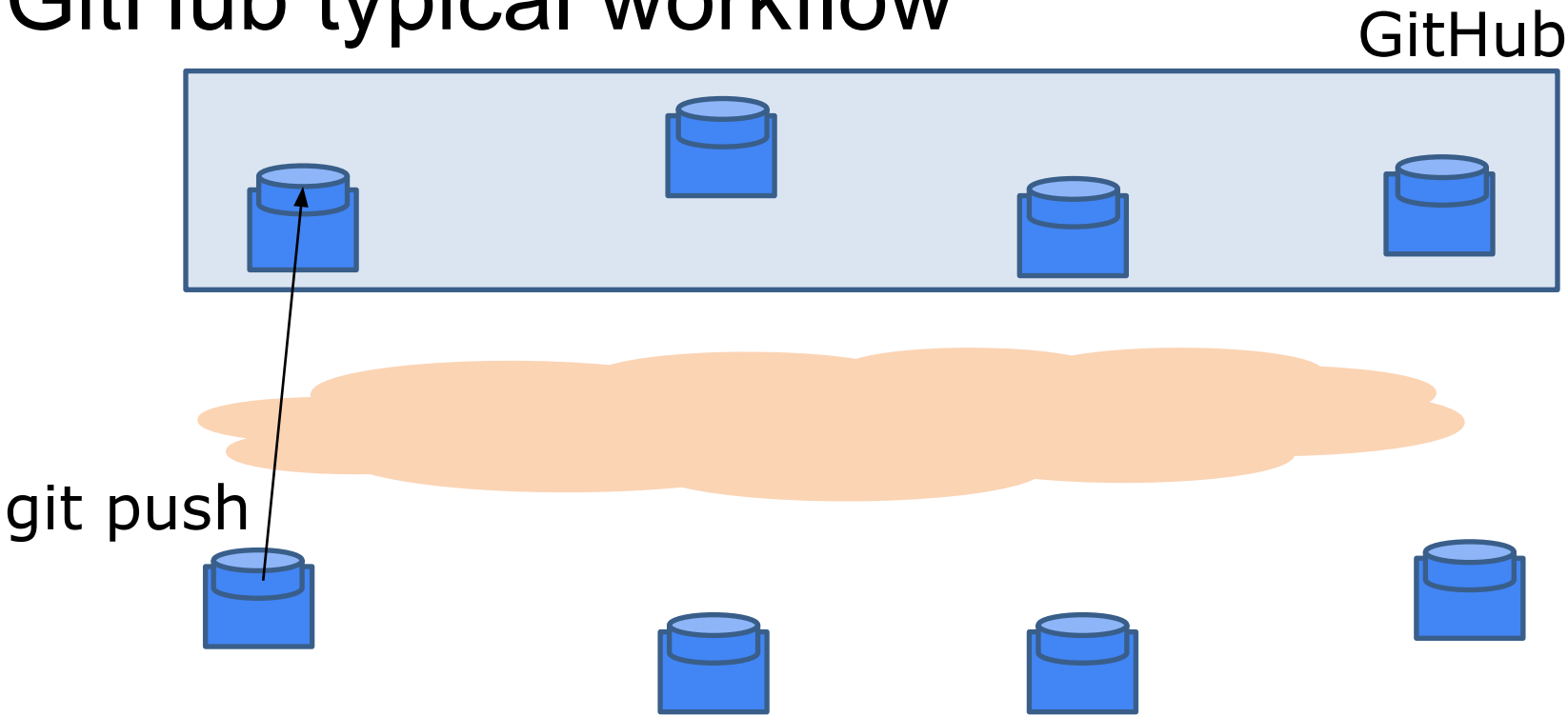
GitHub typical workflow



GitHub typical workflow

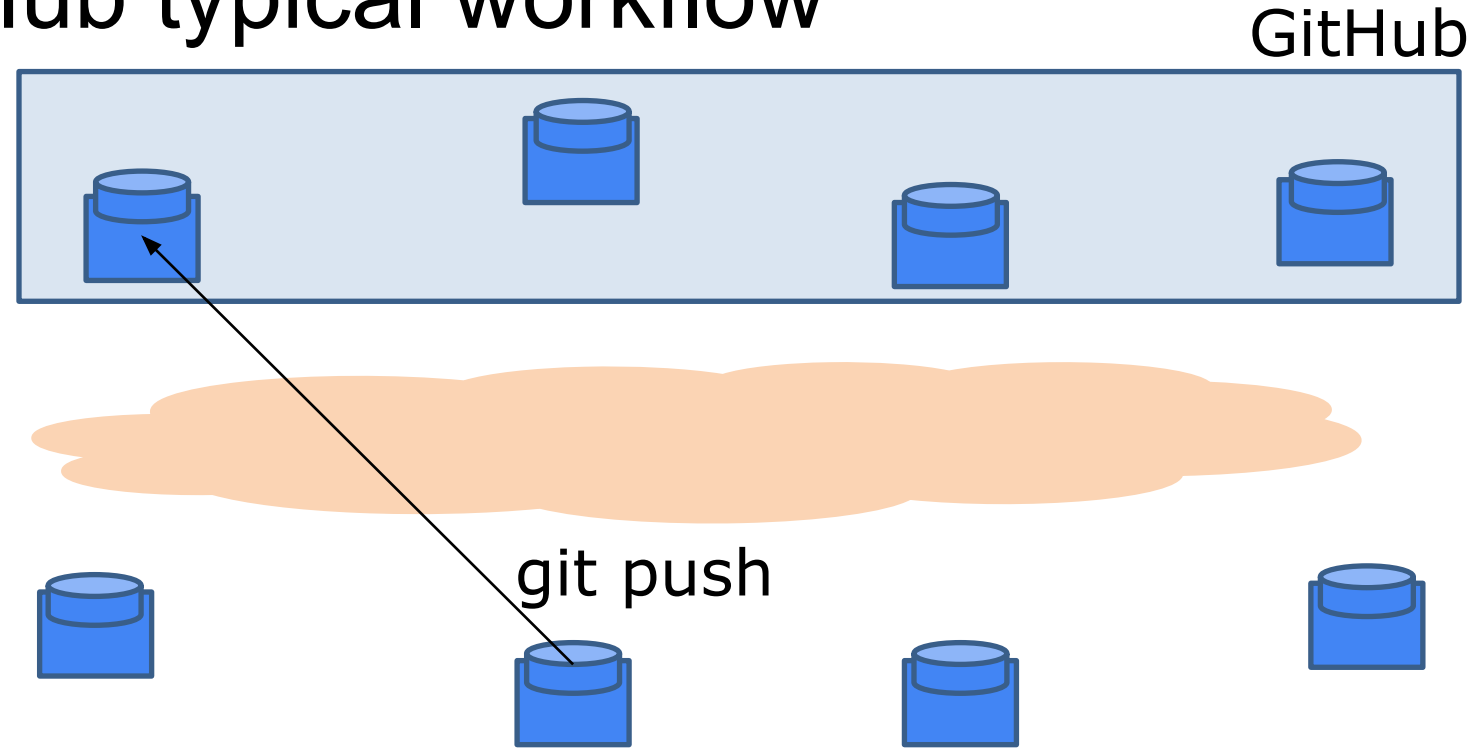


GitHub typical workflow



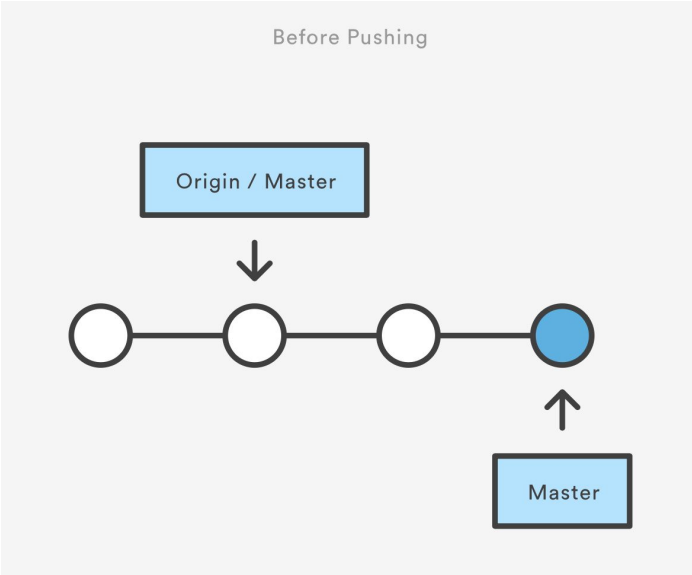
push your local changes into a remote repository.

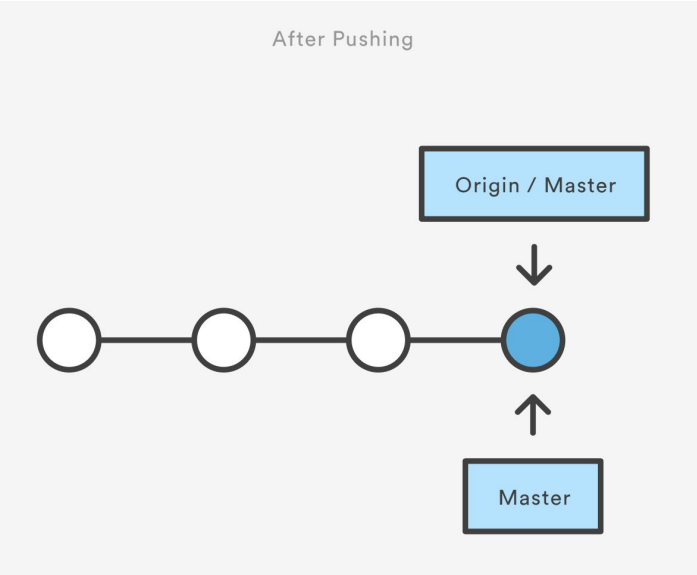
GitHub typical workflow



Collaborators can push too if they have access rights.

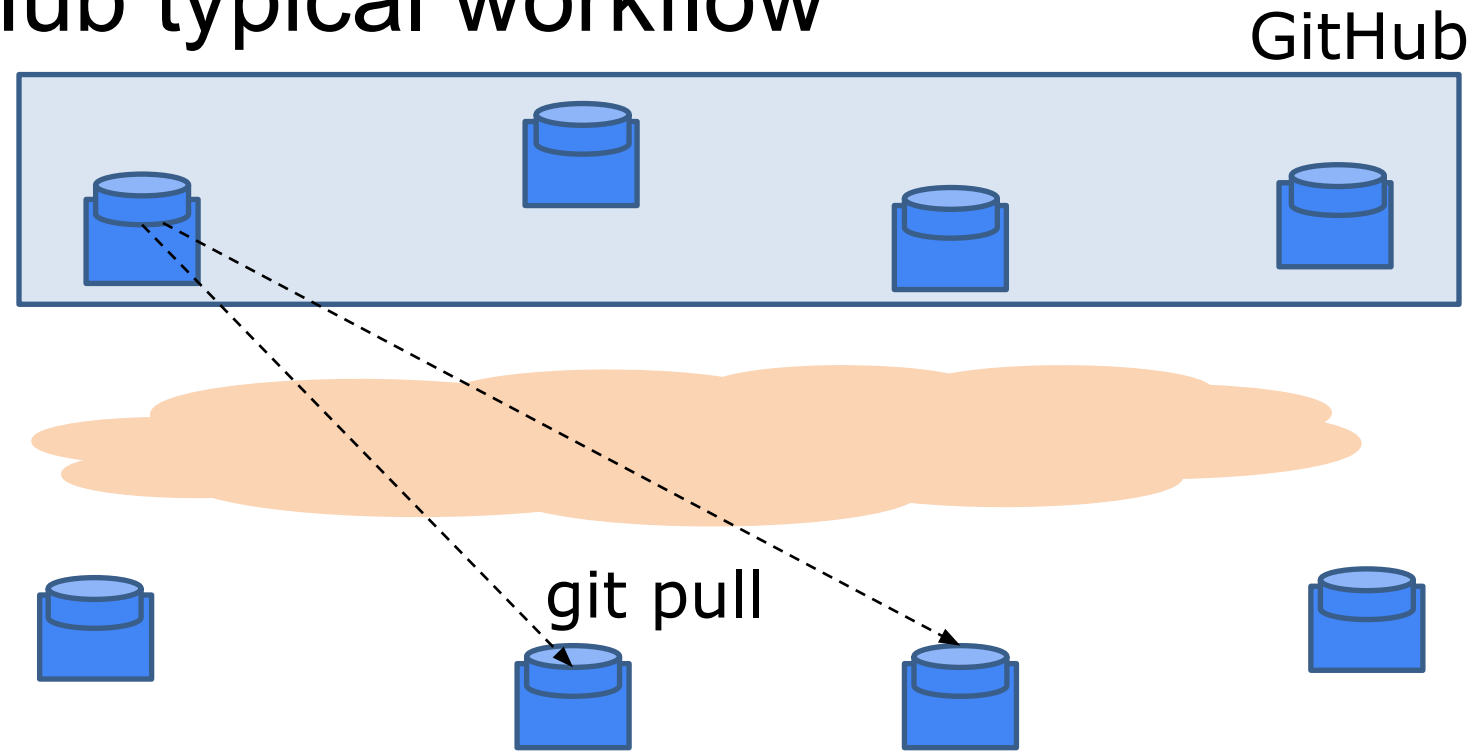
`git push <remote> <branch>`: upload local repository content to a remote repository





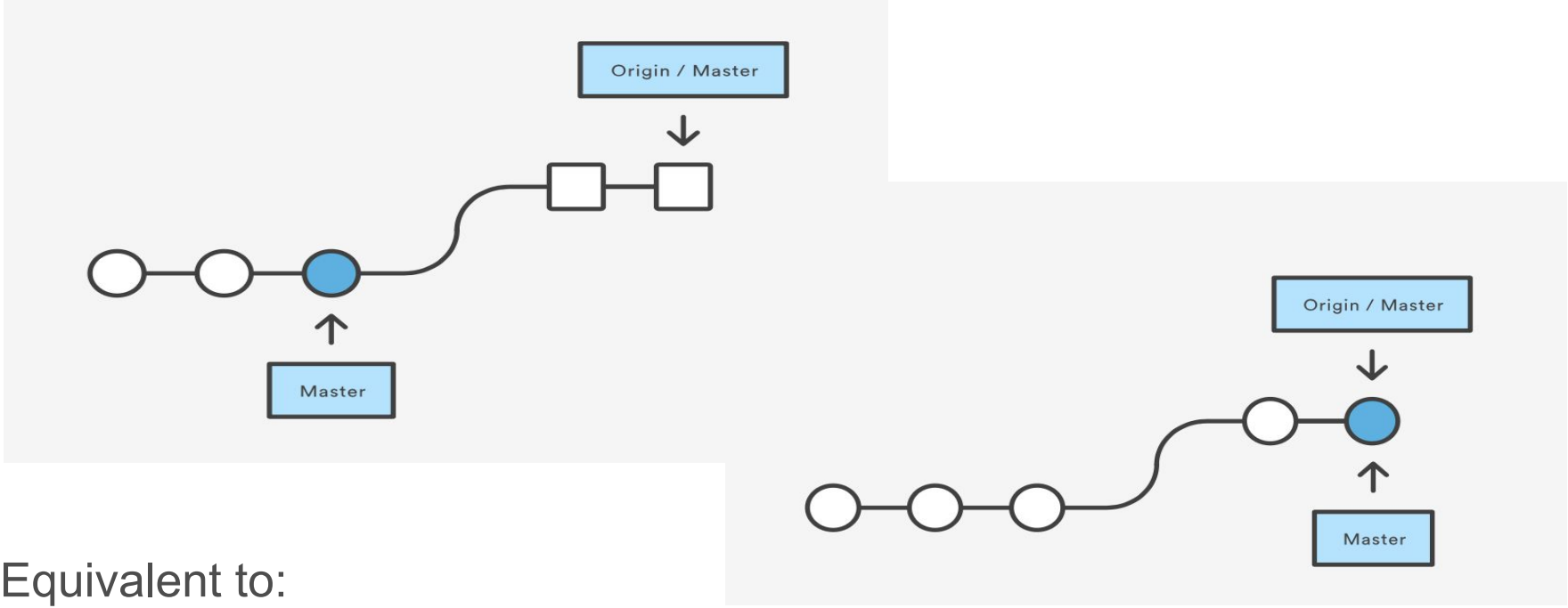
https://www.atlassian.com/git/tutorials/syncing/git-push

GitHub typical workflow



Without access rights, “don’t call us, we’ll call you” (*pull* from trusted sources) … But again requires host names / IP addresses.

`git pull <remote>`: Fetch the specified remote's copy of the current branch and immediately merge it into the local copy

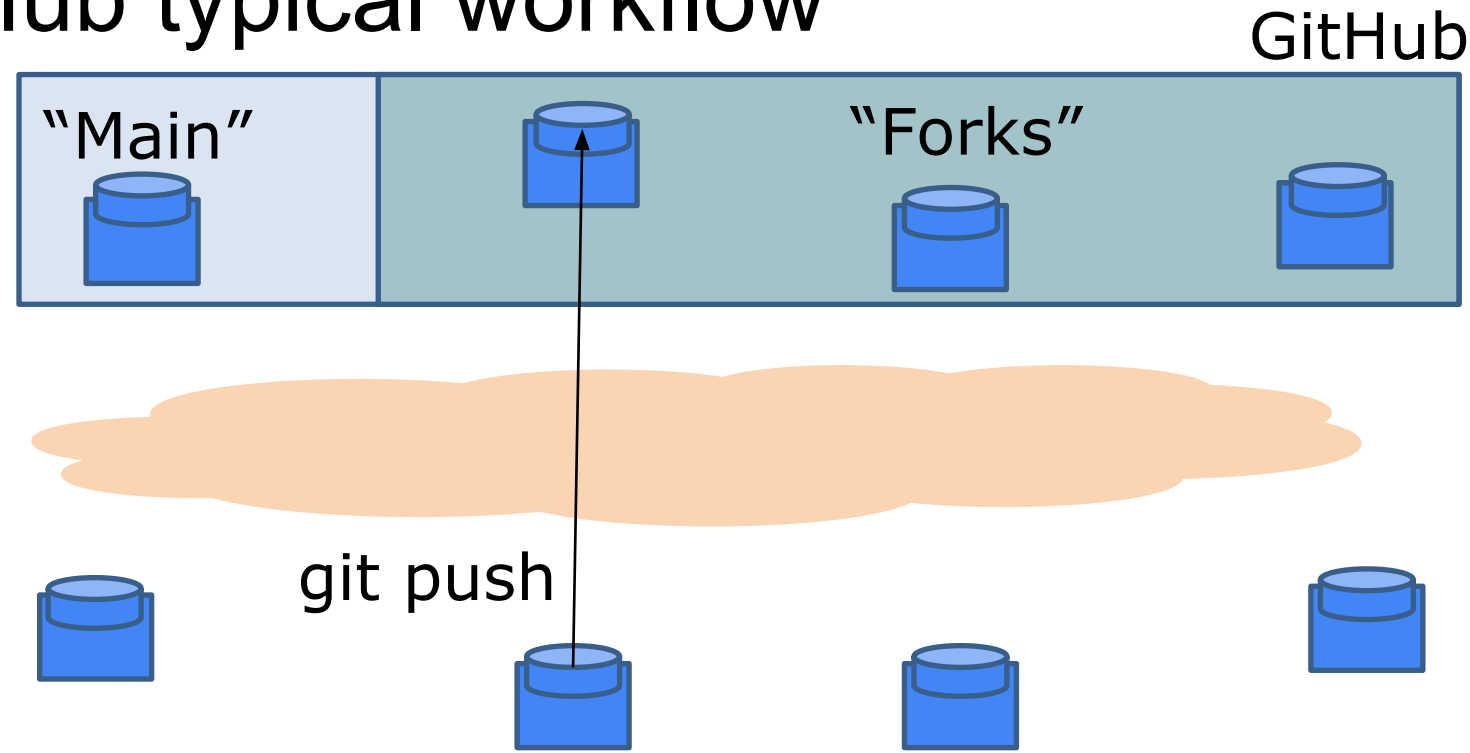


Equivalent to:

`git fetch origin HEAD` + `git merge HEAD`

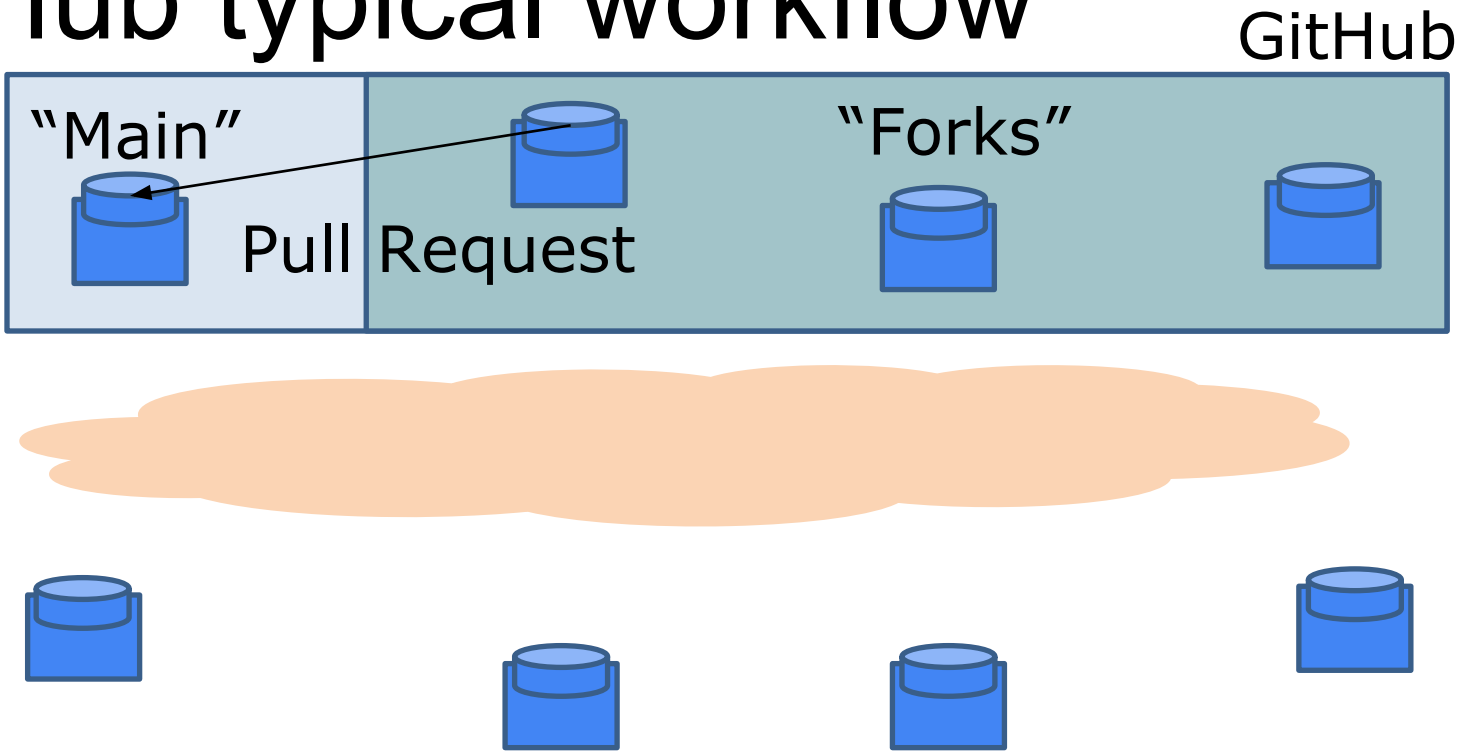
Also possible: `git pull --rebase origin`

GitHub typical workflow



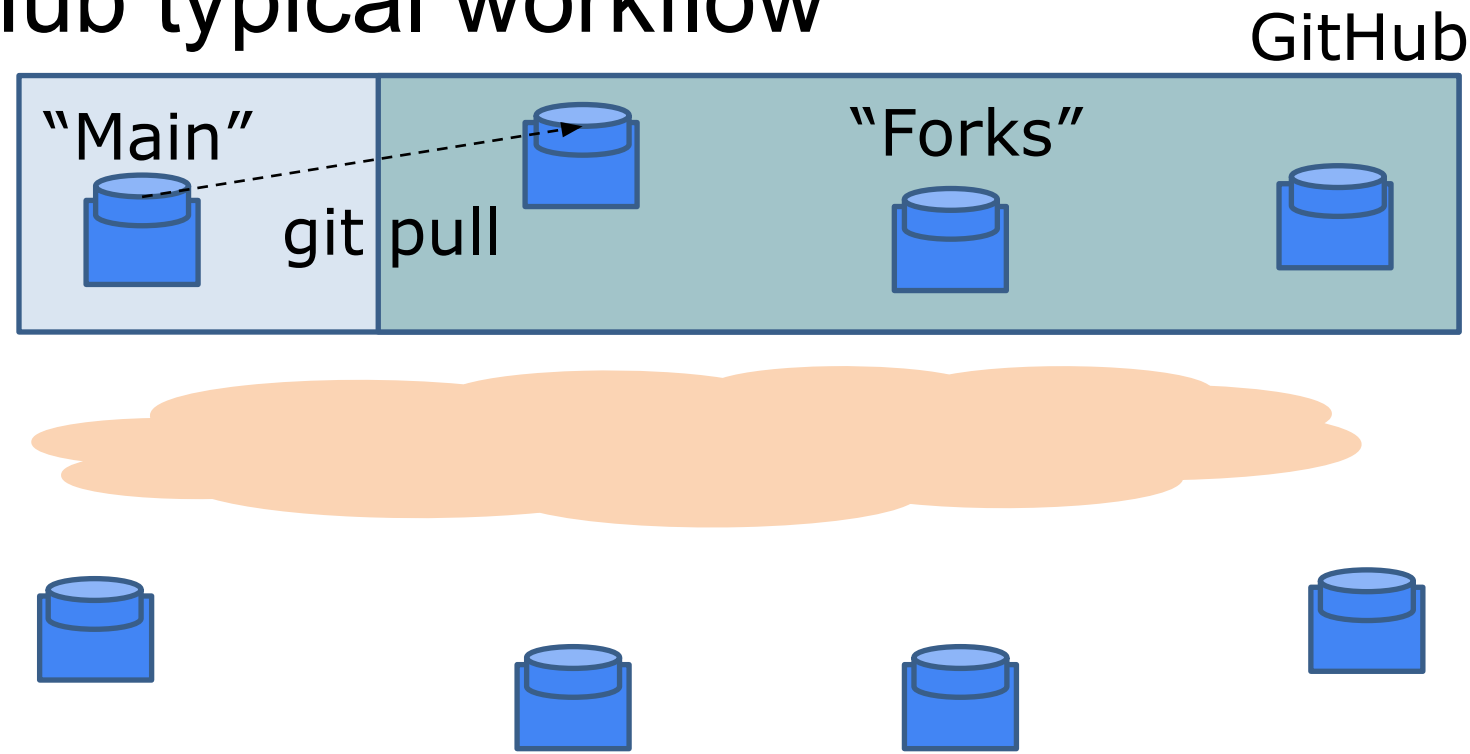
Instead, people maintain public remote “forks” of “main” repository on GitHub and push local changes.

GitHub typical workflow



Availability of new changes is signaled via ”Pull Request”.

GitHub typical workflow



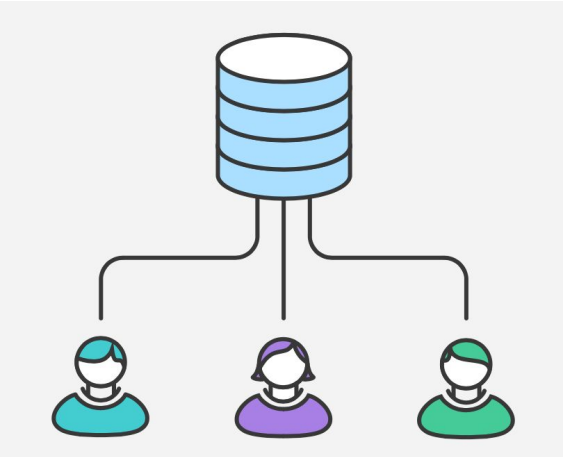
Changes are pulled into main if PR accepted.

BRANCH WORKFLOWS

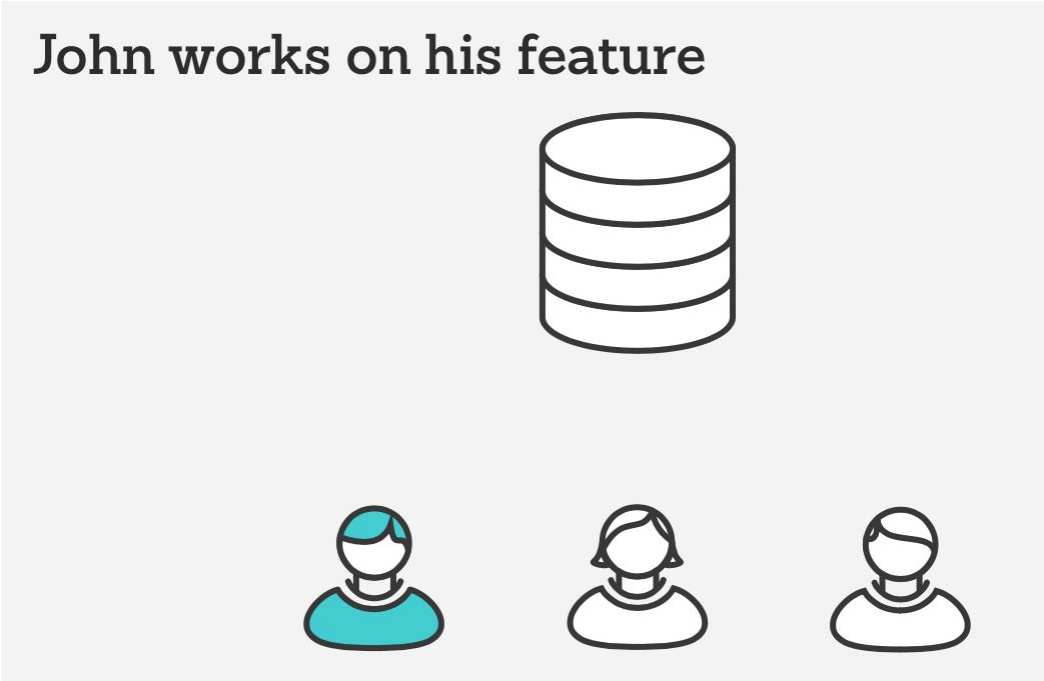
https://www.atlassian.com/git/tutorials/comparing-workflows

1. Centralized workflow

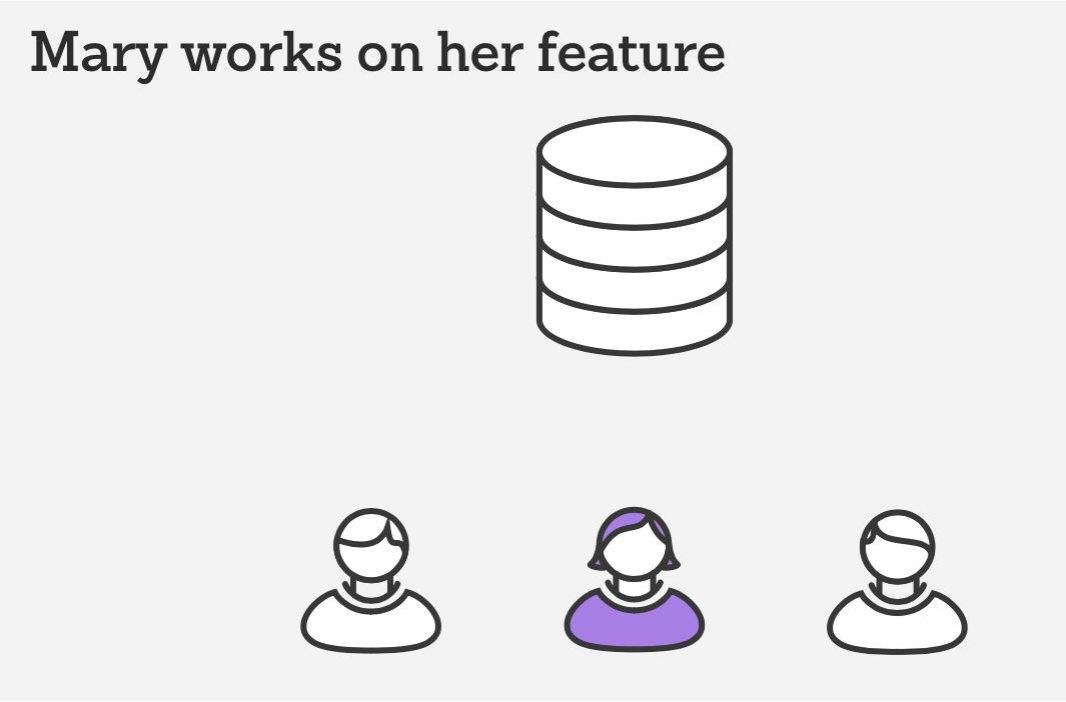
- Central repository to serve as the single point-of-entry for all changes to the project
- Default development branch is called **main**
  - all changes are committed into **main**
  - doesn’t require any other branches



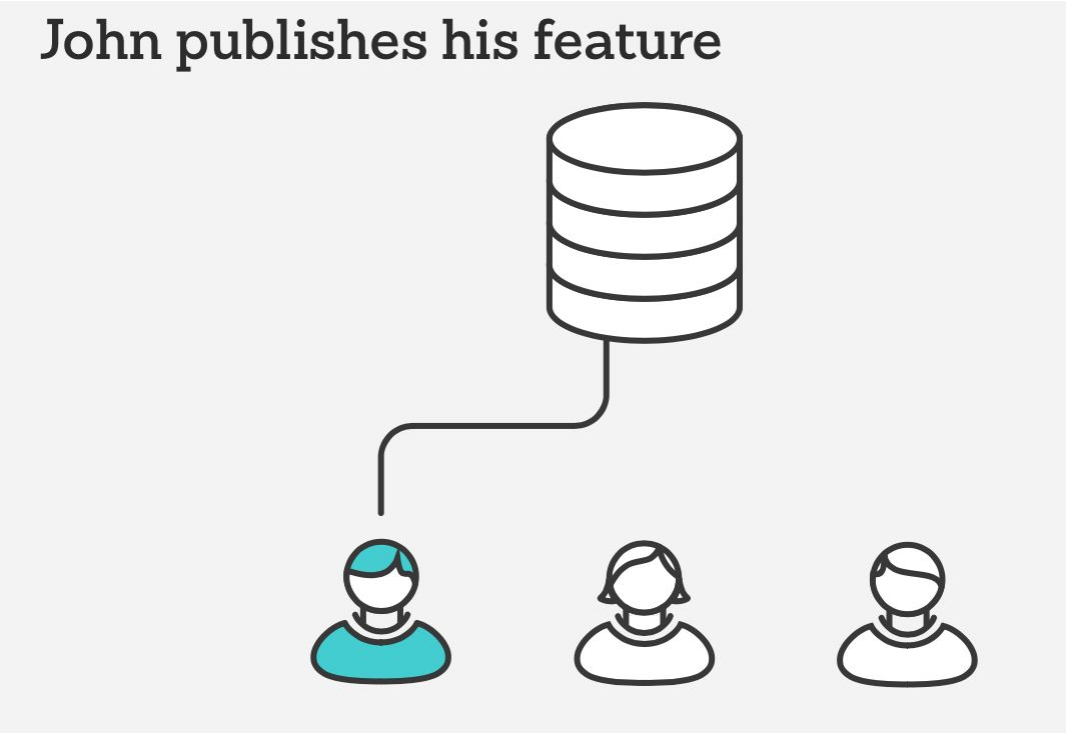
Example



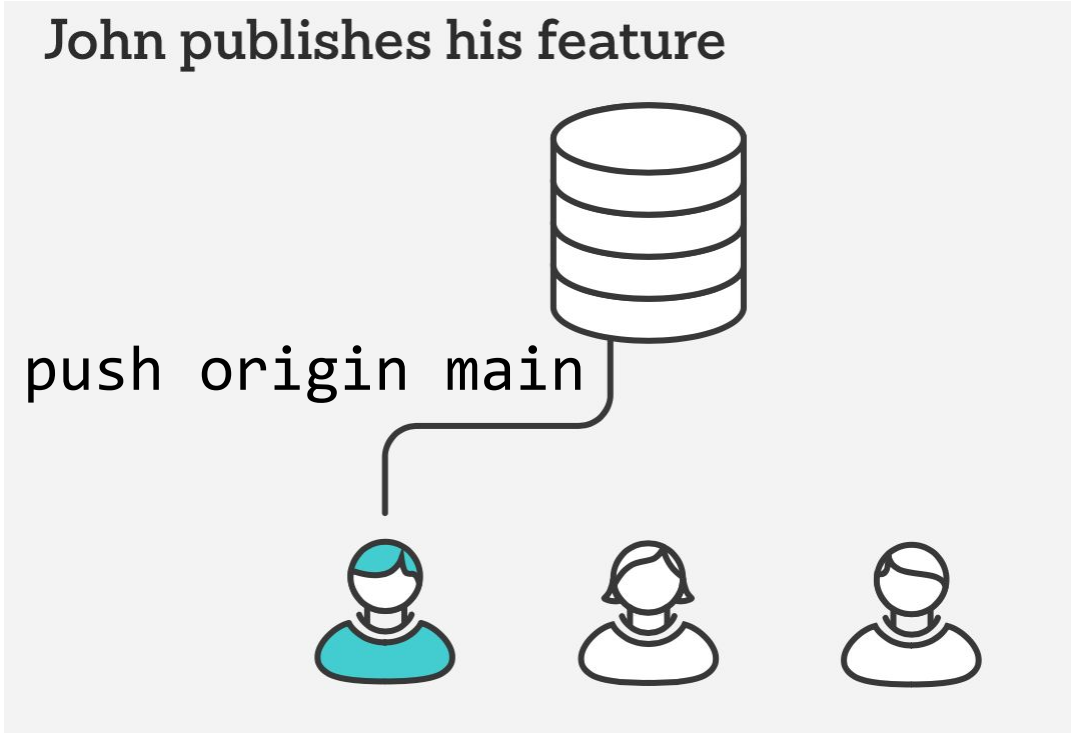
Example



Example

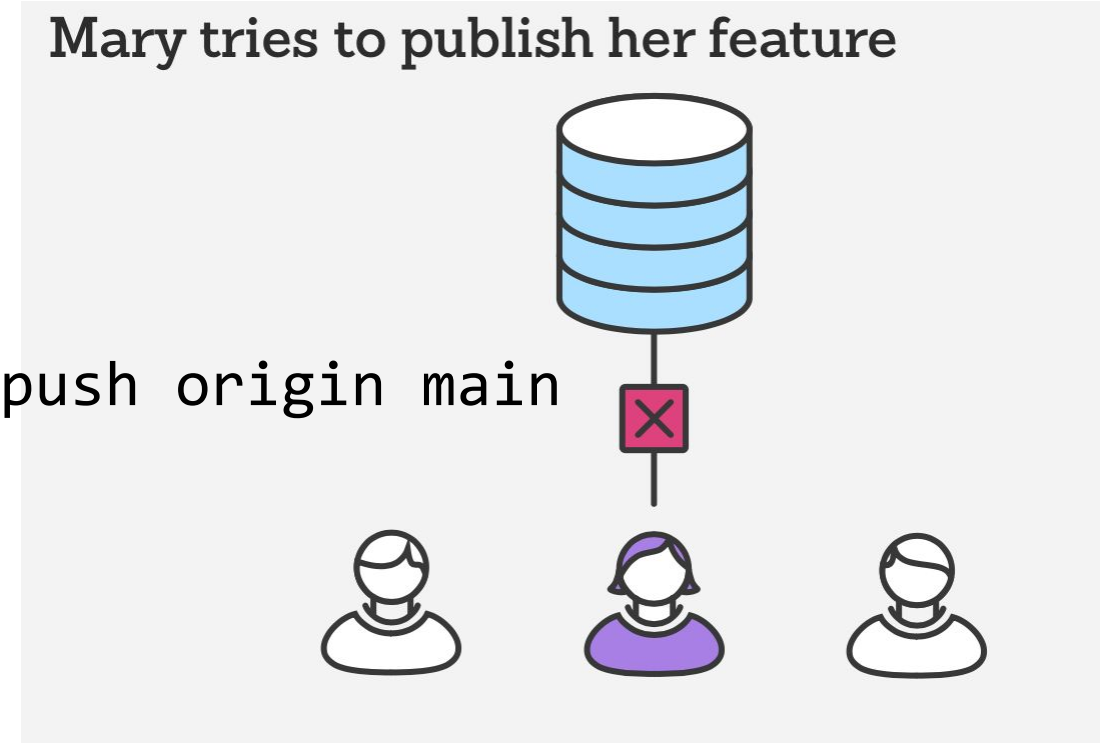


Example

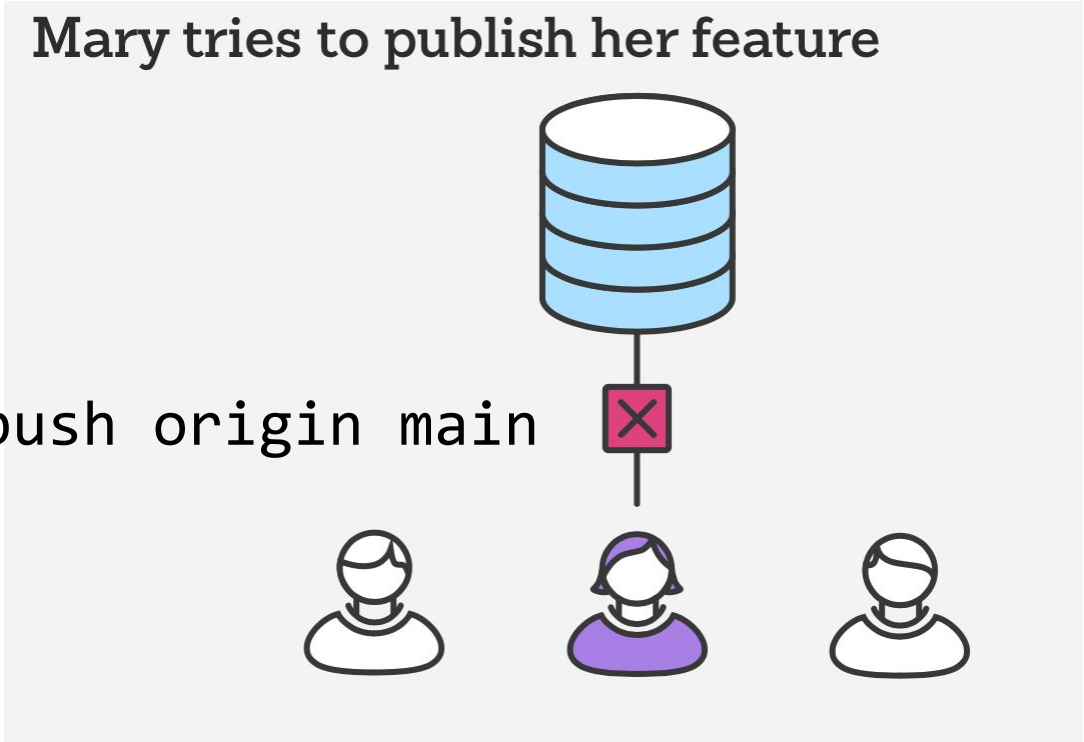


Example

```
git push origin main
```

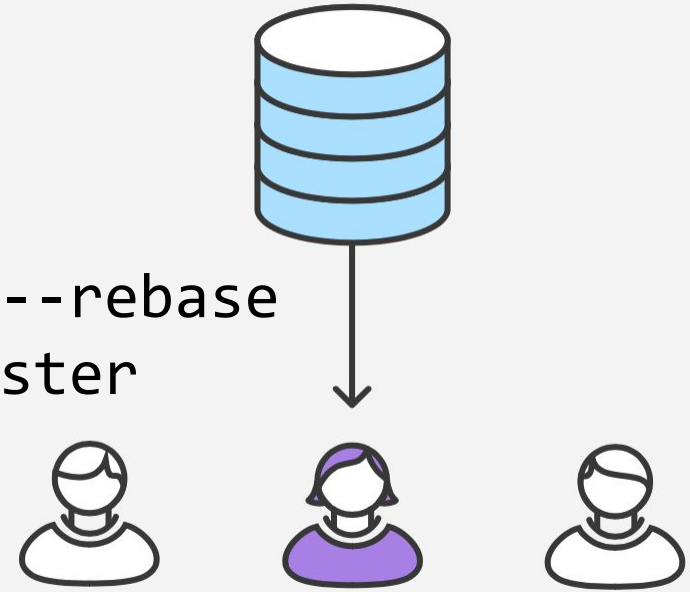


```
error: failed to push some refs to '/path/to/repo.git’
hint: Updates were rejected because the tip of your current branch is behind its
remote counterpart. Merge the remote changes (e.g. 'git pull') before pushing again.
See the 'Note about fast-forwards' in 'git push --help' for details.
```

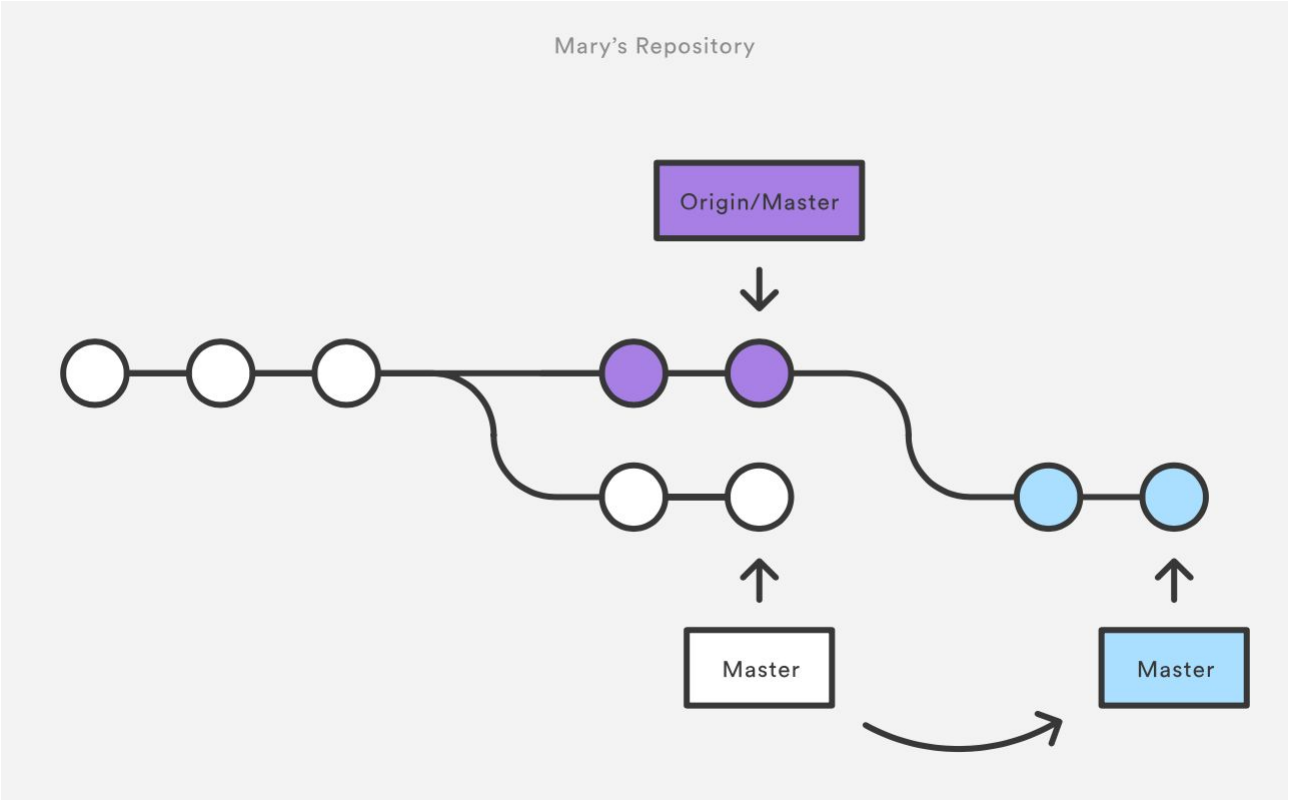


Example

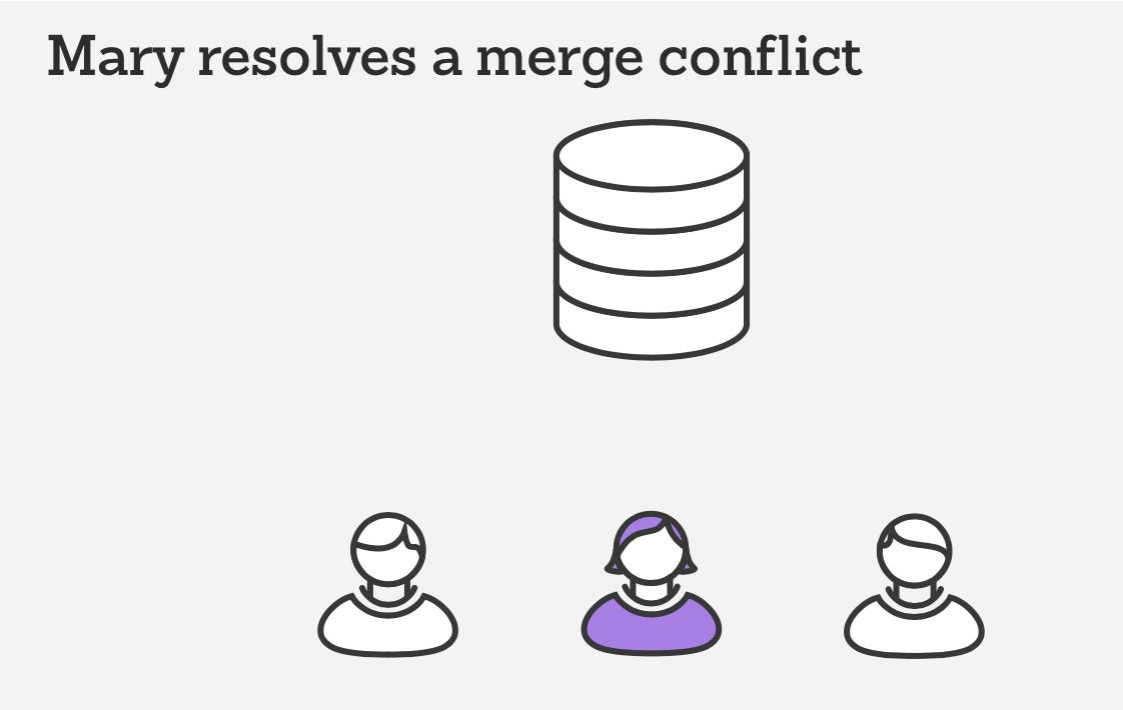
Mary rebases on top of John's commit(s)



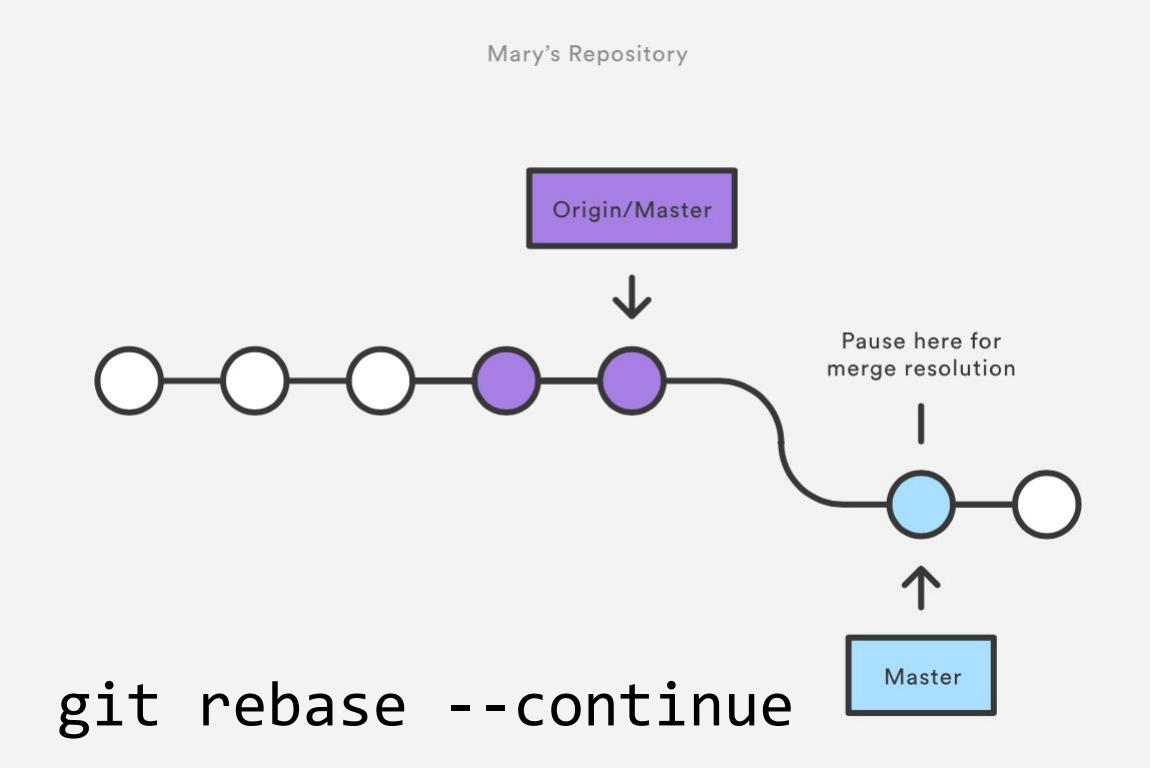
```
git pull --rebase
origin master
```


Mary's Repository
Origin/Master
Master
Master

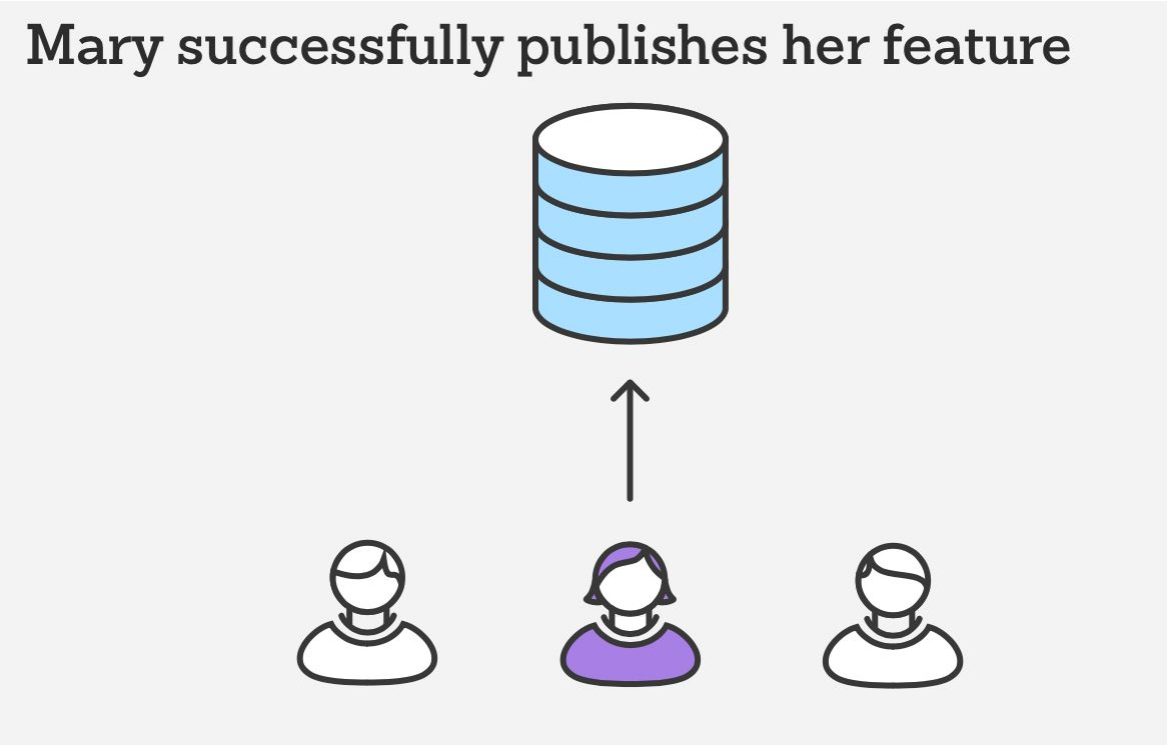
Example



Example


Mary's Repository
Origin/Master
Pause here for
merge resolution
Master
git rebase --continue

Example



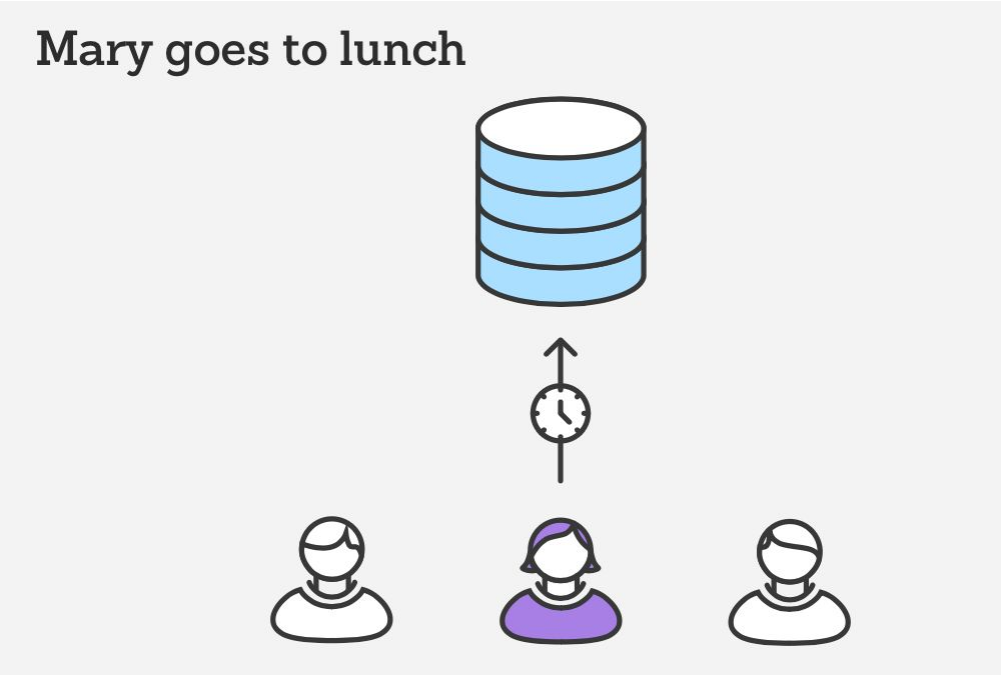
2. Git Feature Branch Workflow

- *All* feature development should take place in a dedicated branch instead of the main branch
- Multiple developers can work on a particular feature without disturbing the main codebase
  - main branch will never contain broken code (enables CI)
  - Enables pull requests (code review)

Example

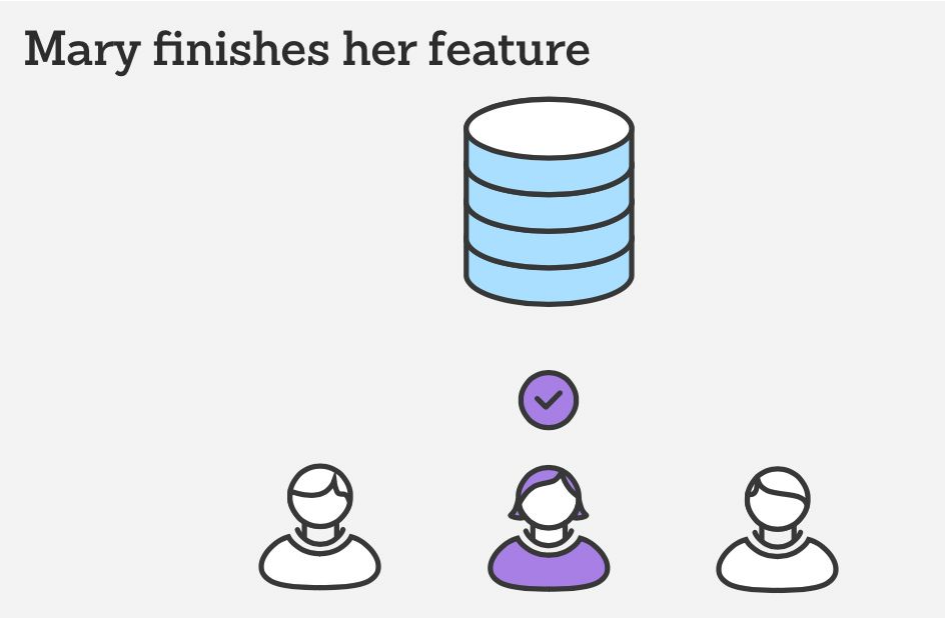

```
git checkout -b marys-feature master
git status
git add <some-file>
git commit
```

Example



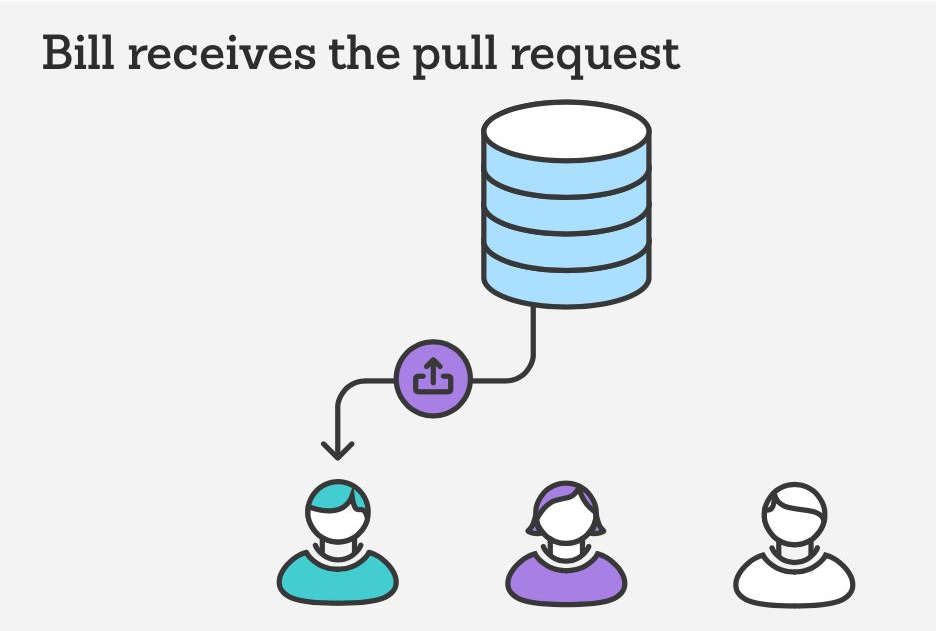

```
git push -u origin marys-feature
```

Example

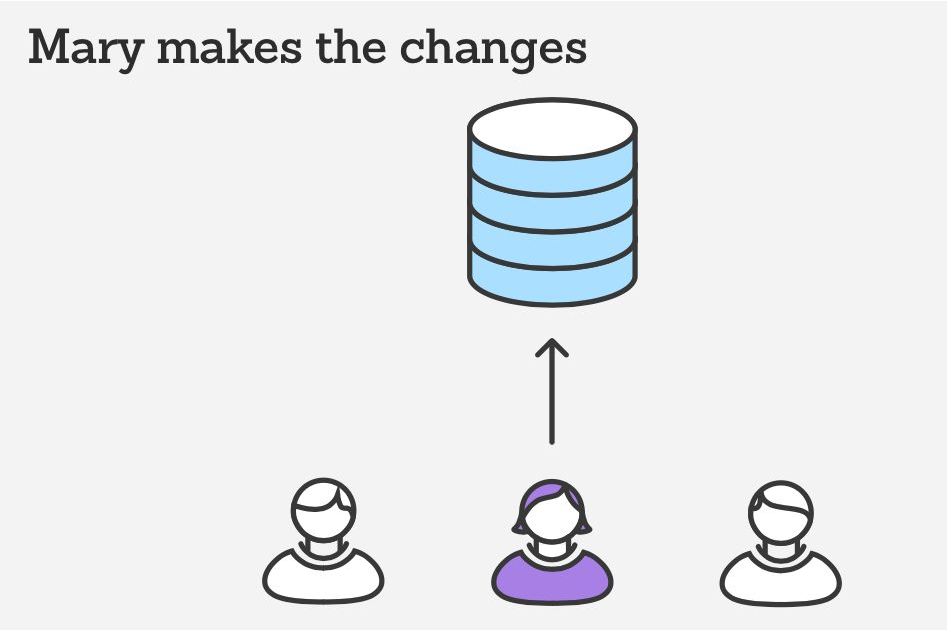



```
git push
```

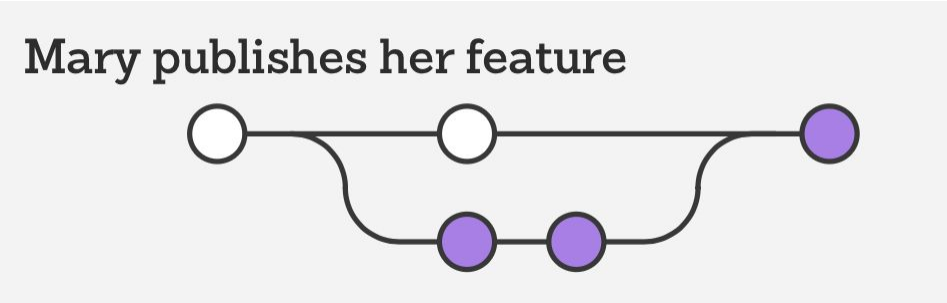
Example



Example

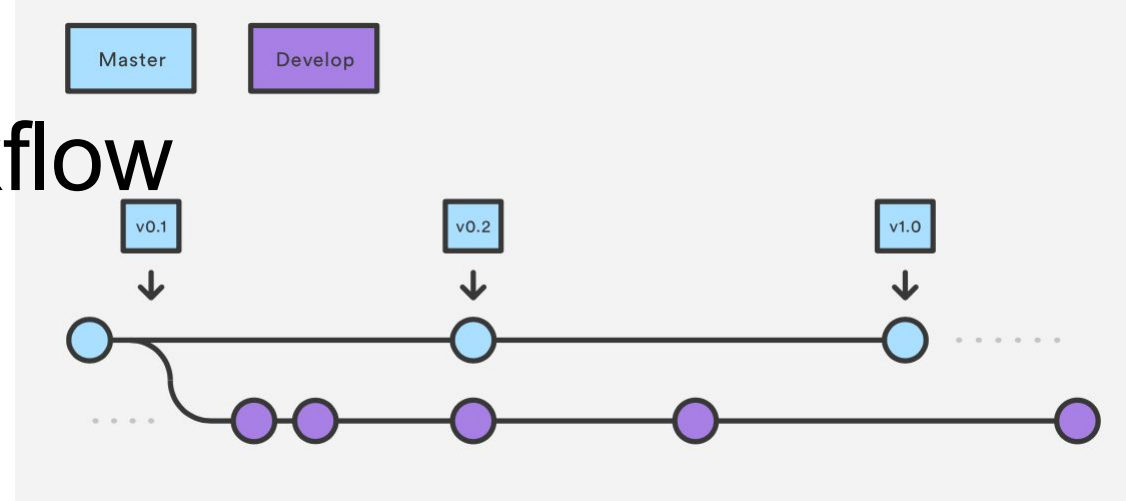


Example - Merge pull request



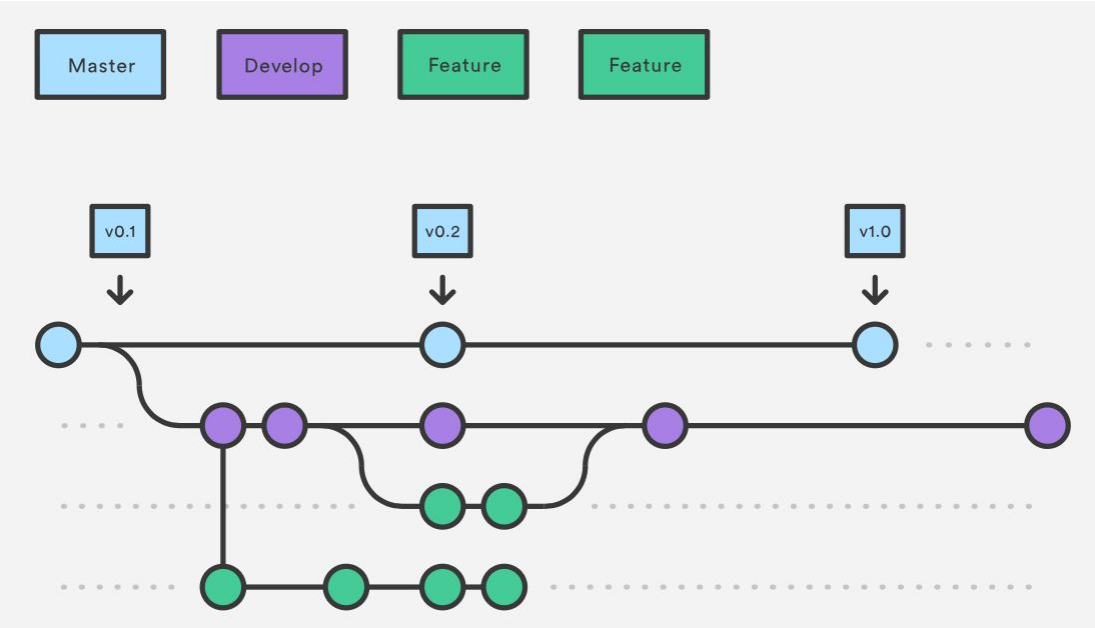

```
git checkout master
git pull
git pull origin marys-feature
git push
```

3. GitFlow Workflow

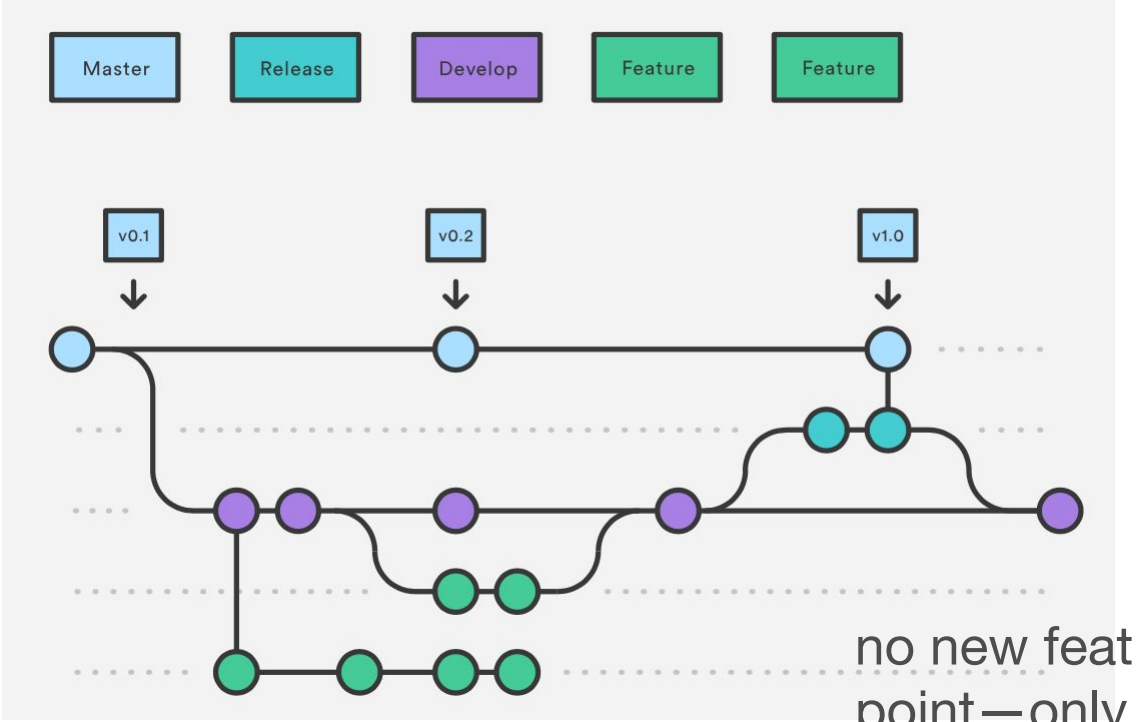


- Strict branching model designed around the project release
  - Suitable for projects that have a scheduled release cycle
- Branches have specific roles and interactions
- Uses two branches
  - main stores the official release history; tag all commits in the main branch with a version number
  - dev(elop) serves as an integration branch for features

GitFlow feature branches (from develop)

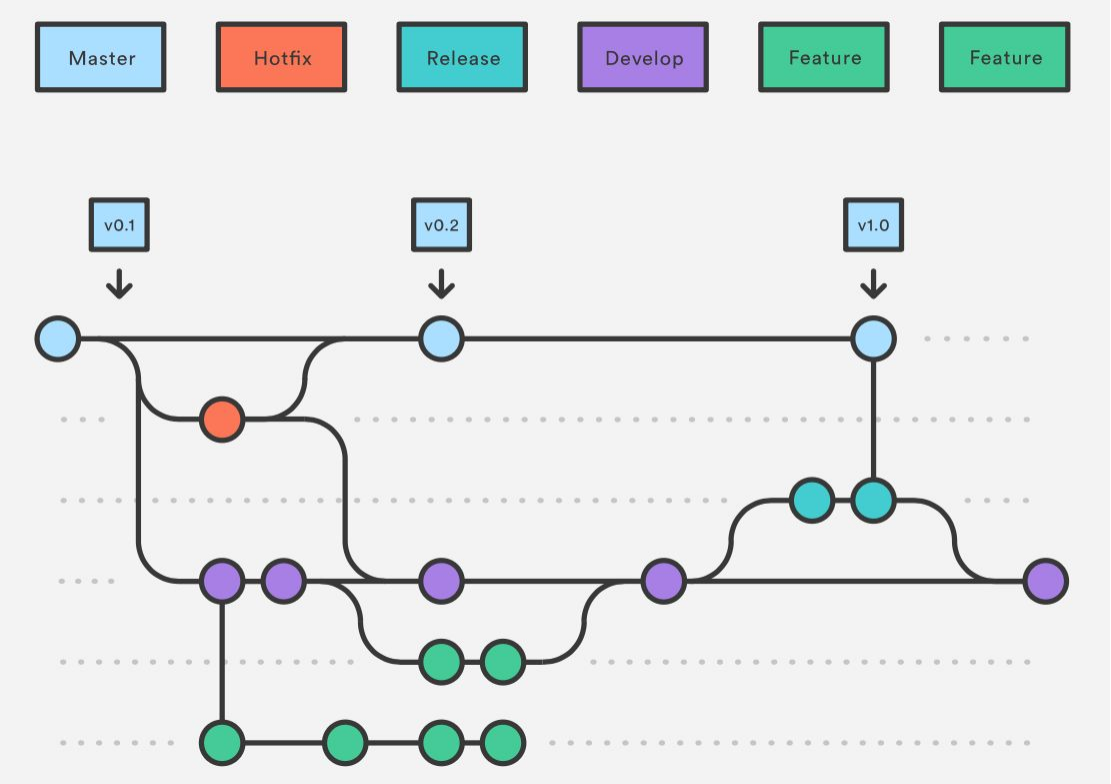


GitFlow release branches (eventually into master)



no new features after this point—only bug fixes, docs, and other release tasks

GitFlow hotfix branches



used to quickly patch production releases

Aside: Semantic Versioning

Semantic Versioning

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

Code status	Stage	Rule	Example version
First release	New product	Start with 1.0.0	1.0.0
Backward compatible bug fixes	Patch release	Increment the third digit	1.0.1
Backward compatible new features	Minor release	Increment the middle digit and reset last digit to zero	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit and reset middle and last digits to zero	2.0.0

https://docs.npmjs.com/about-semantic-versioning

Summary so far

- Version control has many advantages
  - History, traceability, versioning
  - Collaborative and parallel development
- Collaboration with branches
  - Different workflows
- From local to central to distributed version control

DEVELOPMENT AT SCALE

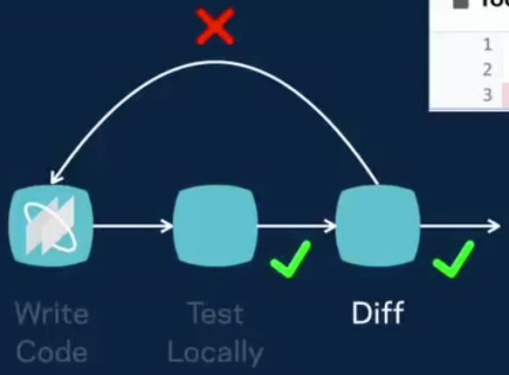
Releasing at scale in industry

- Facebook: https://atscaleconference.com/videos/rapid-release-at-massive-scale/
- Google: https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scal

  https://testing.googleblog.com/2011/06/testing-at-speed-and-scale-of-google.html
- Why Google Stores Billions of Lines of Code in a Single Repository: https://www.youtube.com/watch?v=W71BTkUbdqE
- F8 2015 - Big Code: Developer Infrastructure at Facebook's Scale: https://www.youtube.com/watch?v=X0VH78ye4yY

Pre-2017 release management model at Facebook

Diff lifecycle: local testing



Tools/xctool/xctool/xctool/Version.m

View Options

```
1  #import "Version.h"                                  1  #import "Version.h"
2                                                       2
3  NSString * const XCToolVersionString = @"0.2.1";    3  NSString * const XCToolVersionString = @"0.2.2";
```

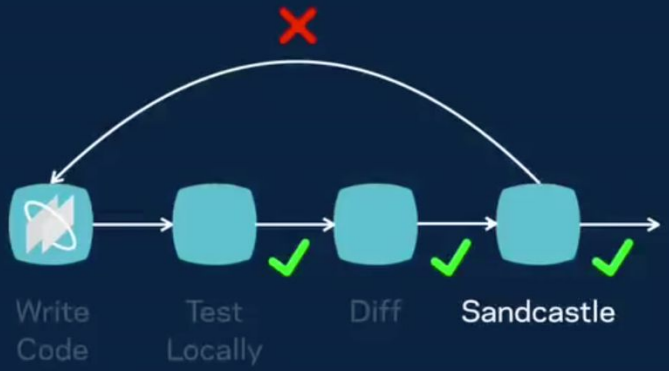
```
PASS ExampleTest (0.050s)
.

OK (1 test, 4 assertions)

OK
(1 tests, 4 assertions, 0 incomplete, 0 failures)
```

Test and lint locally

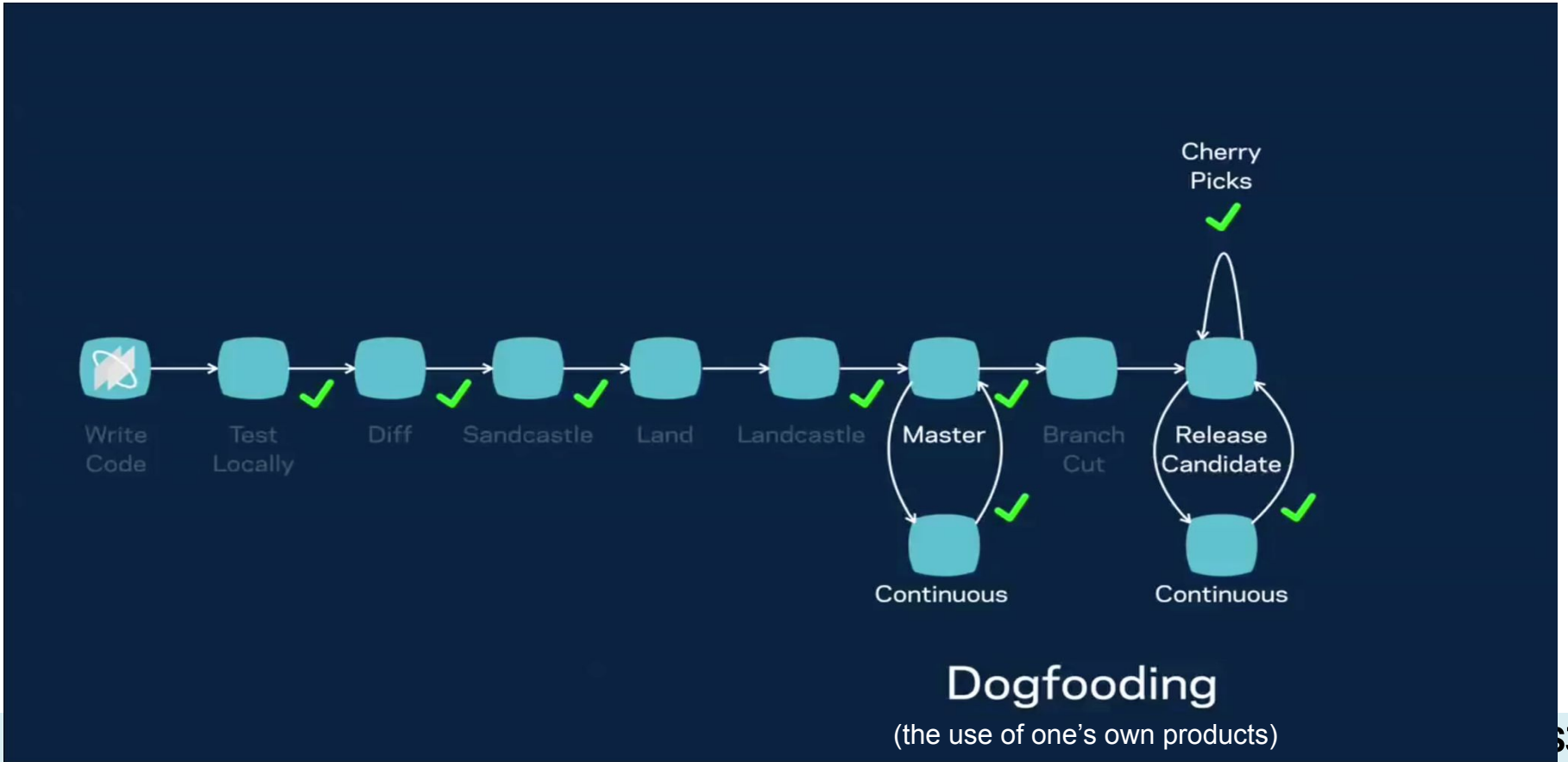
Diff lifecycle: CI testing (data center)



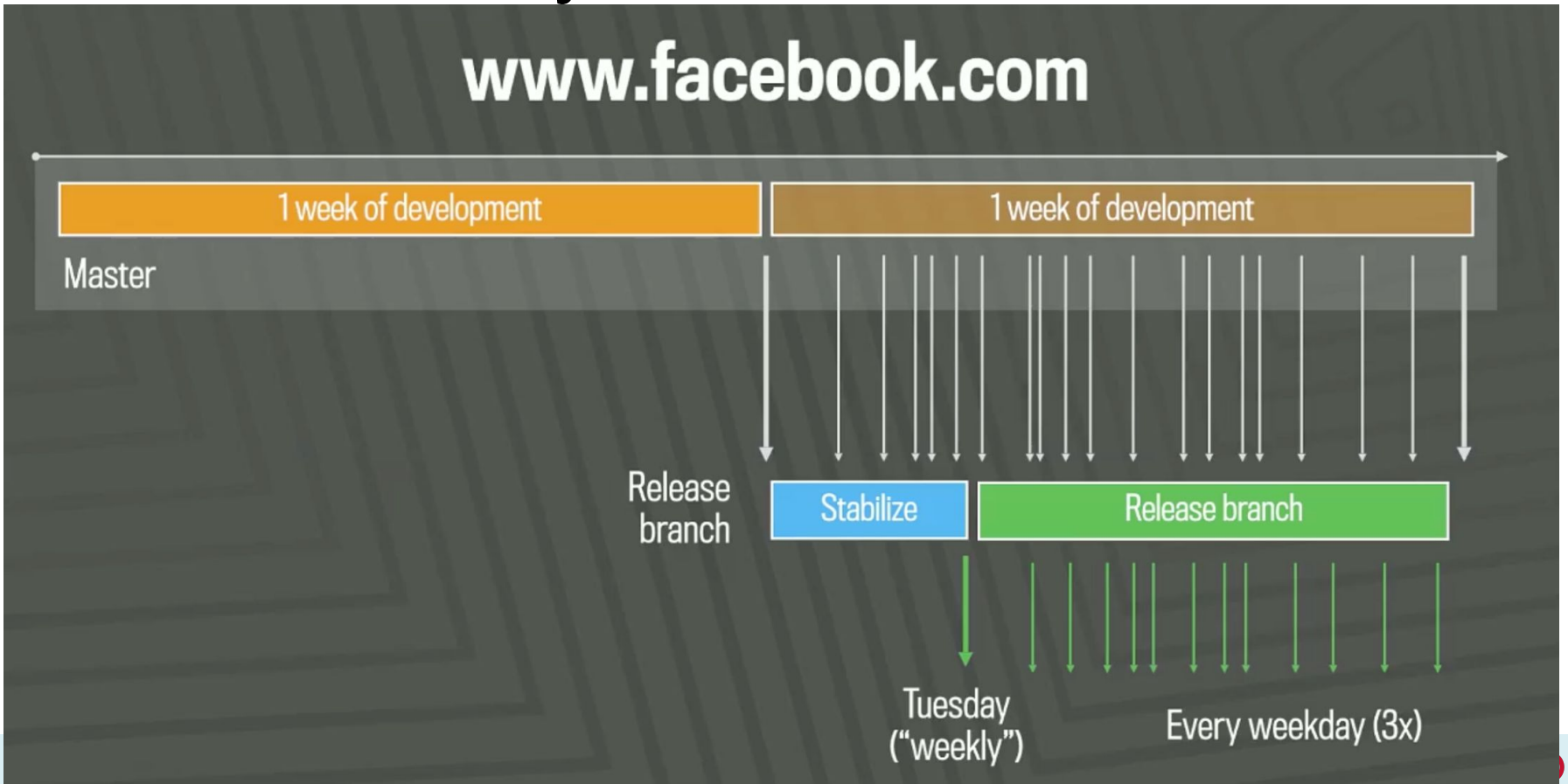
	Facebook	Messenger	Groups	...
arm	✓	✓	✓	✓
x86	✓	✓	✓	✓
...	✓	✓	✓	✓

App and Build Configuration Matrix

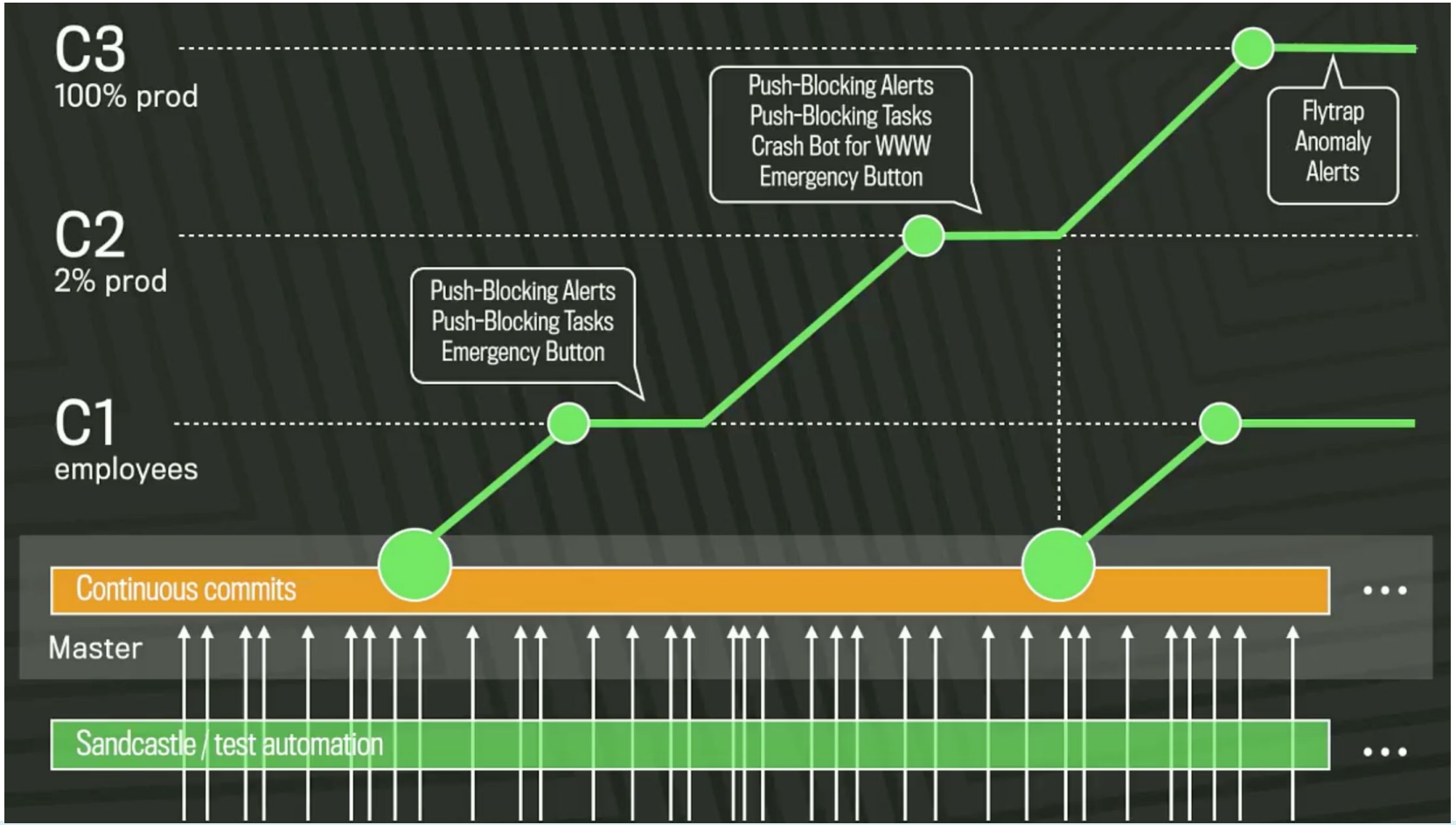
Diff lifecycle: diff ends up on main branch



Release every two weeks



Quasi-continuous push from master (1,000+ devs, 1,000 diffs/day); 10 pushes/day



https://samritchie.wordpress.com/2013/10/16/build-server-traffic-lights/

https://www.softwire.com/blog/2013/09/26/continuous-integration-traffic-lights-revamp/index.html

You've Probably Seen These

Status

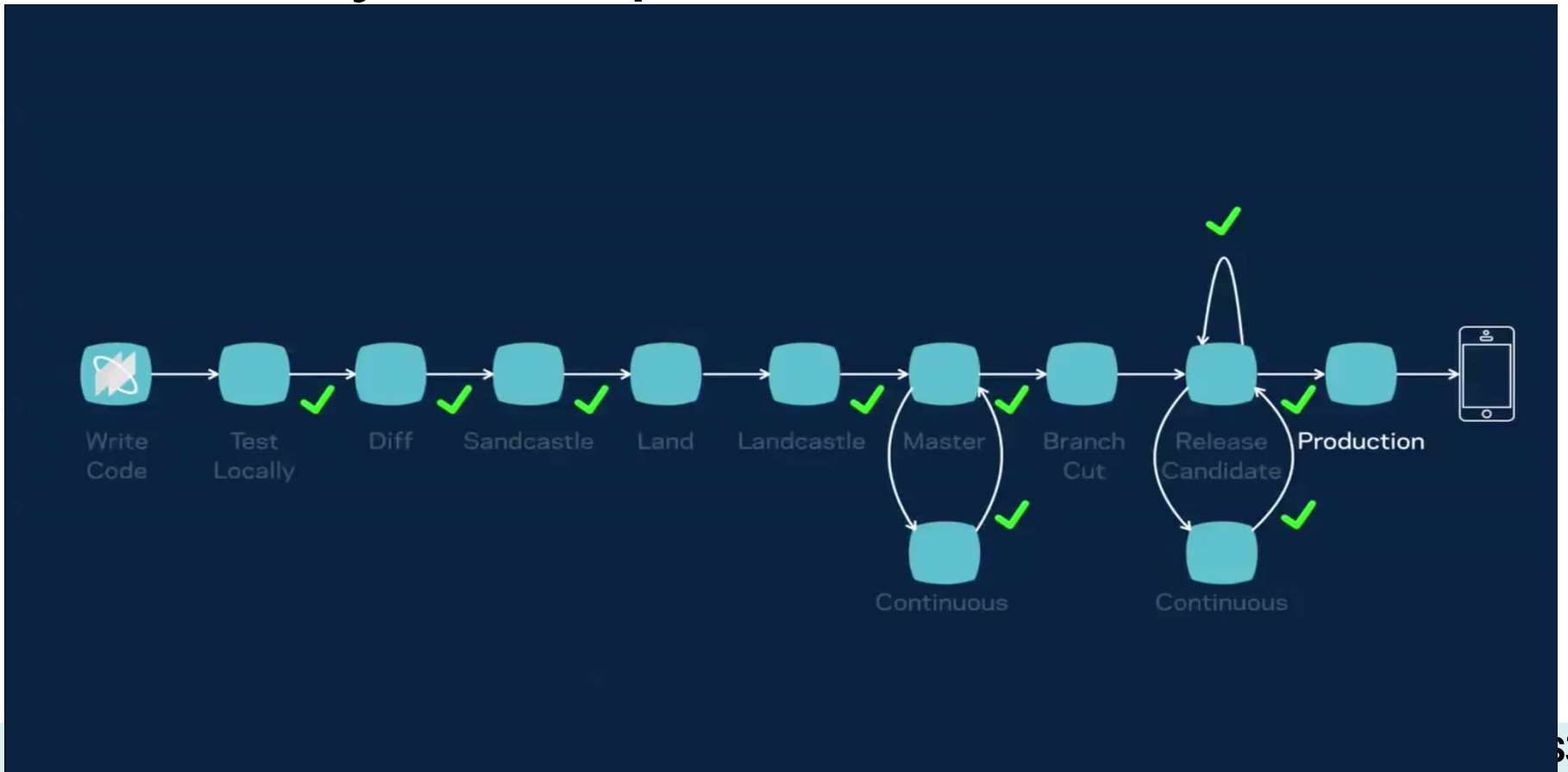
Build Pipeline

Release Pipeline

Dev	Test	Prod
		
		

https://blog.devops4me.com/status-badges-in-azure-devops-pipelines/

Diff lifecycle: in production



Google: similar story. Giant code base

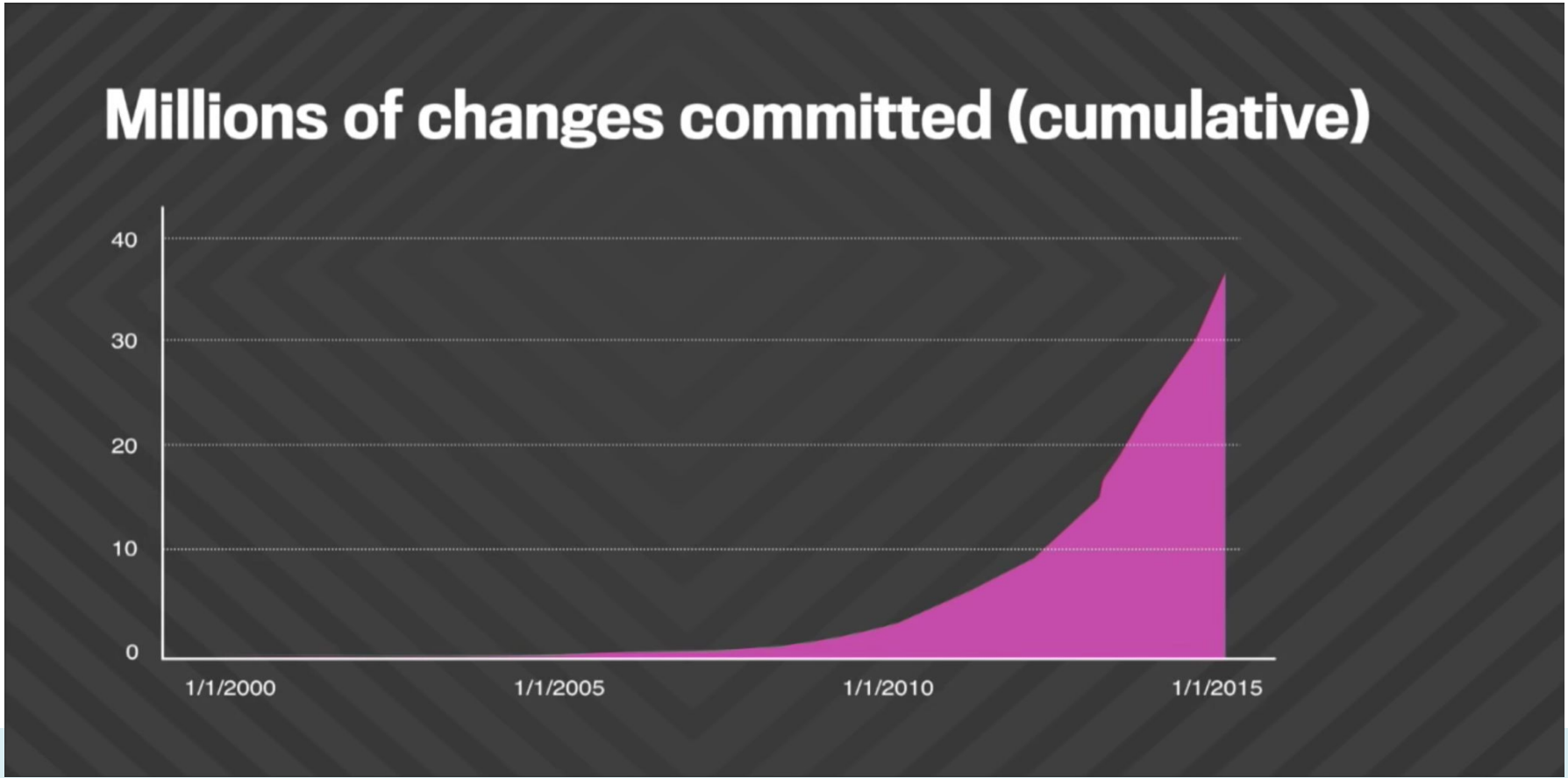
Google repository statistics

As of Jan 2015

Total number of files*	1 billion
Number of source files	9 million
Lines of code	2 billion
Depth of history	35 million commits
Size of content	86 terabytes
Commits per workday	45 thousand

*The total number of files includes source files copied into release branches, files that are deleted at the latest revision, configuration files, documentation, and supporting data files.

Exponential growth



Google Speed and Scale

- >30,000 developers in 40+ offices
- 13,000+ projects under active development
- 30k submissions per day (1 every 3 seconds)
- Single monolithic code tree with mixed language code
- Development on one branch - submissions at head
- All builds from source
- 30+ sustained code changes per minute with 90+ peaks
- 50% of code changes monthly
- 150+ million test cases / day, > 150 years of test / day
- Supports continuous deployment for all Google teams!

2016 numbers

Google code base vs Linux kernel code base

Some perspective

Linux kernel

- 15 million lines of code in 40 thousand files (total)

Google repository

- 15 million lines of code in 250 thousand files *changed per week, by humans*
- 2 billion lines of code, in 9 million source files (total)

How do they do it?

Automation & Processes

1. Lots of (automated) testing

Google workflow

- All code is reviewed before commit (by humans and automated tooling)
- Each directory has a set of owners who must approve the change to their area of the repository
- Tests and automated checks are performed before and after commit
- Auto-rollback of a commit may occur in the case of widespread breakage

2. Lots of automation

Additional tooling support

Now also: language model-based completions:
https://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html

Critique	Code review
CodeSearch*	Code browsing, exploration, understanding, and archeology
Tricorder**	Static analysis of code surfaced in Critique, CodeSearch
Presubmits	Customizable checks, testing, can block commit
TAP	Comprehensive testing before and after commit, auto-rollback
Rosie	Large-scale change distribution and management

* See "How Developers Search for Code: A Case Study", In European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2015
** See "Tricorder: Building a program analysis ecosystem". In International Conference on Software Engineering (ICSE), 2015

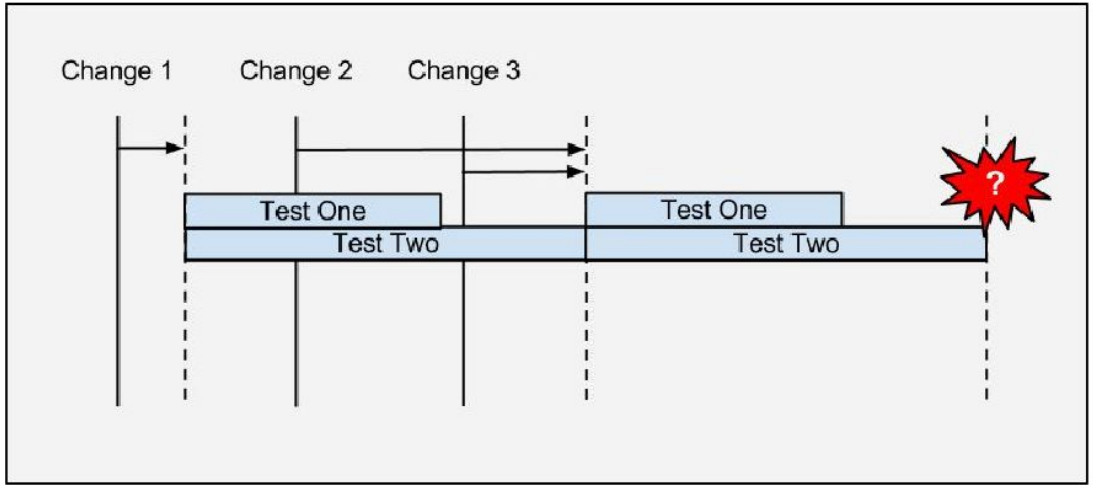
3. Smarter tooling

- Build system
- Version control
- …

3a. Build system

Google Standard Continuous Build System

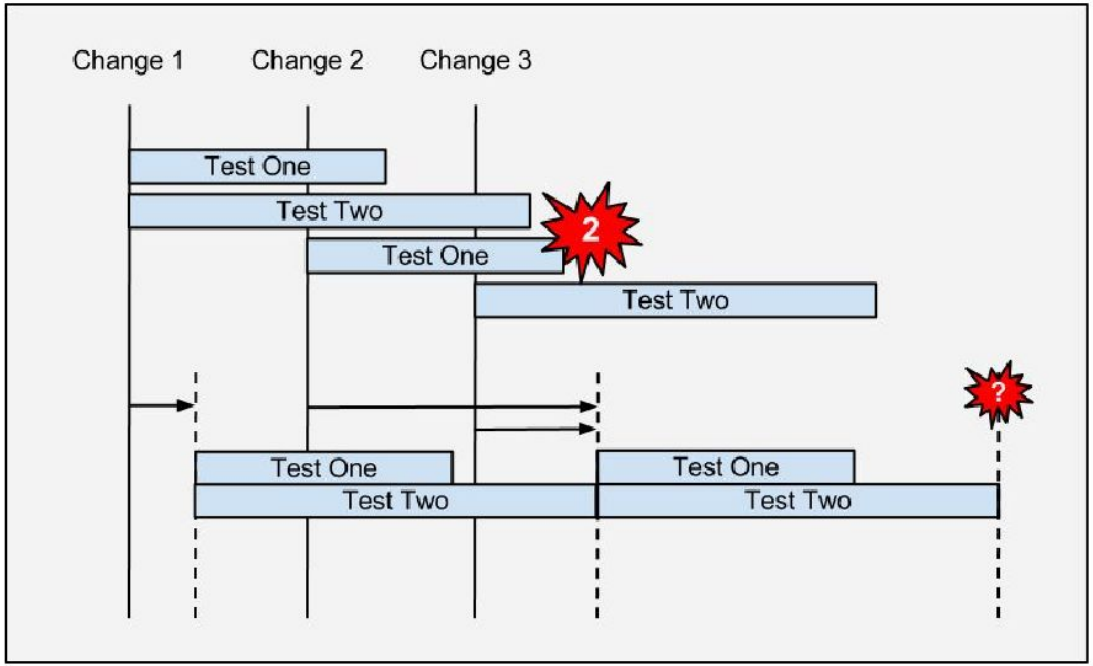
- Triggers builds in continuous cycle
- Cycle time = longest build + test cycle
- Tests many changes together
- Which change broke the build?



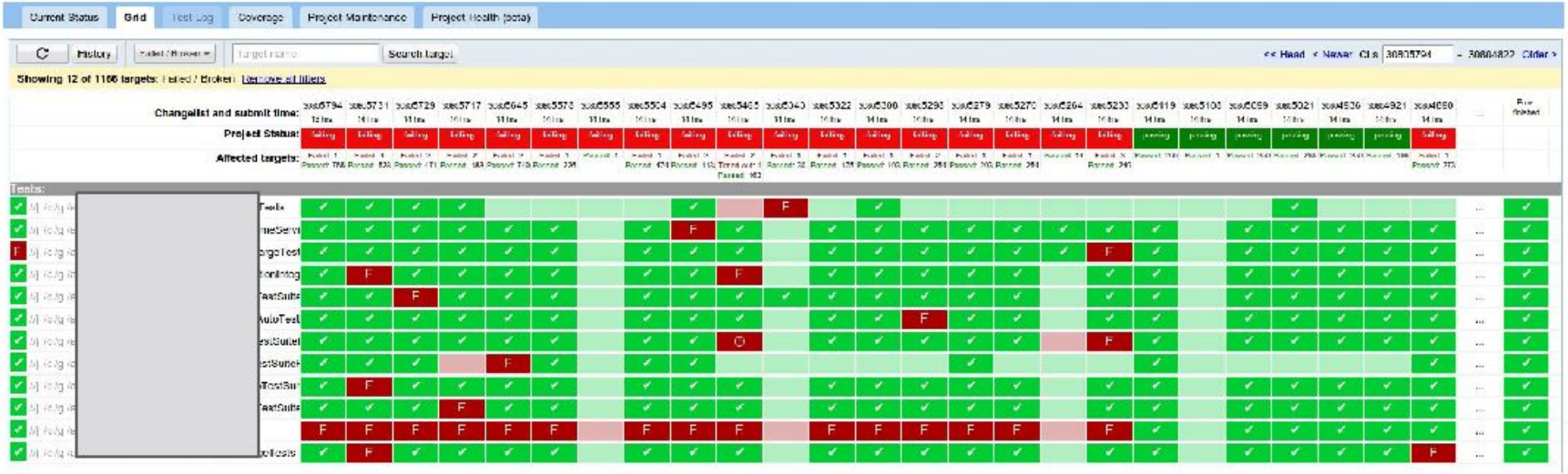
3a. Build system

Google Continuous Build System

- Triggers tests on every change
- Uses fine-grained dependencies
- Change 2 broke test 1



Google Continuous Integration Display



3a. Build system

Google Benefits

- Identifies failures sooner
- Identifies culprit change precisely
  - Avoids divide-and-conquer and tribal knowledge
- Lower compute costs using fine grained dependencies
- Keeps the build green by reducing time to fix breaks
- Accepted enthusiastically by product teams
- Enables teams to ship with fast iteration times
  - Supports submit-to-production times of less than 36 hours for some projects

3a. Build system

Google Costs

- Requires enormous investment in compute resources (it helps to be at Google) grows in proportion to:
  - Submission rate
  - Average build + test time
  - Variants (debug, opt, valgrind, etc.)
  - Increasing dependencies on core libraries
  - Branches
- Requires updating dependencies on each change
  - Takes time to update - delays start of testing

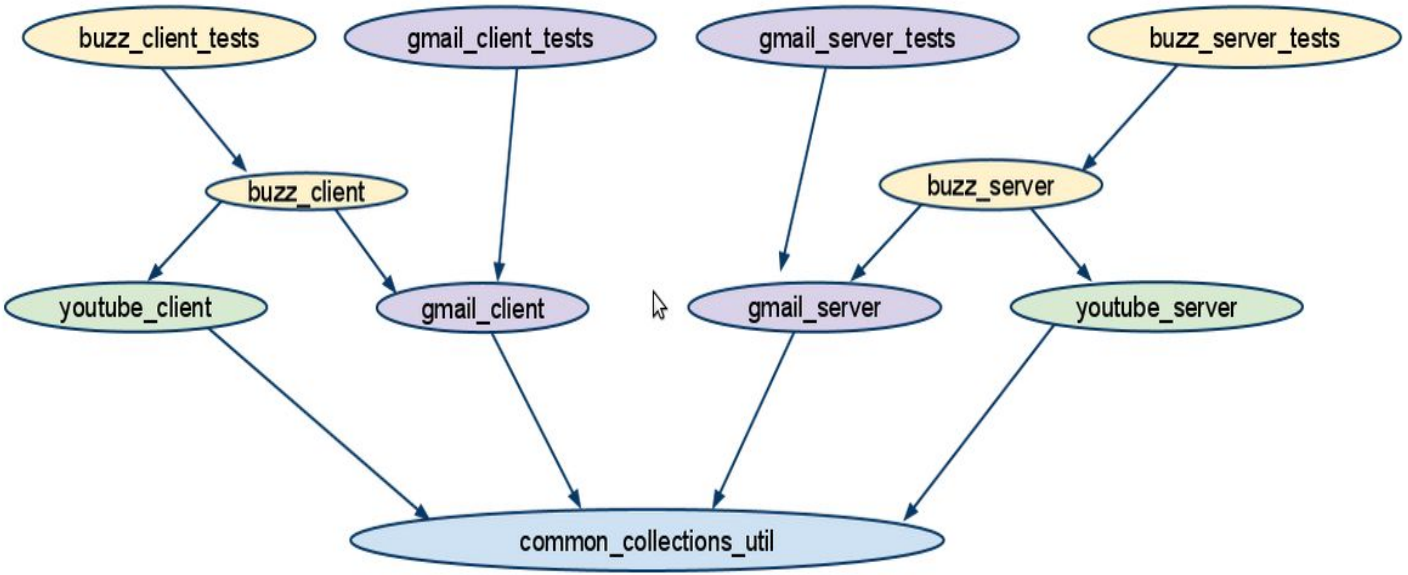
Which tests to run?

GMAIL

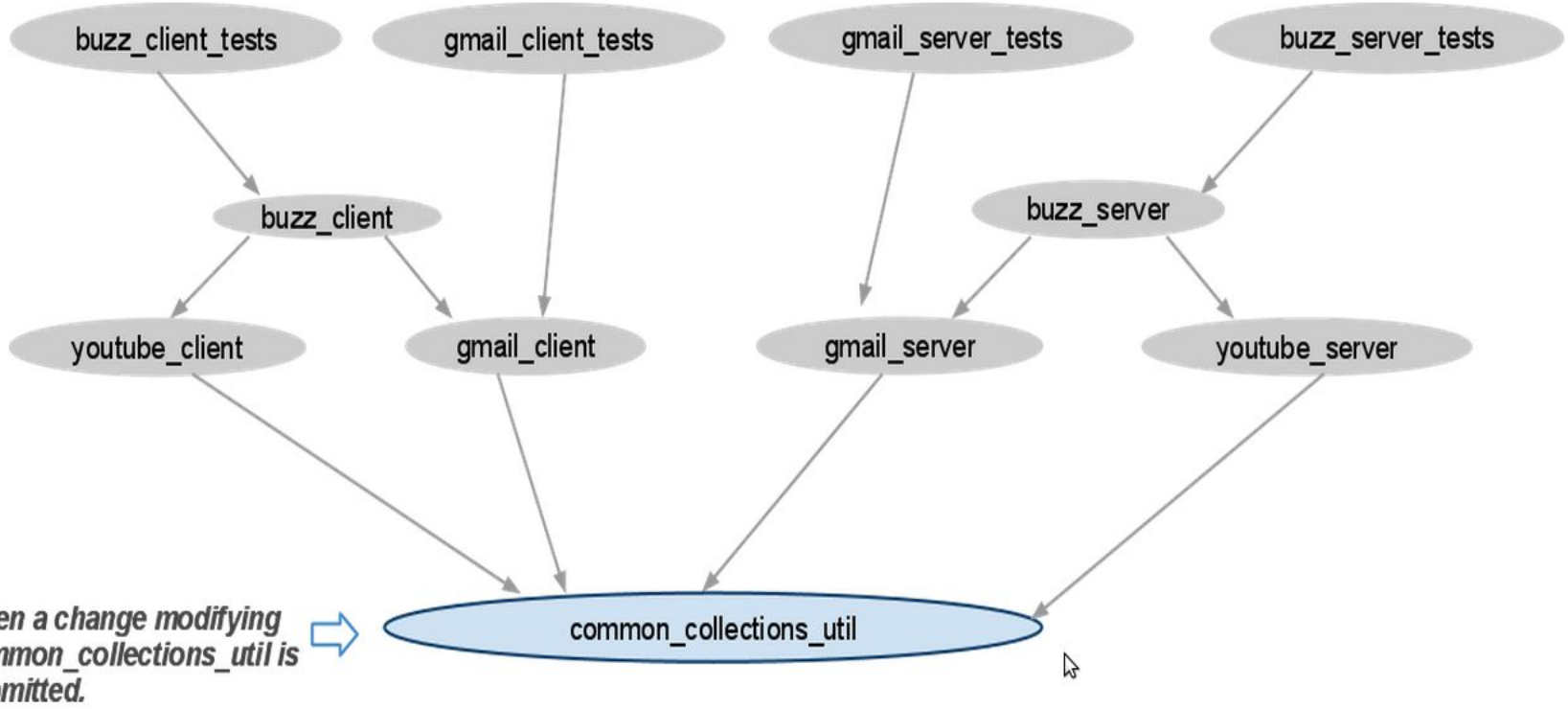
Test Target:
name: //depot/gmail_client_tests
name: //depot/gmail_server_tests

BUZZ

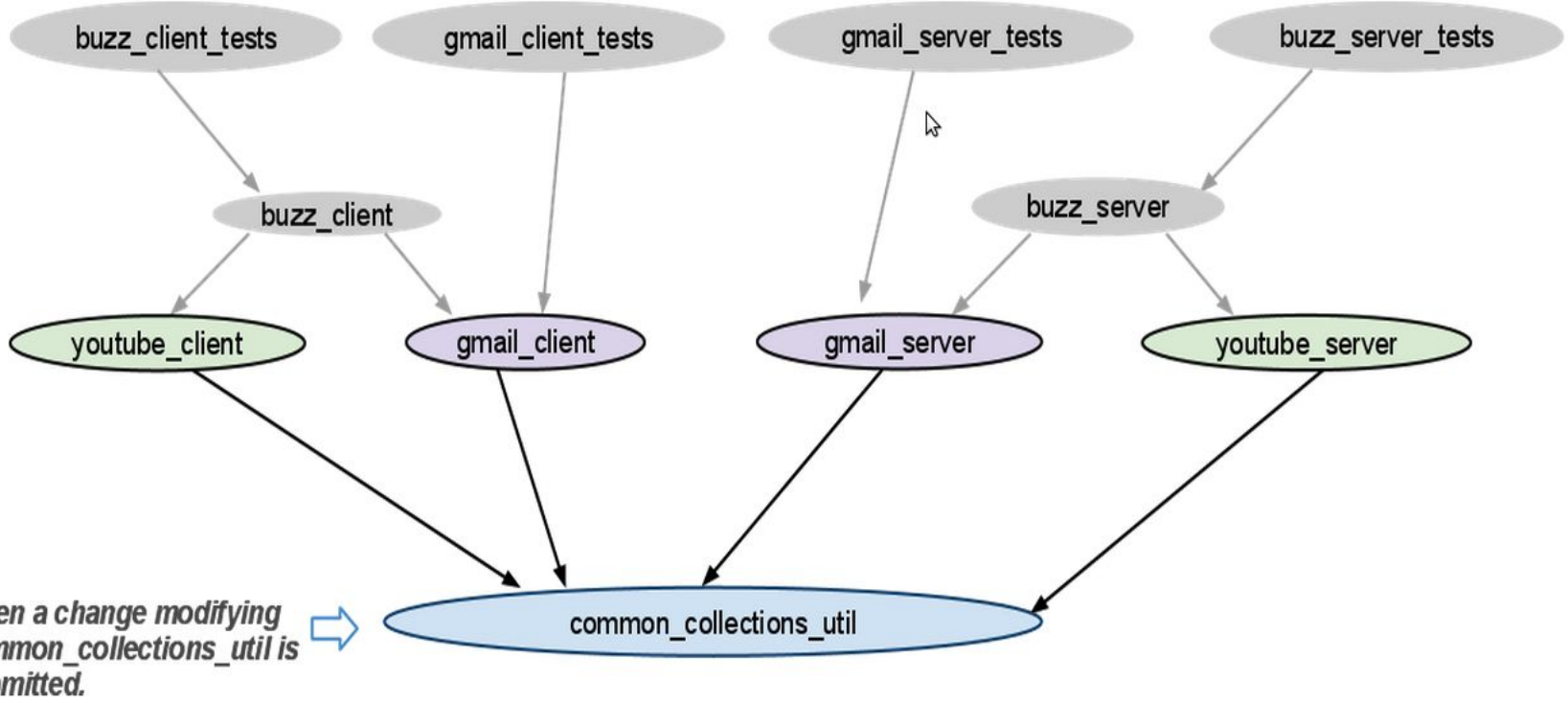
Test targets:
name: //depot/buzz_server_tests
name: //depot/buzz_client_tests



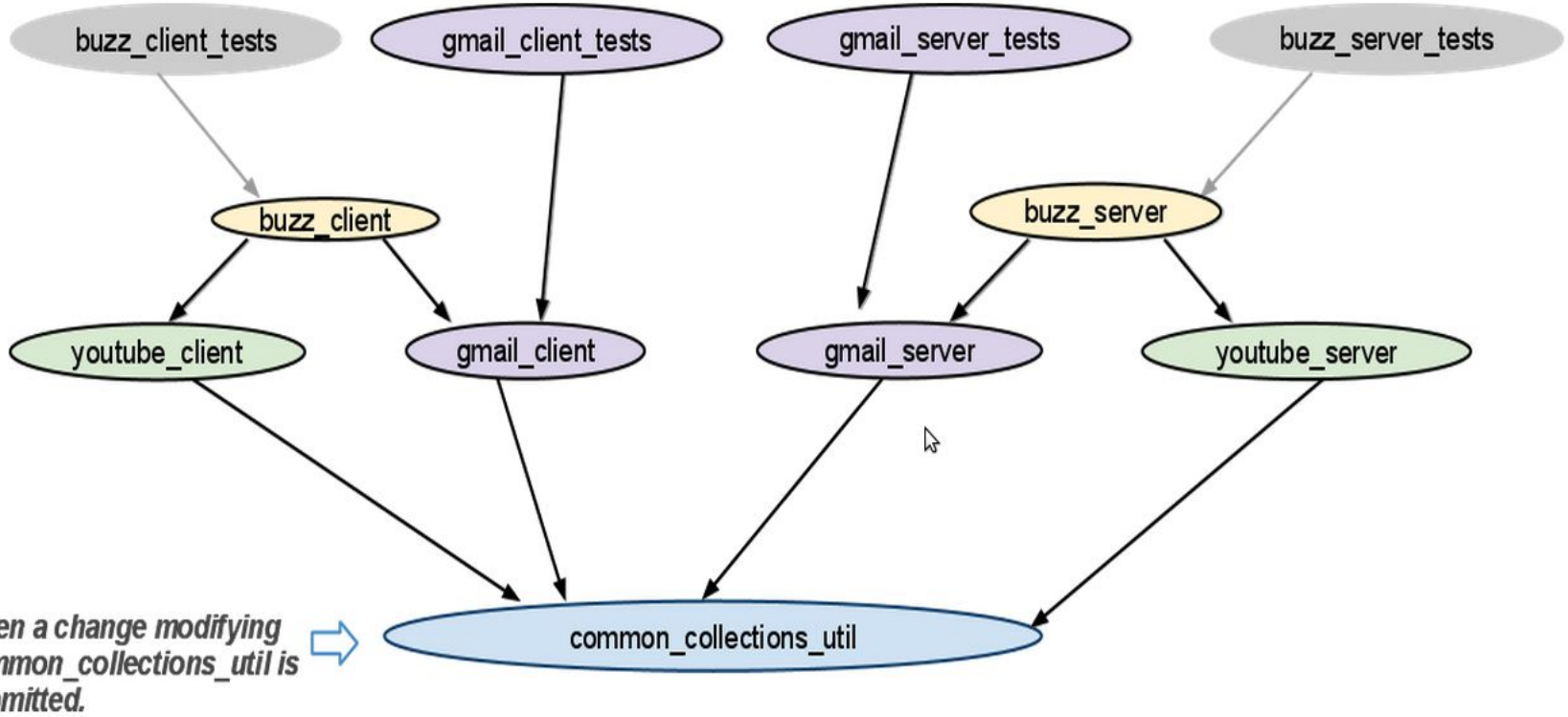
Scenario 1: a change modifies common_collections_util



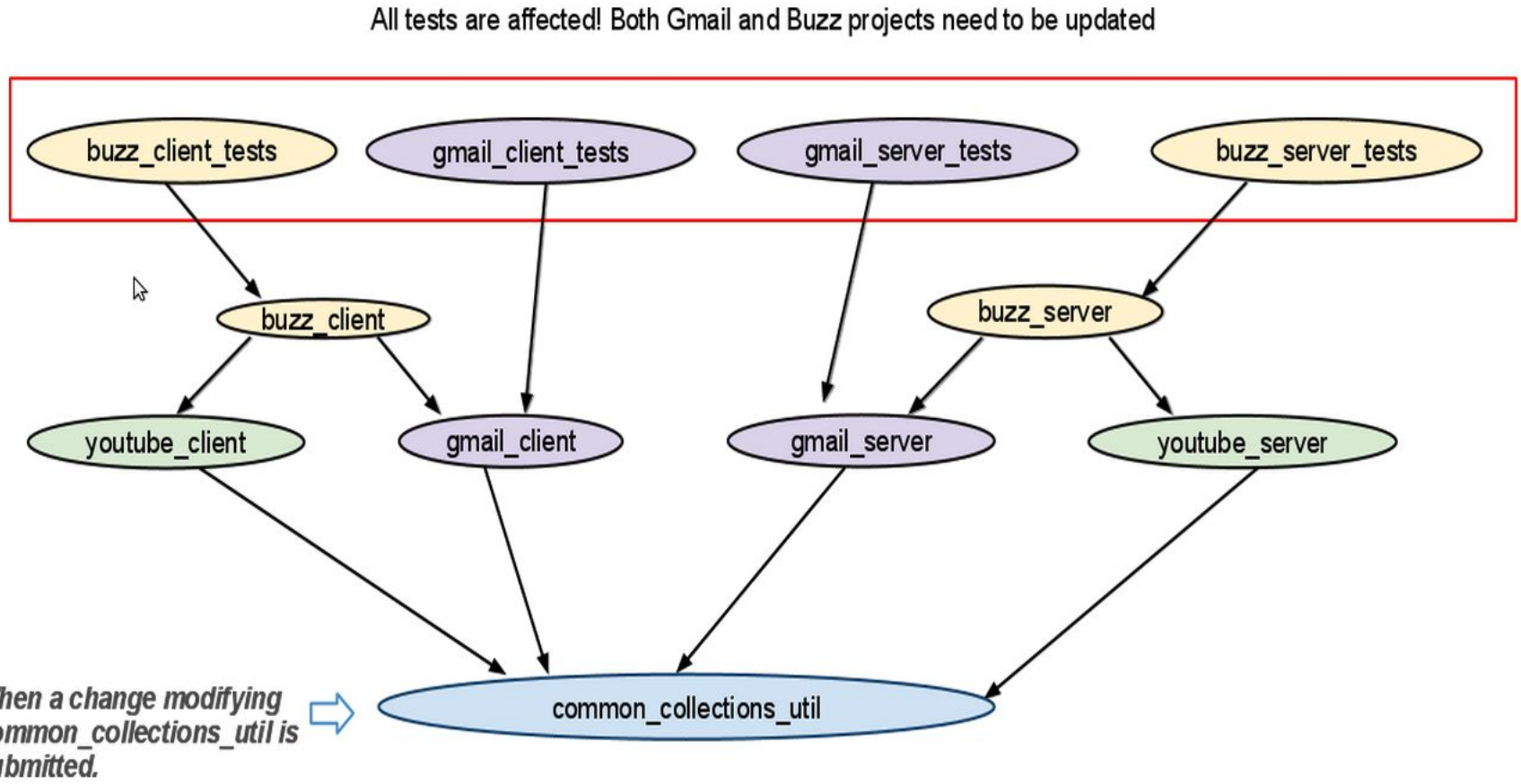
Scenario 1: a change modifies common_collections_util



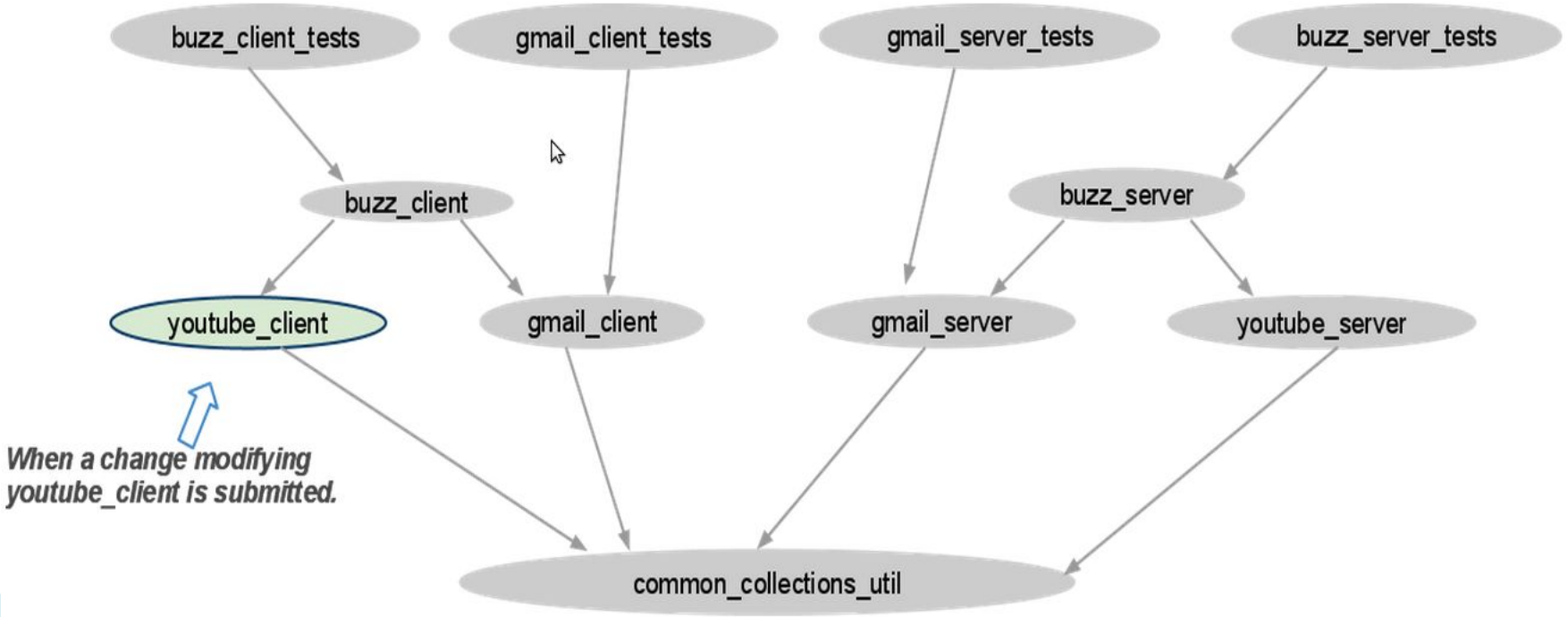
Scenario 1: a change modifies common_collections_util



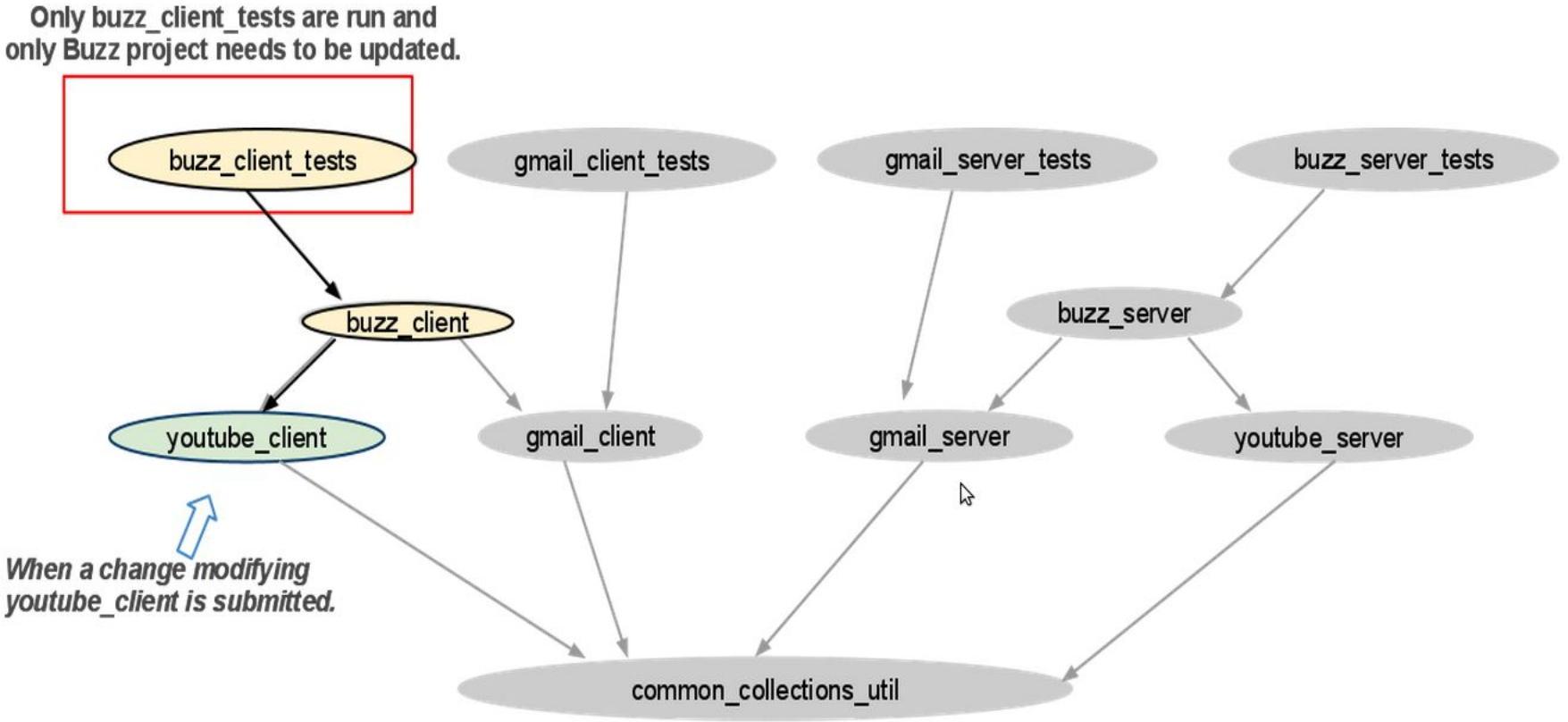
Scenario 1: a change modifies common_collections_util



Scenario 2: a change modifies the youtube_client

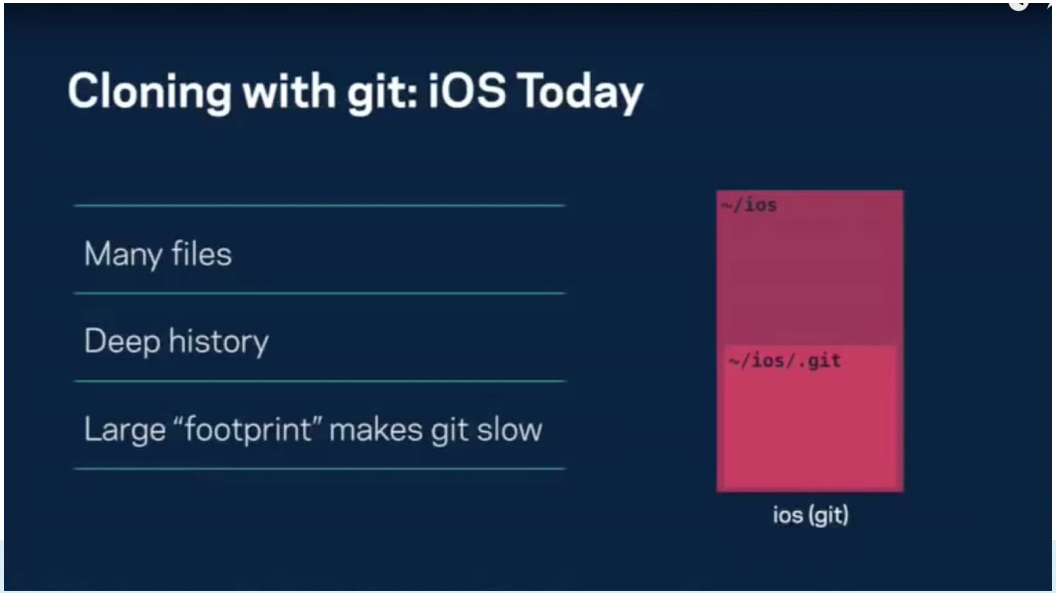


Scenario 2: a change modifies the youtube_clien



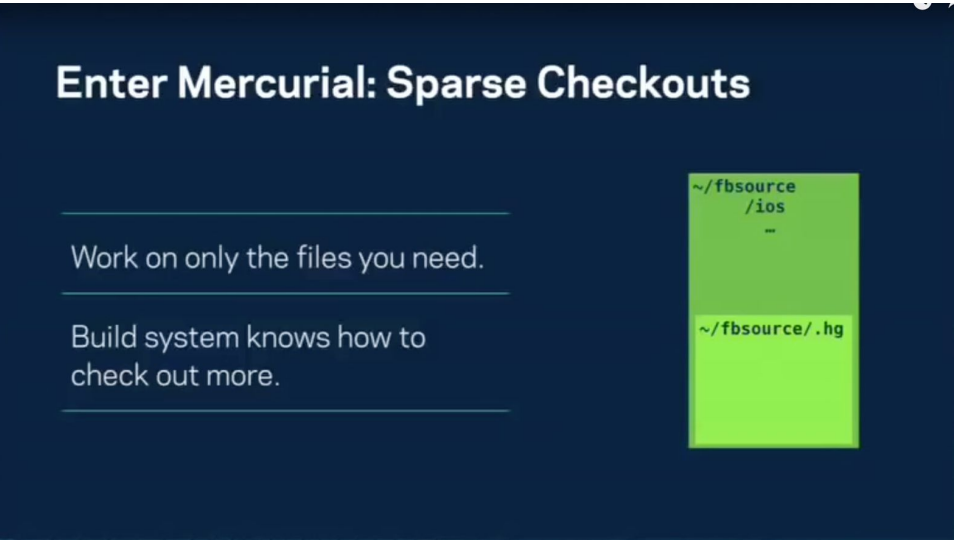
3b. Version control

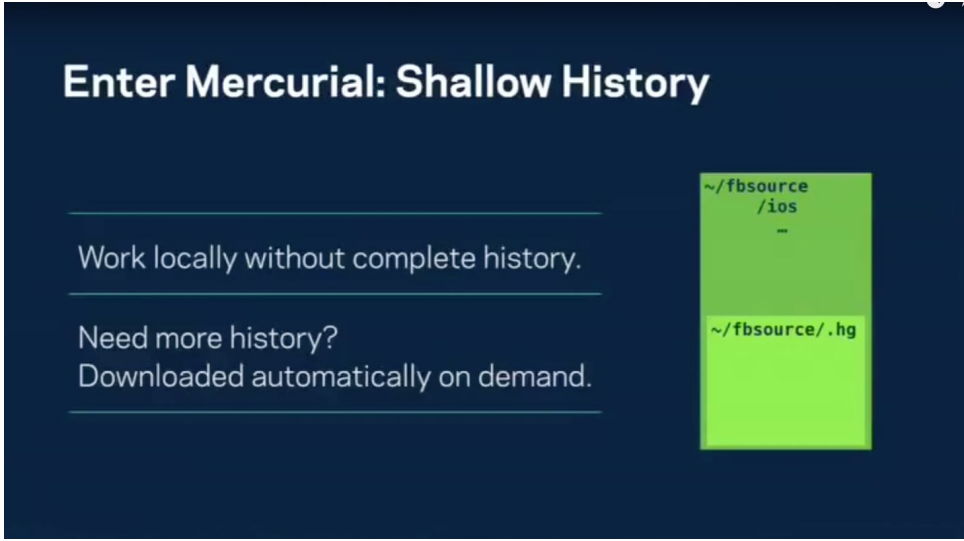
- Problem: even git can get slow at Facebook scale
  - 1M+ source control commands run per day
  - 100K+ commits per week



3b. Version control

- Solution: redesign version control
  - Sparse checkouts: only fetch metadata (lightweight), get source on-demand
  - Don’t fetch entire history. Can do this with git too (git clone --depth=1), but won’t work for distributed collaboration





Some Common Principles

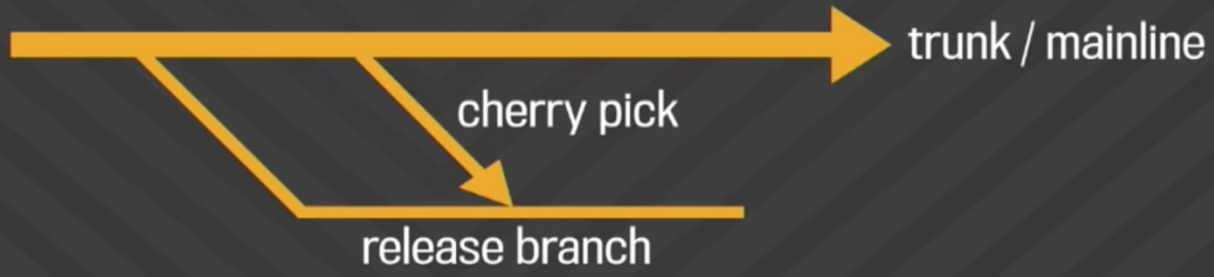
- Ensure Isolation
  - Of impacts of a given changeset
    - On the build status
    - On production code
  - Not dissimilar to distributed systems!
    - Which makes sense; this is also a distributed system, just made up of people
- Work incrementally
  - Release carefully, monitor heavily
  - Cut costs where possible by building & testing as little as possible

Monolithic repository – no major use of branches for development

Trunk-based development

Combined with a centralized repository, this defines the monolithic model

- Piper users work at “head”, a consistent view of the codebase
- All changes are made to the repository in a single, serial ordering
- There is no significant use of branching for development
- Release branches are cut from a specific revision of the repository



A recent history of code organization

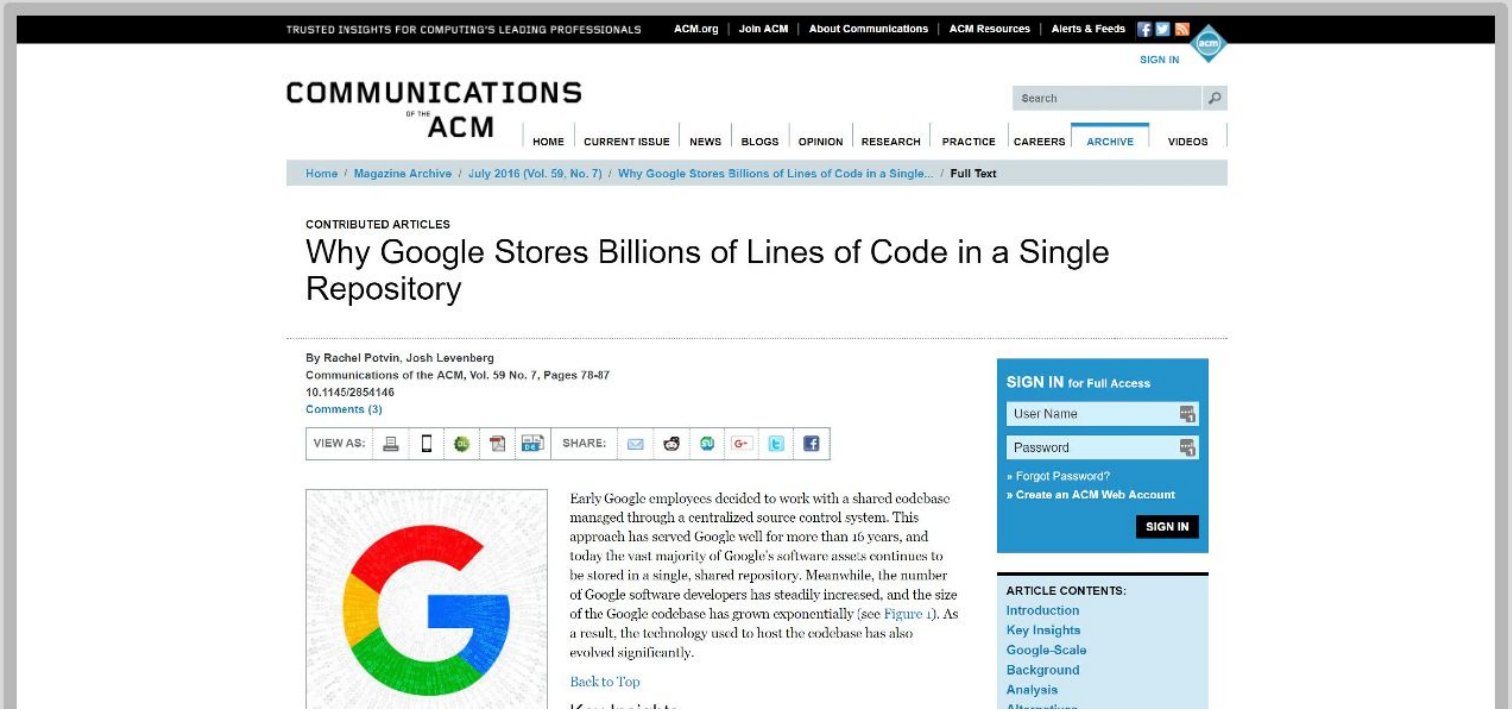
- A single team with a monolithic application in a single repository
- …
- Multiple teams with many separate applications in many separate repositories
- Multiple teams with many ~~separate applications~~ **microservices** in many separate repositories
- A single team with many microservices in many repositories
- …
- Many teams with many applications in one big **Monorepo**

What is a monolithic repository (monorepo)?

- A **single** version control repository containing multiple
  - Projects
  - Applications
  - Libraries
- Often using a common build system

Monorepos in industry

Google (computer science version)



TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS ACM.org | Join ACM | About Communications | ACM Resources | Alerts & Feeds

SIGN IN

COMMUNICATIONS OF THE ACM

Search

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRACTICE | CAREERS | ARCHIVE | VIDEOS

Home / Magazine Archive / July 2016 (Vol. 59, No. 7) / Why Google Stores Billions of Lines of Code in a Single... / Full Text

CONTRIBUTED ARTICLES

Why Google Stores Billions of Lines of Code in a Single Repository

By Rachel Potvin, Josh Levenberg
Communications of the ACM, Vol. 59 No. 7, Pages 78-87
10.1145/2854146
Comments (3)

VIEW AS: SHARE:

Early Google employees decided to work with a shared codebase managed through a centralized source control system. This approach has served Google well for more than 16 years, and today the vast majority of Google's software assets continues to be stored in a single, shared repository. Meanwhile, the number of Google software developers has steadily increased, and the size of the Google codebase has grown exponentially (see Figure 1). As a result, the technology used to host the codebase has also evolved significantly.

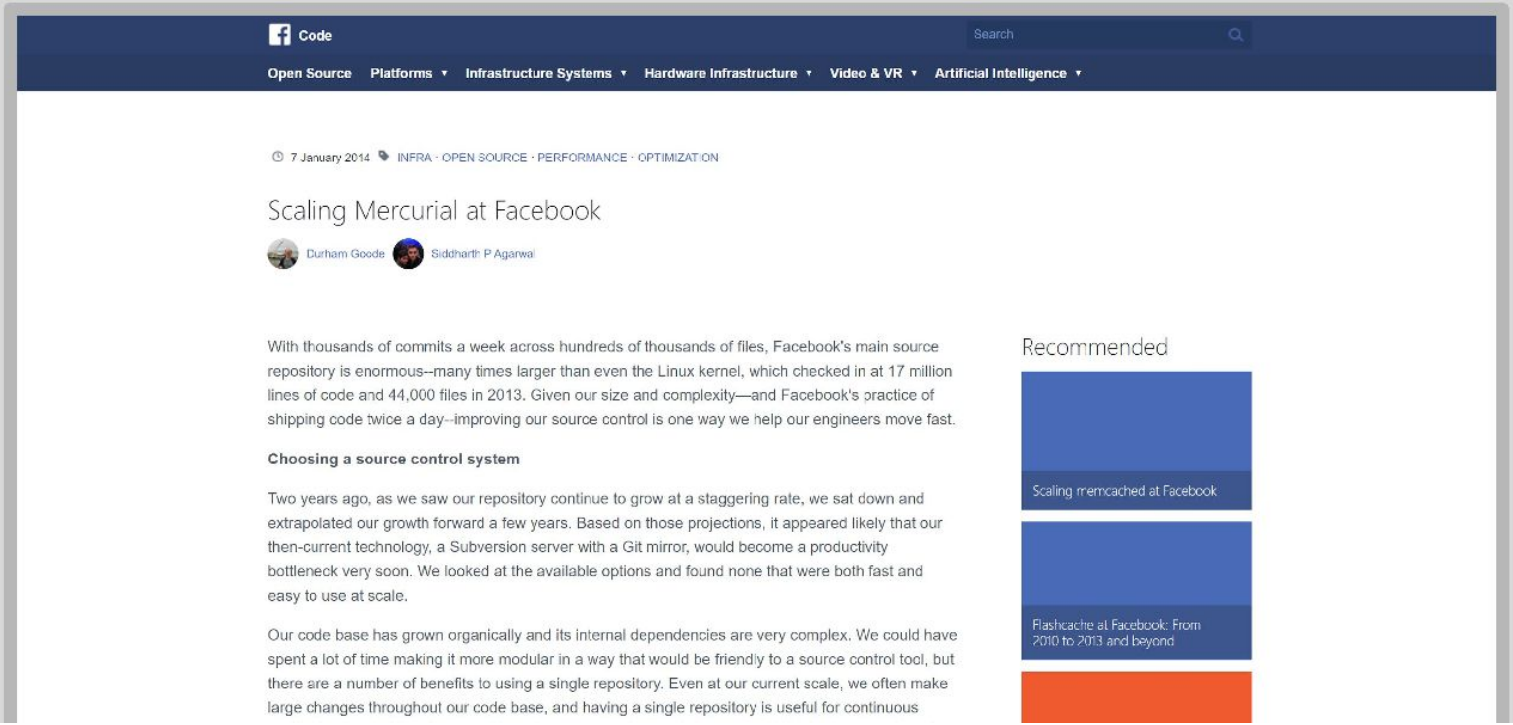
Back to Top

SIGN IN for Full Access
User Name
Password
» Forgot Password?
» Create an ACM Web Account
SIGN IN

ARTICLE CONTENTS:
Introduction
Key Insights
Google-Scale
Background
Analysis

Monorepos in industry

Scaling Mercurial at Facebook



Code

Search

Open Source | Platforms | Infrastructure Systems | Hardware Infrastructure | Video & VR | Artificial Intelligence

7 January 2014 · INFRA · OPEN SOURCE · PERFORMANCE · OPTIMIZATION

Scaling Mercurial at Facebook

Durham Goode · Siddharth P Agarwal

With thousands of commits a week across hundreds of thousands of files, Facebook's main source repository is enormous--many times larger than even the Linux kernel, which checked in at 17 million lines of code and 44,000 files in 2013. Given our size and complexity—and Facebook's practice of shipping code twice a day--improving our source control is one way we help our engineers move fast.

Choosing a source control system

Two years ago, as we saw our repository continue to grow at a staggering rate, we sat down and extrapolated our growth forward a few years. Based on those projections, it appeared likely that our then-current technology, a Subversion server with a Git mirror, would become a productivity bottleneck very soon. We looked at the available options and found none that were both fast and easy to use at scale.

Our code base has grown organically and its internal dependencies are very complex. We could have spent a lot of time making it more modular in a way that would be friendly to a source control tool, but there are a number of benefits to using a single repository. Even at our current scale, we often make large changes throughout our code base, and having a single repository is useful for continuous

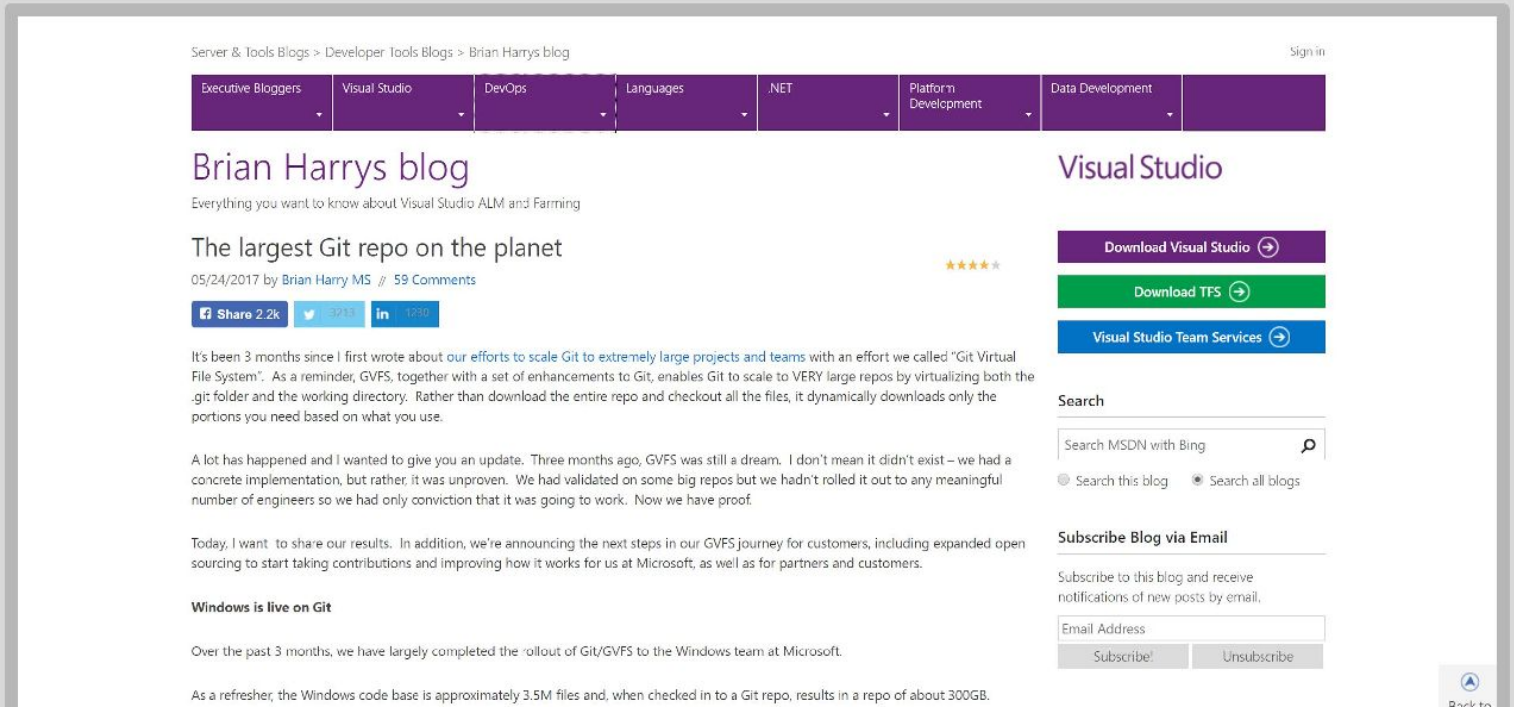
Recommended

Scaling memcached at Facebook

Flashcache at Facebook: From 2010 to 2013 and beyond

Monorepos in industry

Microsoft claim the largest git repo on the planet



Server & Tools Blogs > Developer Tools Blogs > Brian Harrys blog

Sign in

Executive Bloggers | Visual Studio | DevOps | Languages | .NET | Platform Development | Data Development

Brian Harrys blog

Everything you want to know about Visual Studio ALM and Farming

The largest Git repo on the planet

05/24/2017 by Brian Harry MS // 59 Comments

Share 2.2k

It's been 3 months since I first wrote about our efforts to scale Git to extremely large projects and teams with an effort we called "Git Virtual File System". As a reminder, GVFS, together with a set of enhancements to Git, enables Git to scale to VERY large repos by virtualizing both the .git folder and the working directory. Rather than download the entire repo and checkout all the files, it dynamically downloads only the portions you need based on what you use.

A lot has happened and I wanted to give you an update. Three months ago, GVFS was still a dream. I don't mean it didn't exist – we had a concrete implementation, but rather, it was unproven. We had validated on some big repos but we hadn't rolled it out to any meaningful number of engineers so we had only conviction that it was going to work. Now we have proof.

Today, I want to share our results. In addition, we're announcing the next steps in our GVFS journey for customers, including expanded open sourcing to start taking contributions and improving how it works for us at Microsoft, as well as for partners and customers.

Windows is live on Git

Over the past 3 months, we have largely completed the rollout of Git/GVFS to the Windows team at Microsoft.

As a refresher, the Windows code base is approximately 3.5M files and, when checked in to a Git repo, results in a repo of about 300GB.

Visual Studio

Download Visual Studio

Download TFS

Visual Studio Team Services

Search

Search MSDN with Bing

Search this blog | Search all blogs

Subscribe Blog via Email

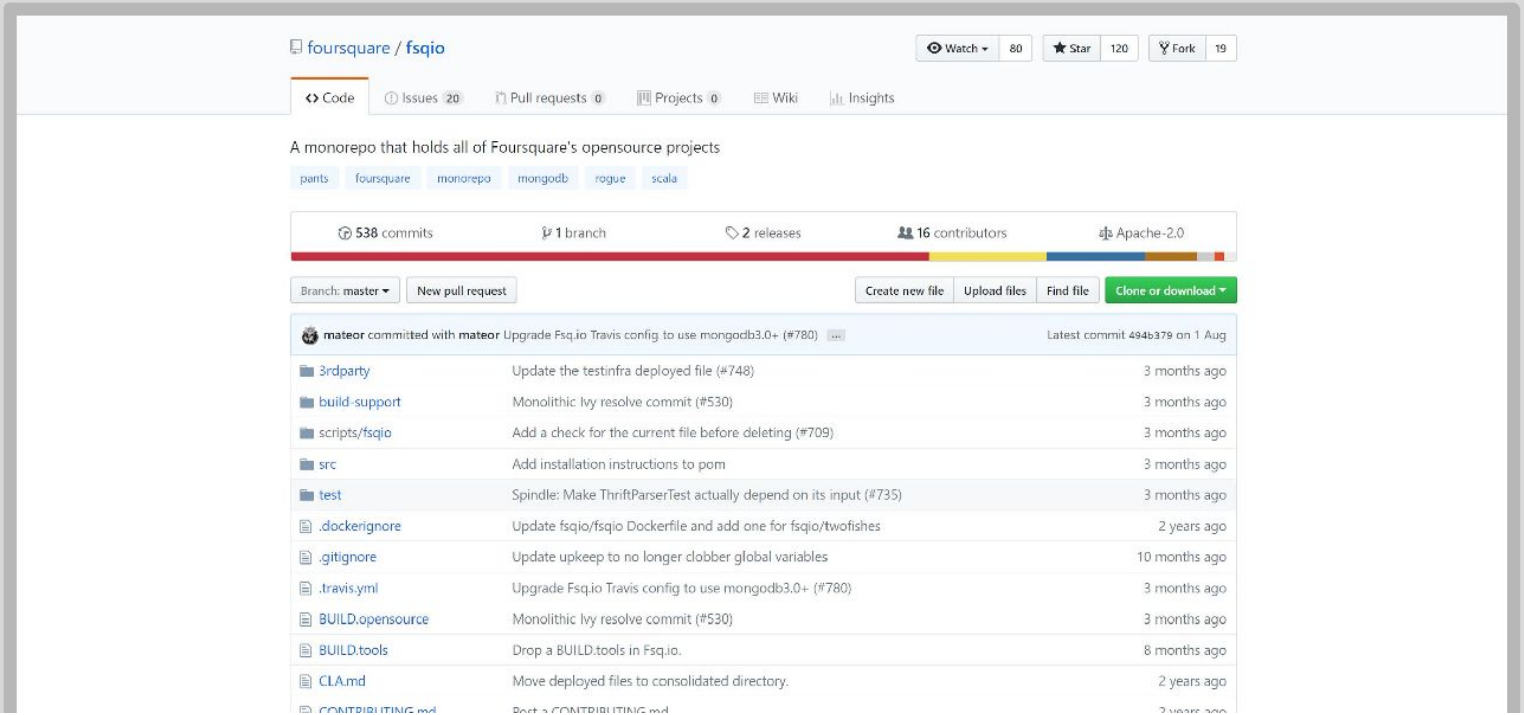
Subscribe to this blog and receive notifications of new posts by email.

Email Address

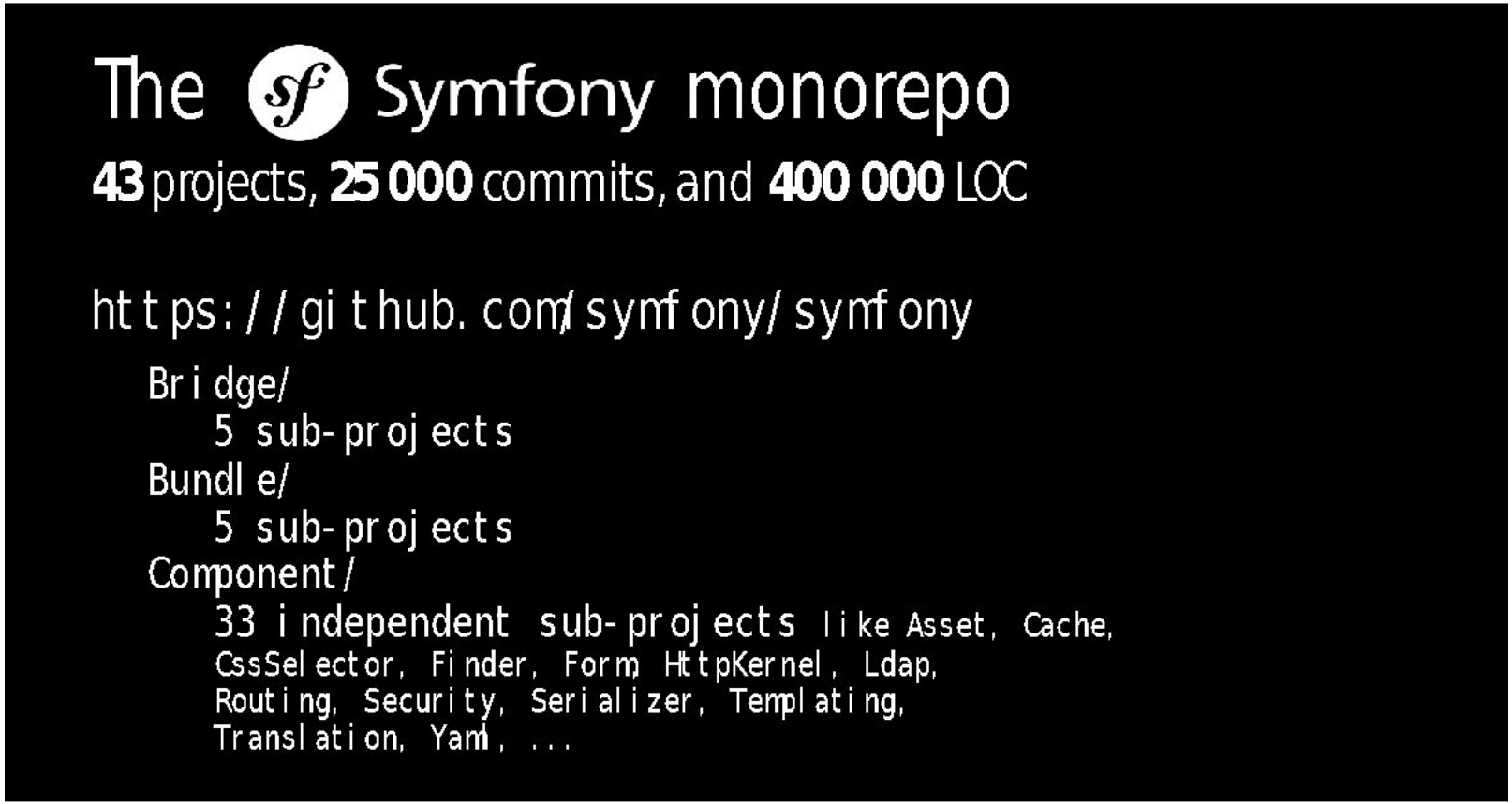
Subscribe! | Unsubscribe

Monorepos in open-source

foresquare public monorepo



Monorepos in open-source



Advantages of Monorepos

- High discoverability
  - Developers can read & search the entire codebase
- High reuse
  - The same tools (e.g., linters, auto-complete) are globally available
  - Any package can become a library
    - Which is why you always build an API!
- Simplifies maintenance
  - Global refactorings, cleanup
    - Orgs like Google will regularly dedicate a specific day to a type of improvement (e.g., improve documentation), flag all potentially problematic sites

Some more advantages

- Easy continuous integration and code review for changes spanning several projects
- (Internal) dependency management is a non-issue
- Less context switching for developers
- Code more reusable in other contexts
- Access control is easy

Summary

- Release management: versioning, branching, …
- Software development at scale requires lots of infrastructure
  - Version control, build managers, testing, CI, deployment, …
- It's hard to scale development
  - Move towards heavy automation (DevOps)
- Continuous deployment increasingly common
- Opportunities from quick release, testing in production, quick rollback