Principles of Software Construction:
Objects, Design, and Concurrency

Software development at scale
Bonus slides: Unseen GoF design patterns

(The end)

Michael Hilton **Bogdan Vasilescu**

School of
Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Final exam Monday May 6th 5:30-8:30 GHC 4401
- Review session Saturday May 4th 1pm NSH 3305

Intro to Java

Git, CI

UML

Static Analysis

GUIs

More Git

Streams

Software Engineering in Practice

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Unit Testing, Design Patterns	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Design for Reuse at Scale: Frameworks and APIs	

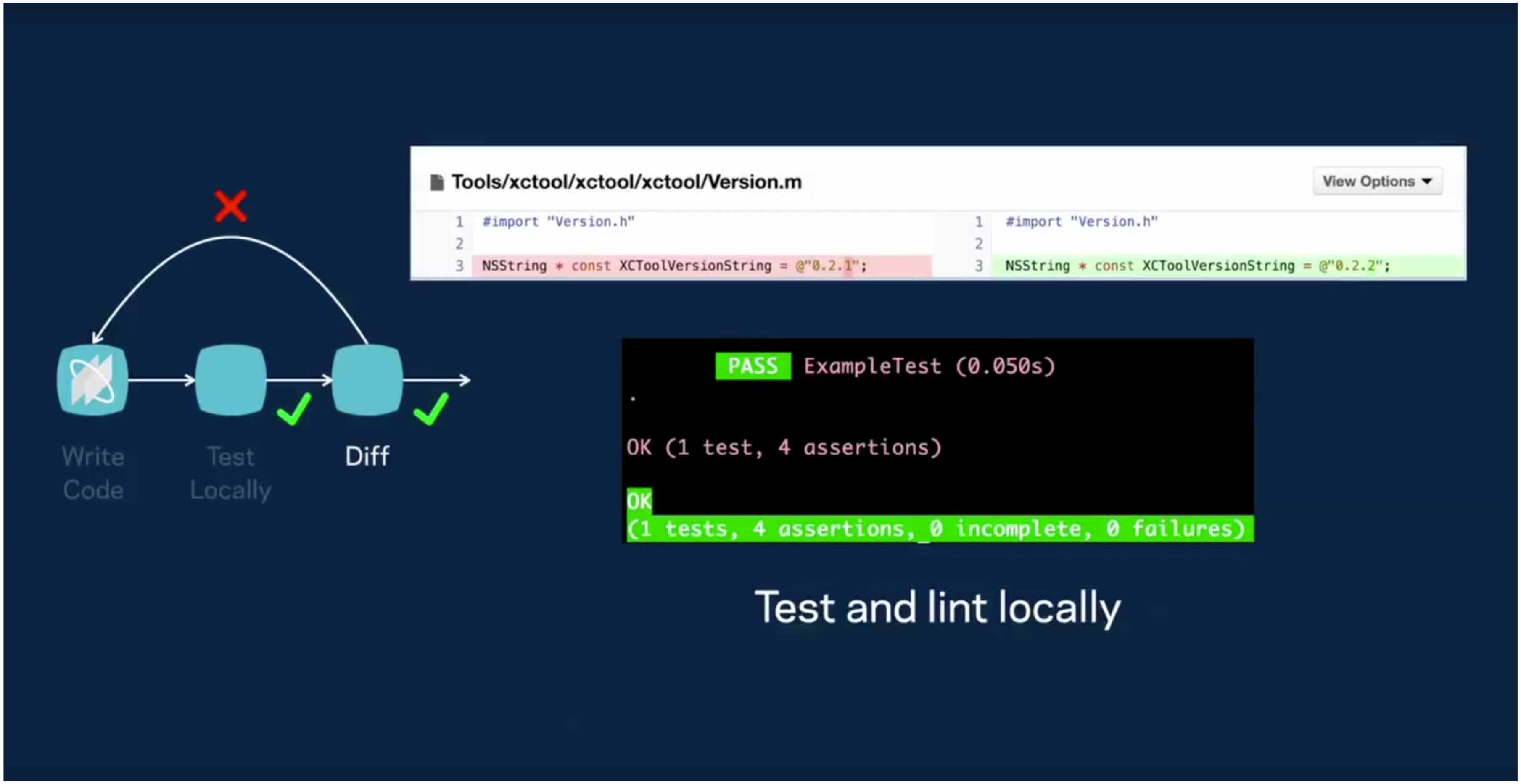
SOFTWARE DEVELOPMENT AT SCALE

Releasing at scale in industry

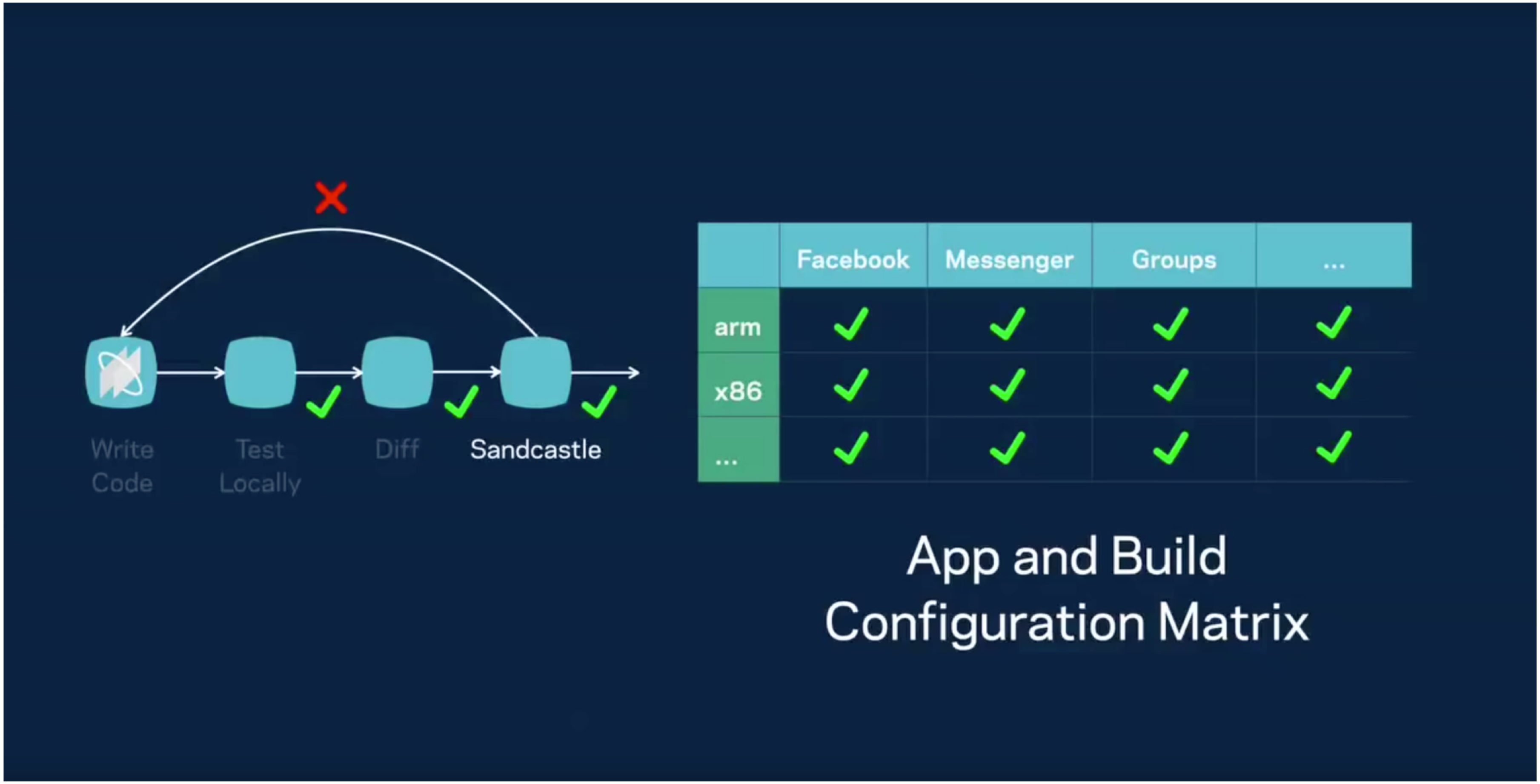
- Facebook: https://atscaleconference.com/videos/rapid-release-at-massive-scale/
- Google: https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale
  - https://testing.googleblog.com/2011/06/testing-at-speed-and-scale-of-google.html
- Why Google Stores Billions of Lines 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: First, local testing

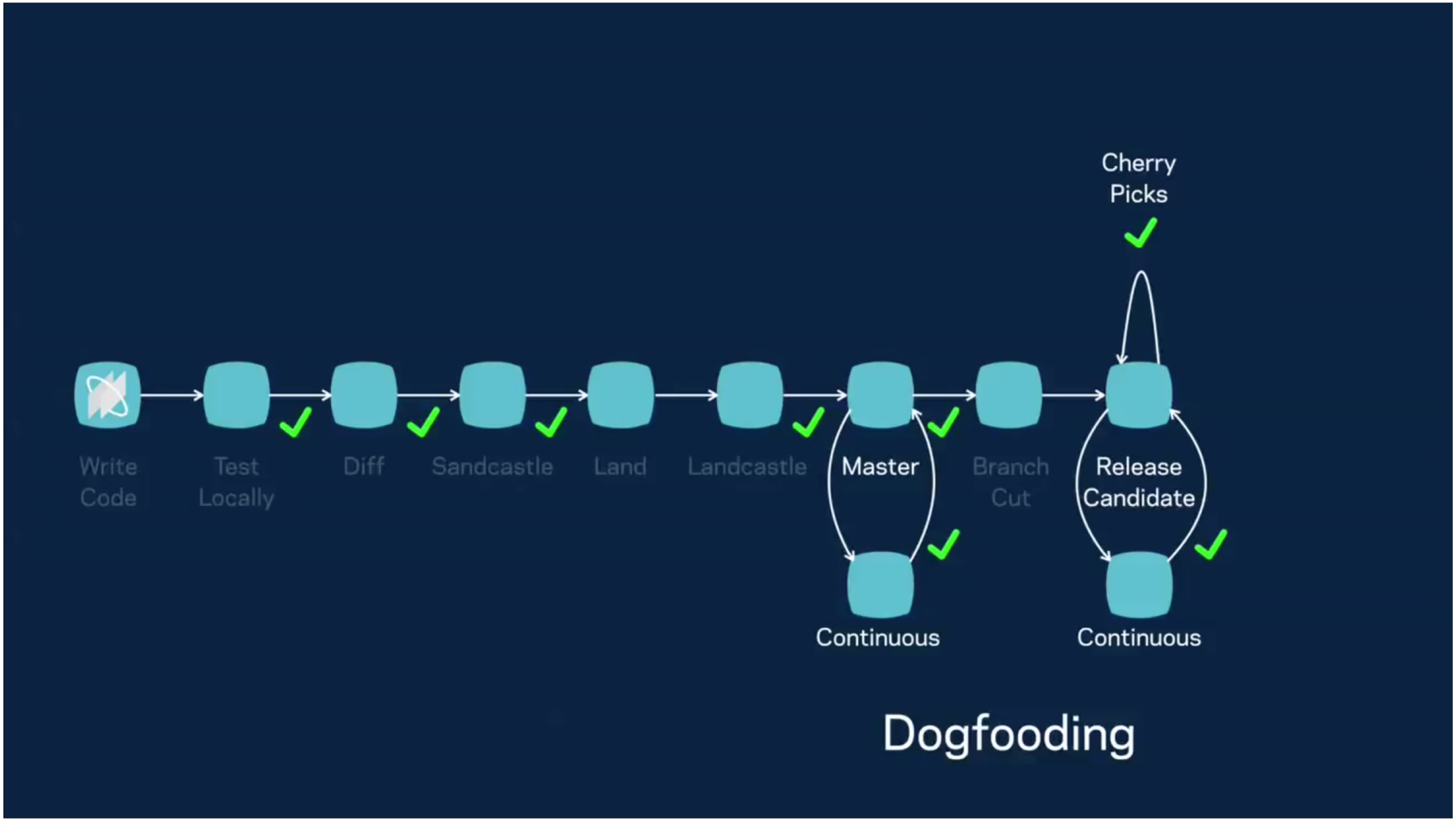


Diff lifecycle: Next, CI testing (data center)

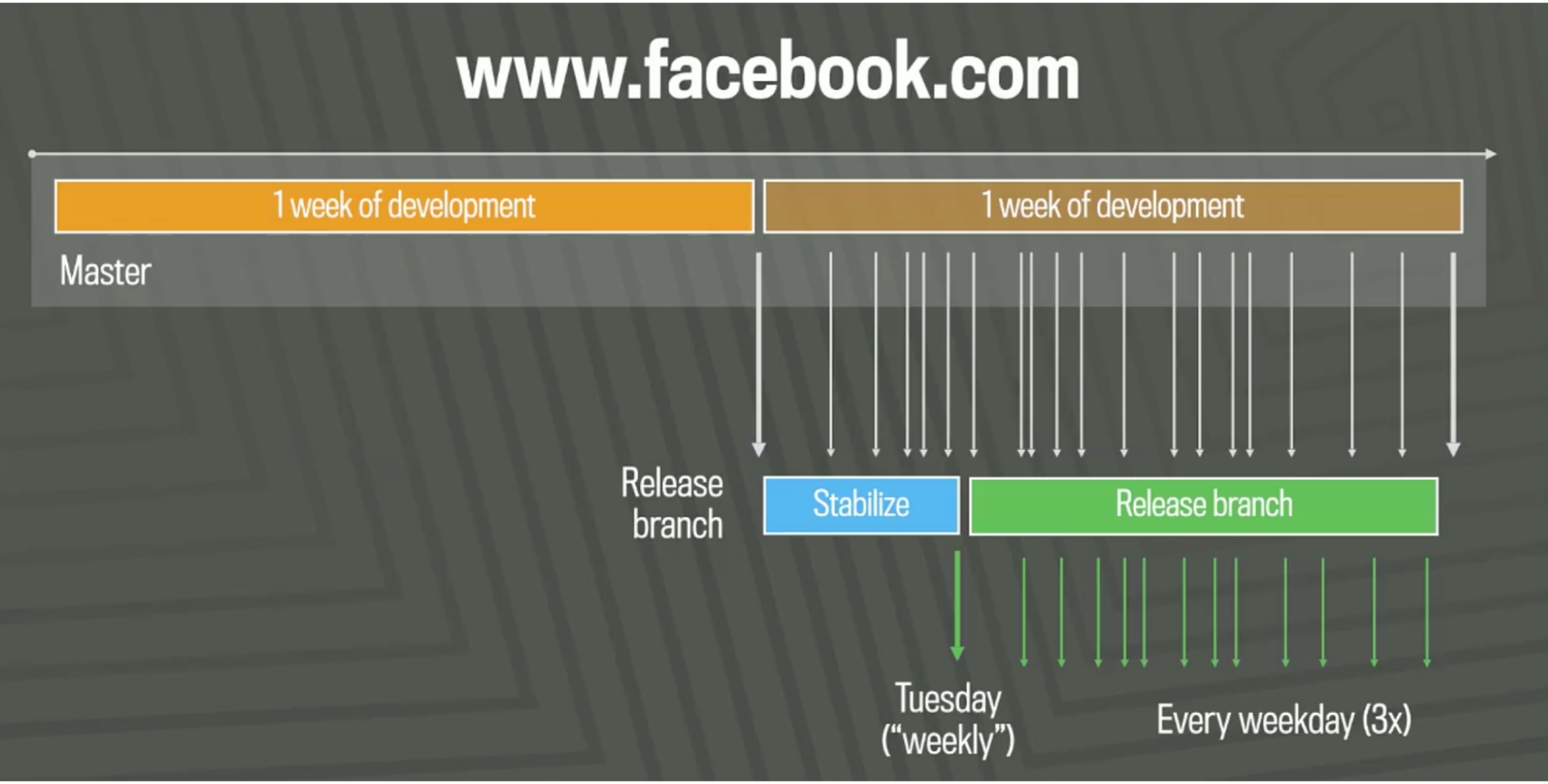


institute for SOFTWARE RESEARCH

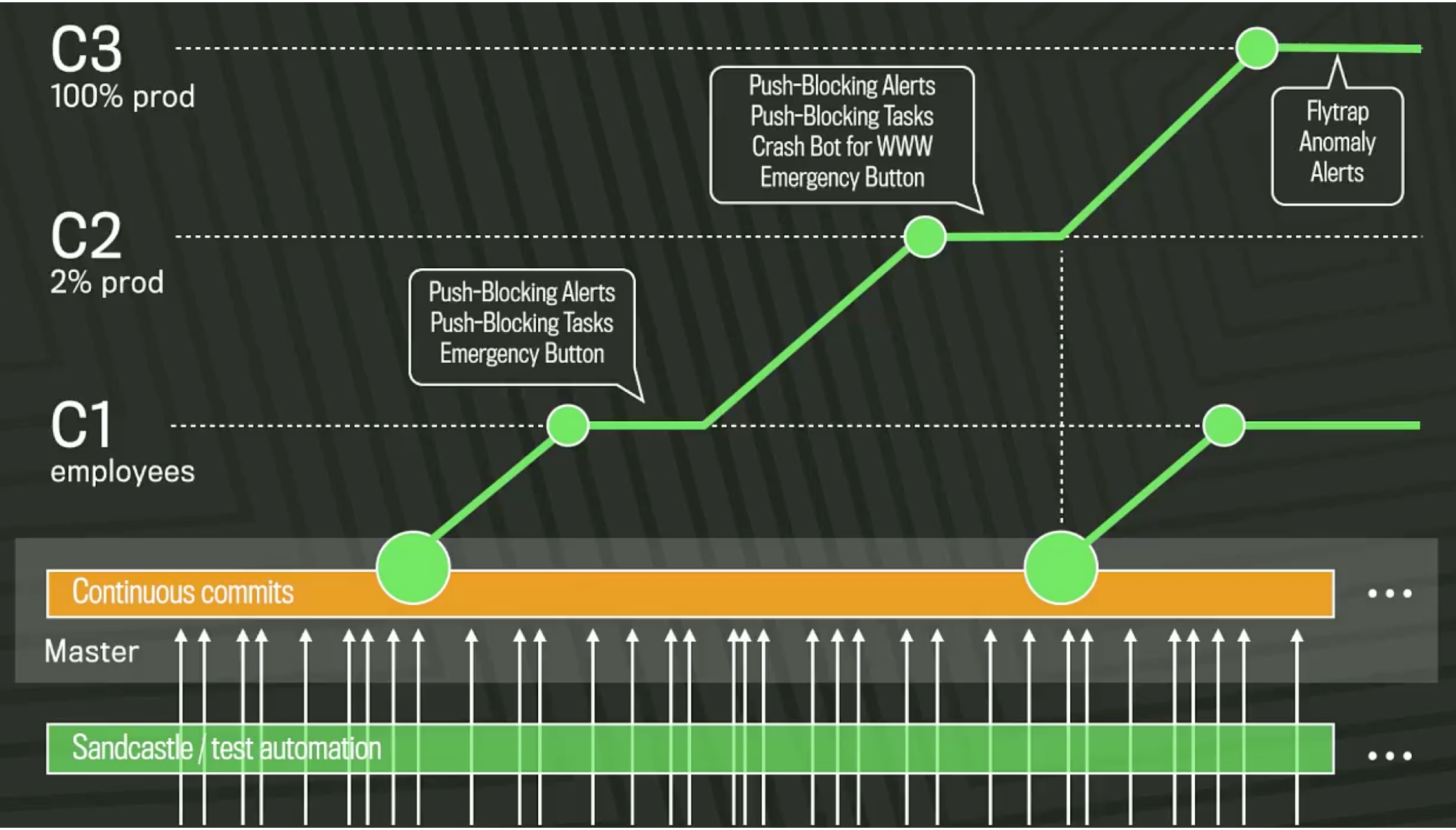
Diff lifecycle: Then, diff ends up on master



Release every two weeks



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



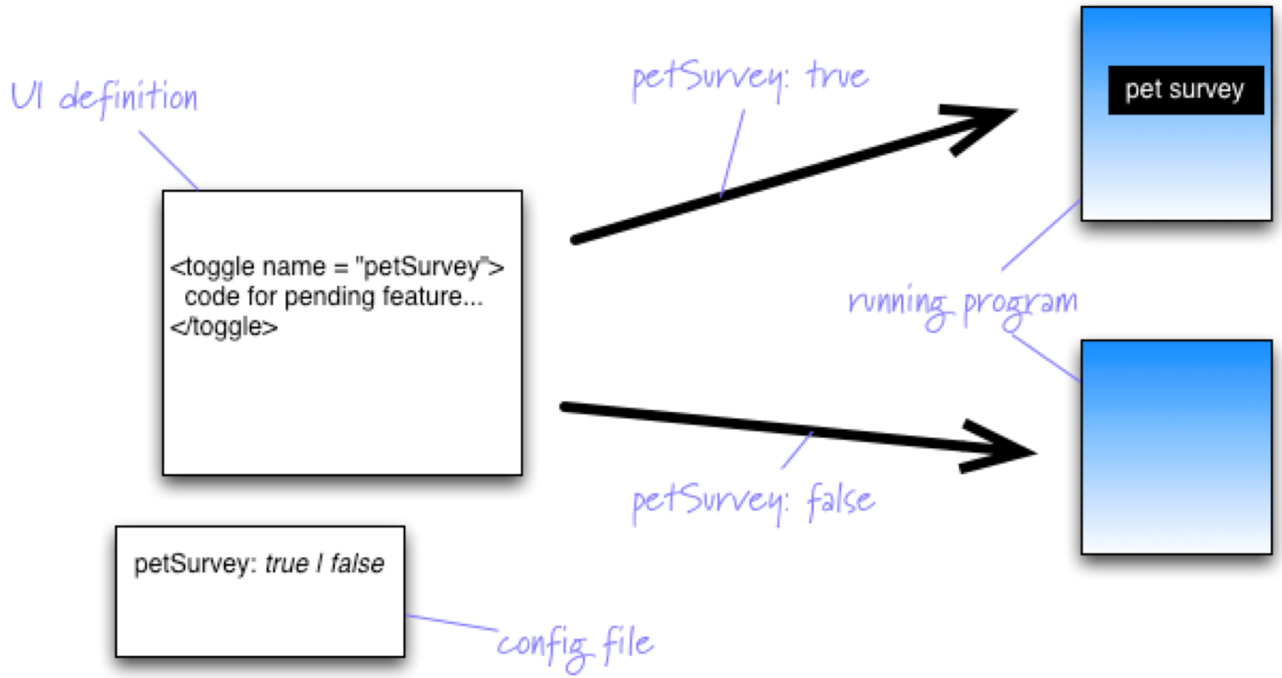
Aside: Key idea – fast to deploy, slow to release

Dark launches at Instagram

- **Early**: Integrate as soon as possible. Find bugs early. Code can run in production about 6 months before being publicly announced ("dark launch").
- **Often**: Reduce friction. Try things out. See what works. Push small changes just to gather metrics, feasibility testing. Large changes just slow down the team. Do dark launches, to see what performance is in production, can scale up and down. *"Shadow infrastructure" is too expensive, just do in production.*
- **Incremental**: Deploy in increments. Contain risk. Pinpoint issues.

Aside: Feature Flags

Typical way to implement a dark launch.



http://swreflections.blogspot.com/2014/08/feature-toggles-are-one-of-worst-kinds.html

http://martinfowler.com/bliki/FeatureToggle.html

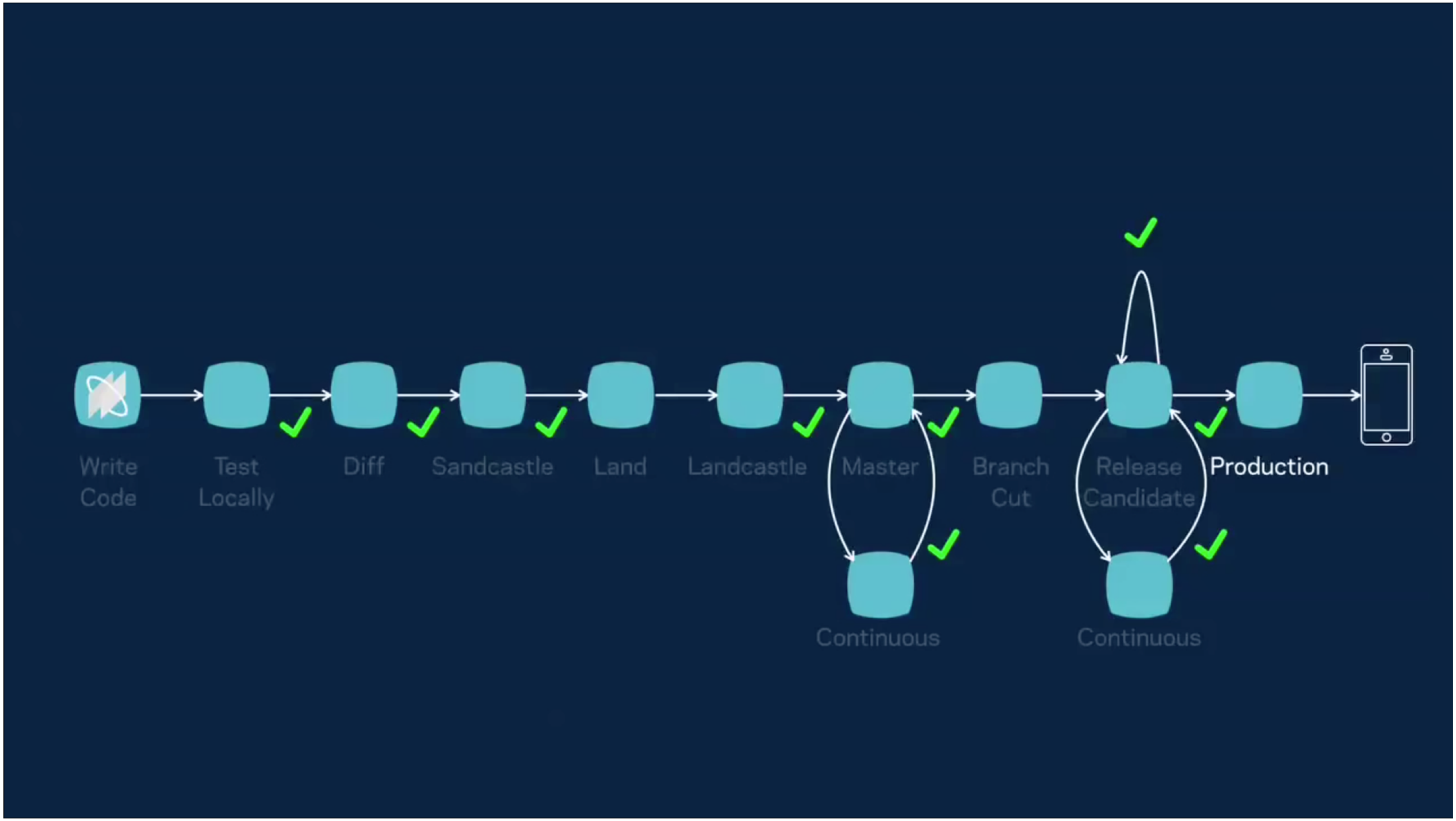
Issues with feature flags

Feature flags are “technical debt”

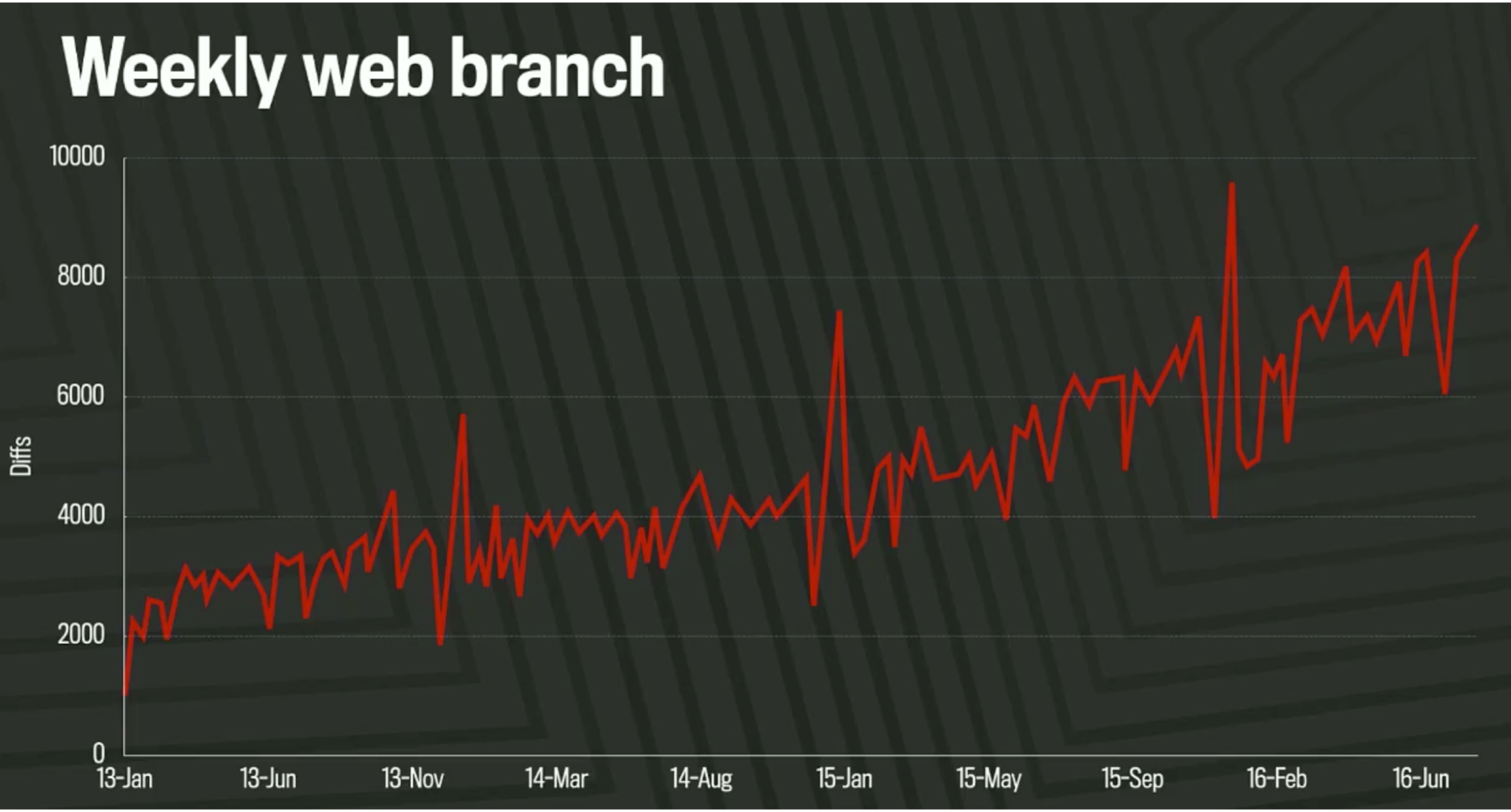
Example: financial services company with nearly $400 million in assets went bankrupt in 45 minutes.

http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/

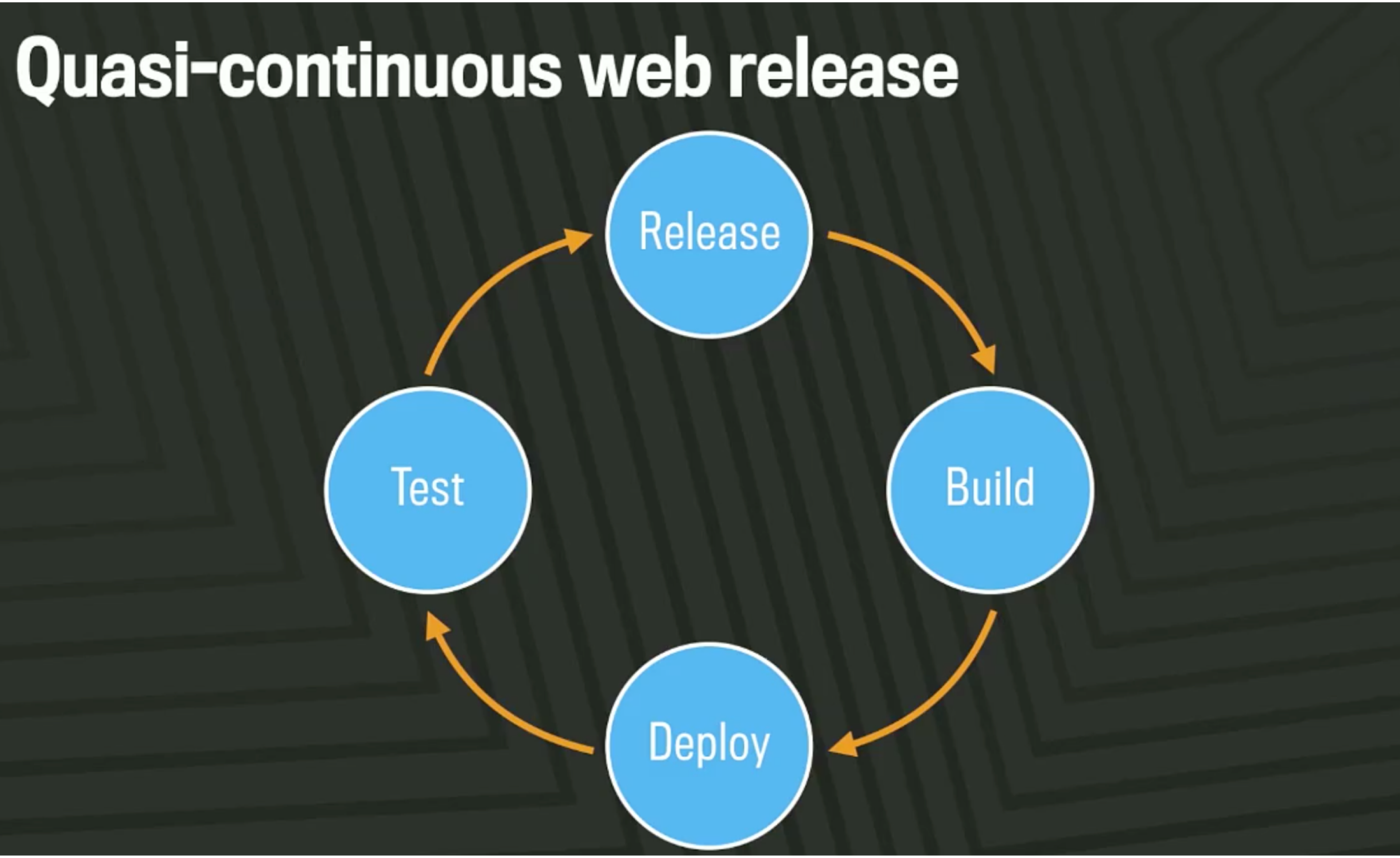
Diff lifecycle: Finally, in production



What's in a weekly branch cut? (The limits of branches)



Post-2017 release management model at Facebook



Google: similar story. HUGE code base

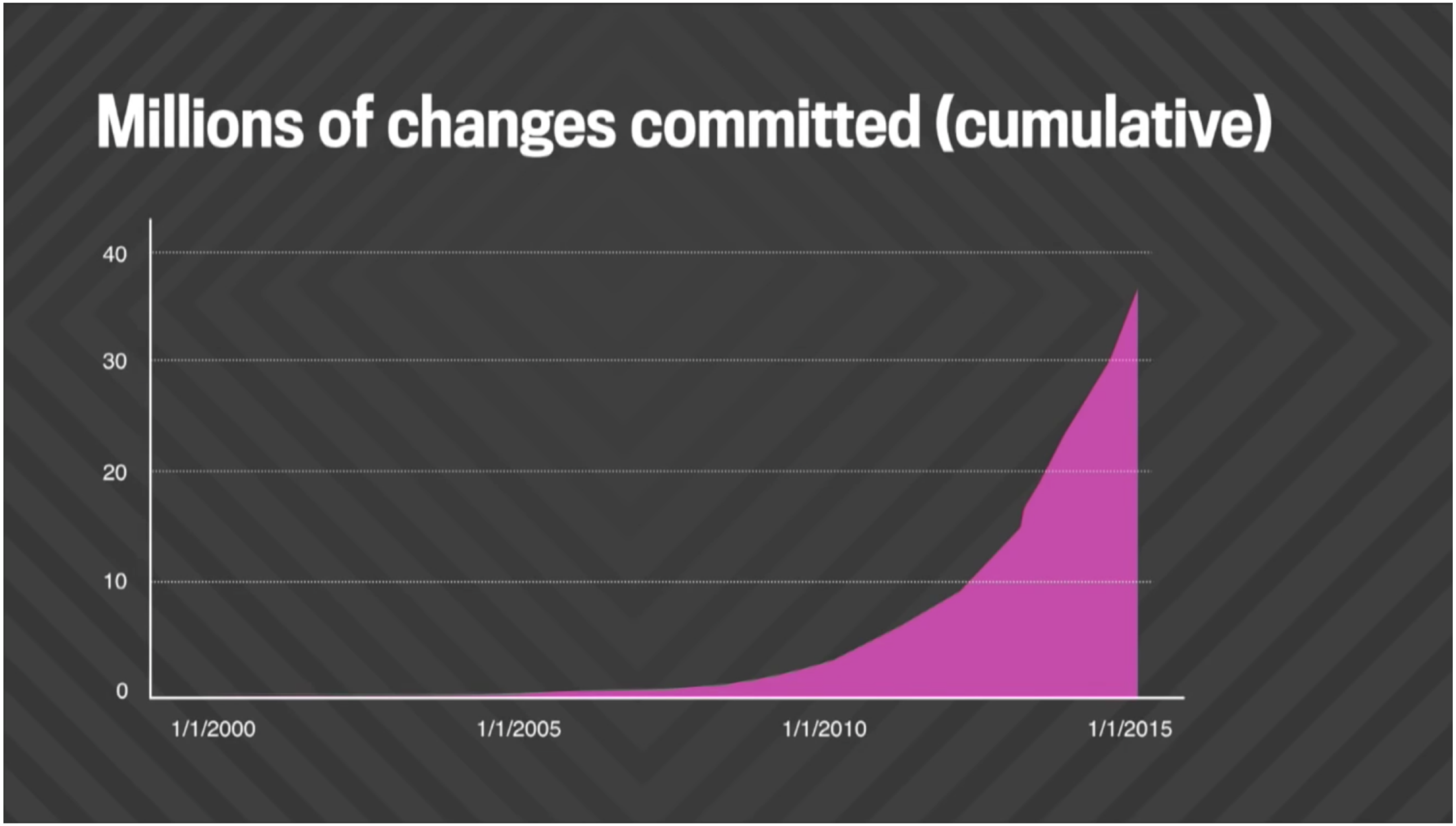
Google repository statistics

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)

- 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!

Google code base vs Linux kernel code base

Some perspective

Linux kernel

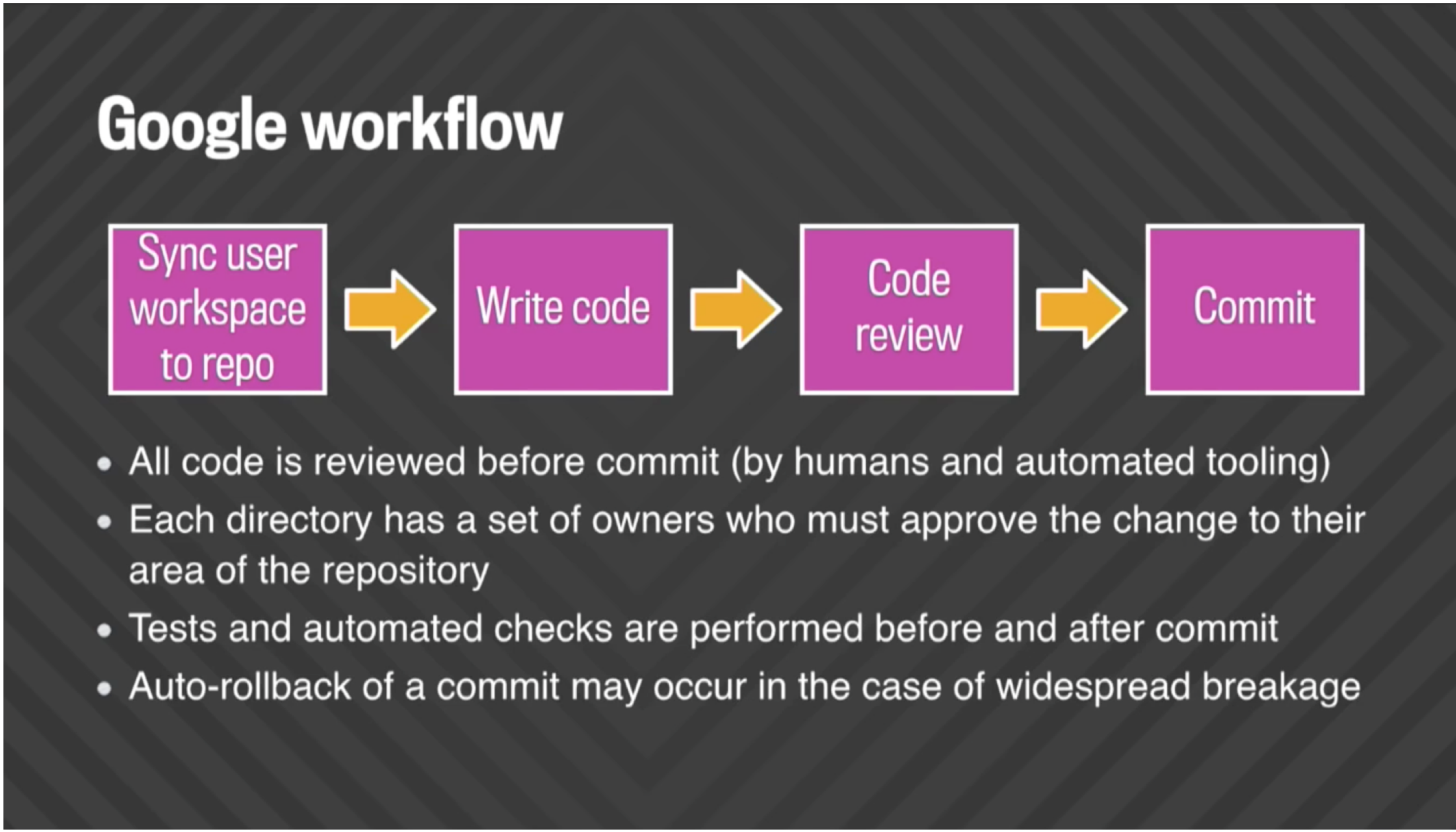
- 15 million lines of code in 40 thousand files (total)

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?

1. Lots of (automated) testing



2. Lots of automation

Additional tooling support

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

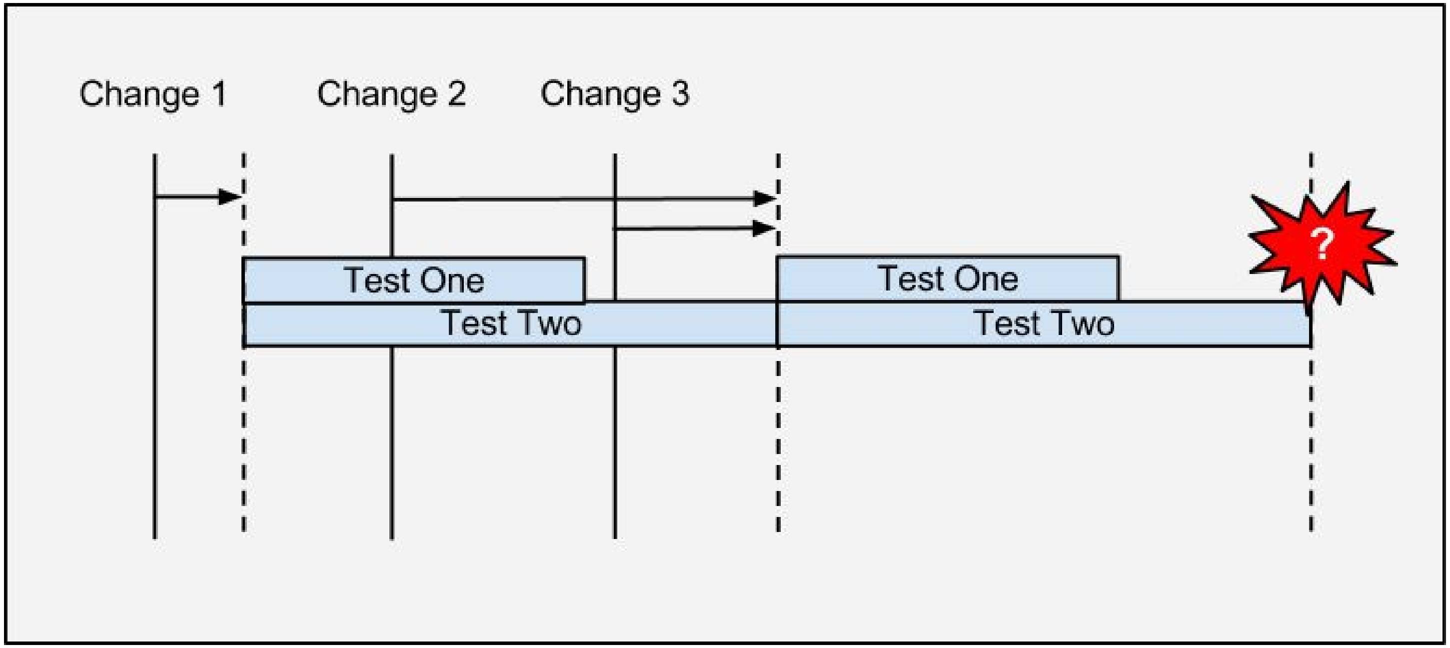
3. Smarter tooling

- Build system
- Version control
- …

3a. Build system

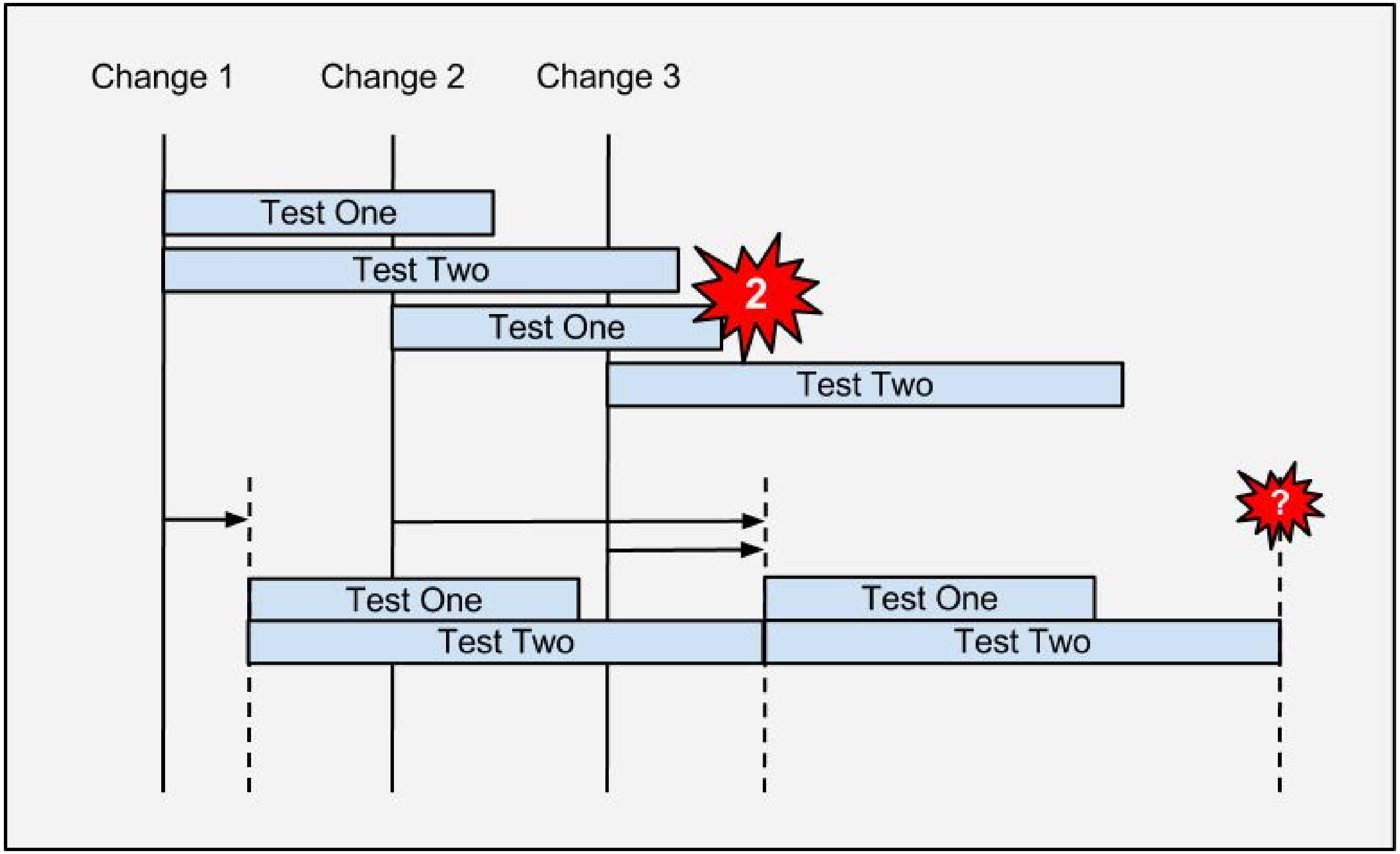
Standard Continuous Build System

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



Google Continuous Build System

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



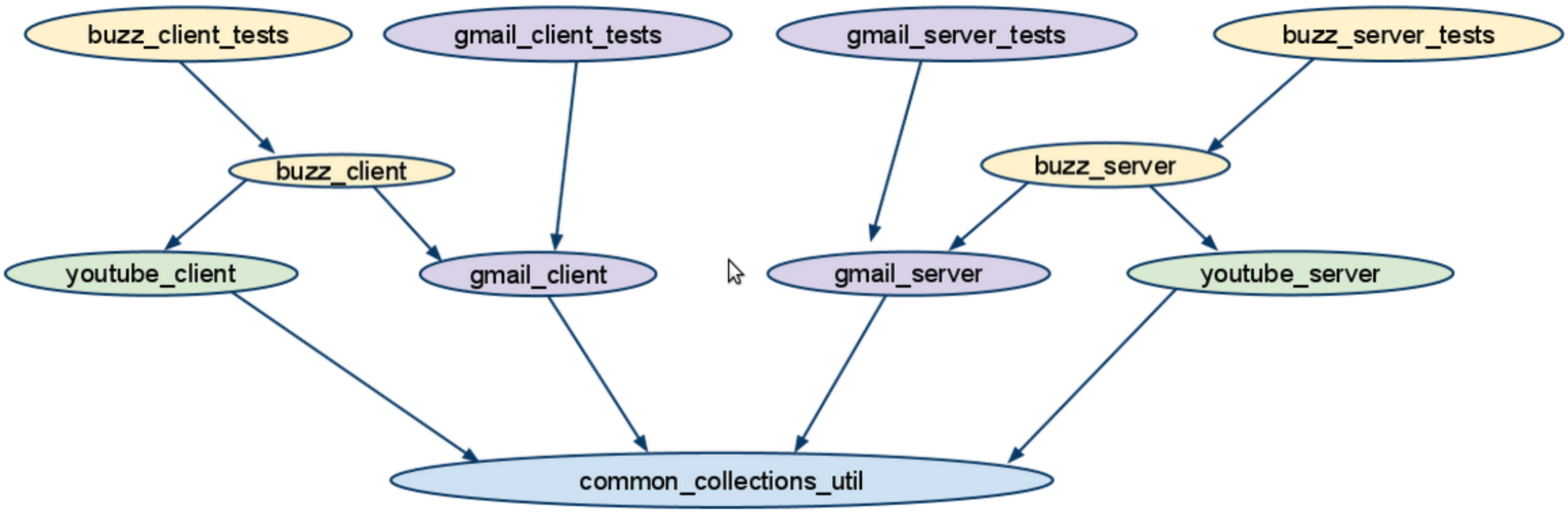
Which tests to run?

GMAIL

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

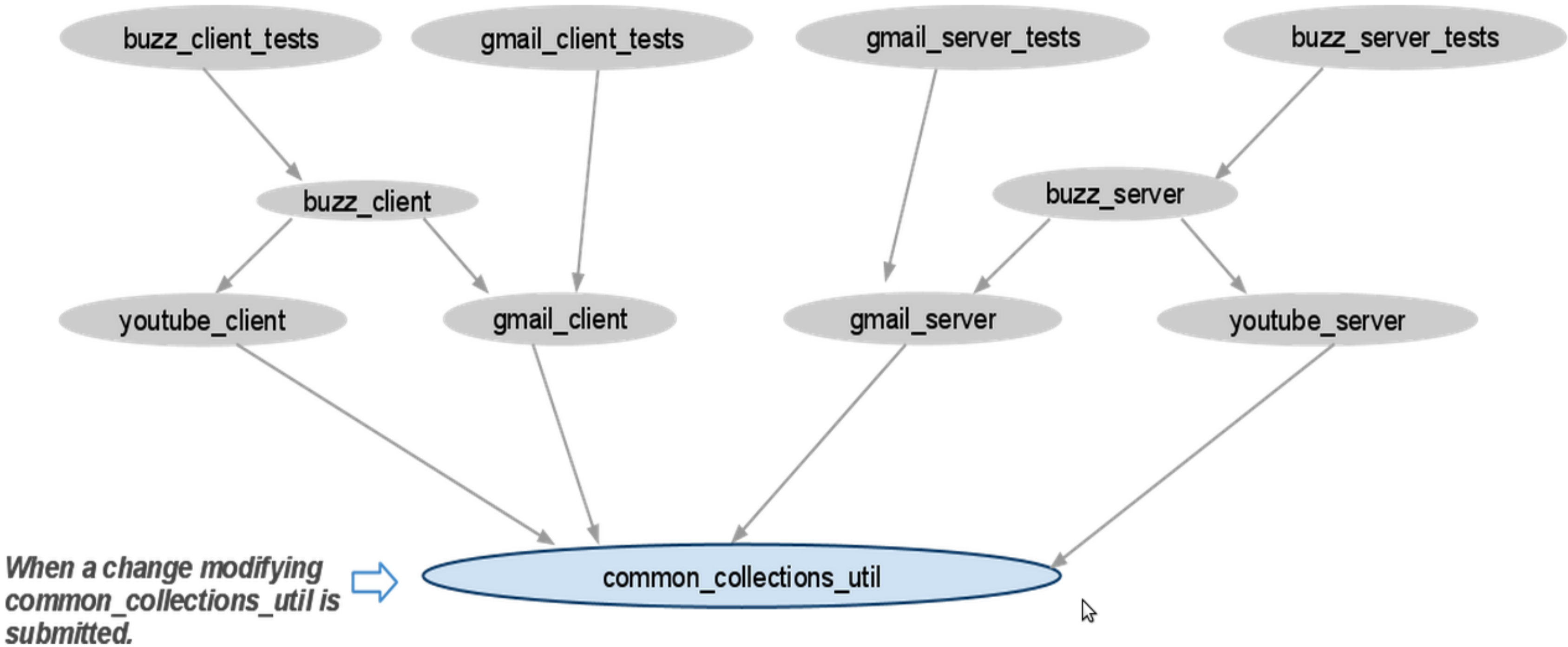
BUZZ

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

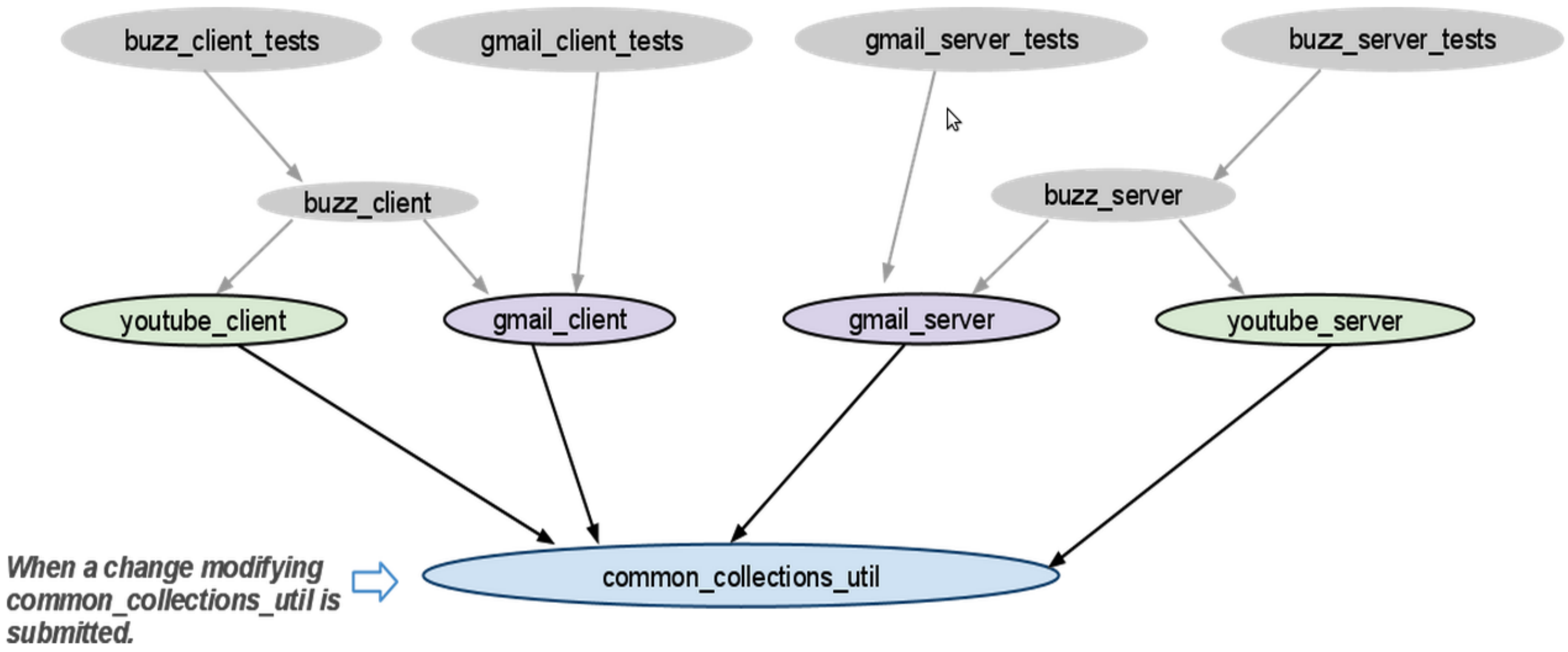


institute for SOFTWARE RESEARCH

Scenario 1: a change modifies common_collections_util

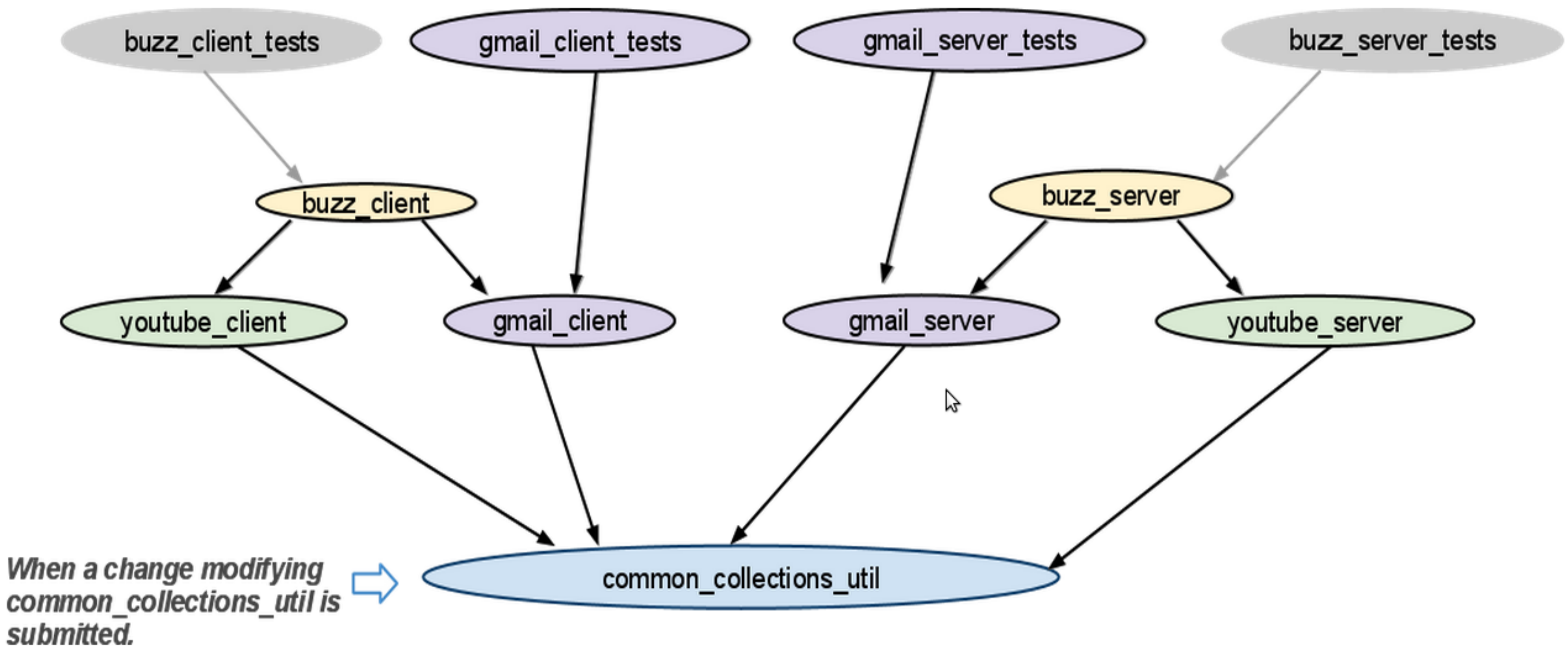


Scenario 1: a change modifies common_collections_util

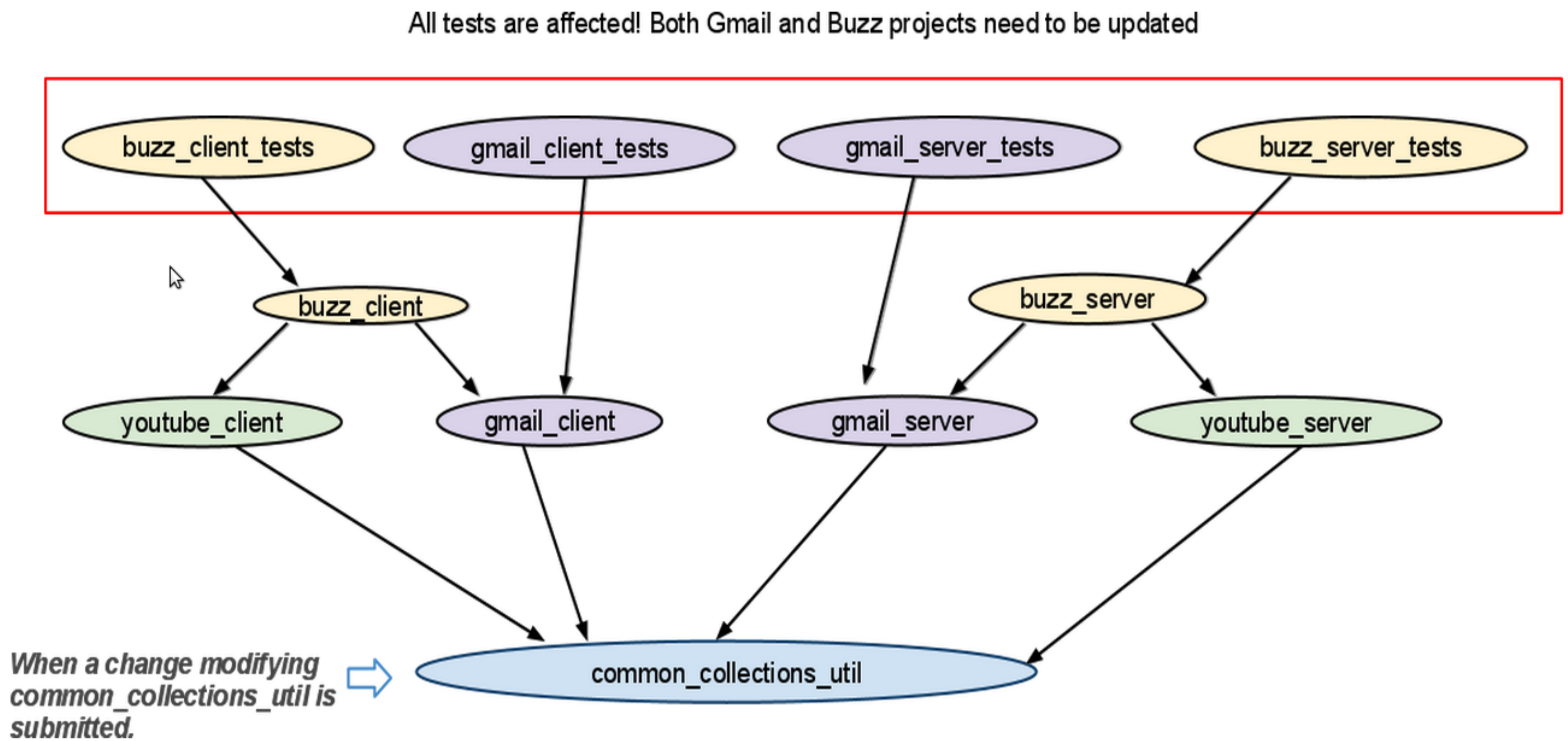


institute for SOFTWARE RESEARCH

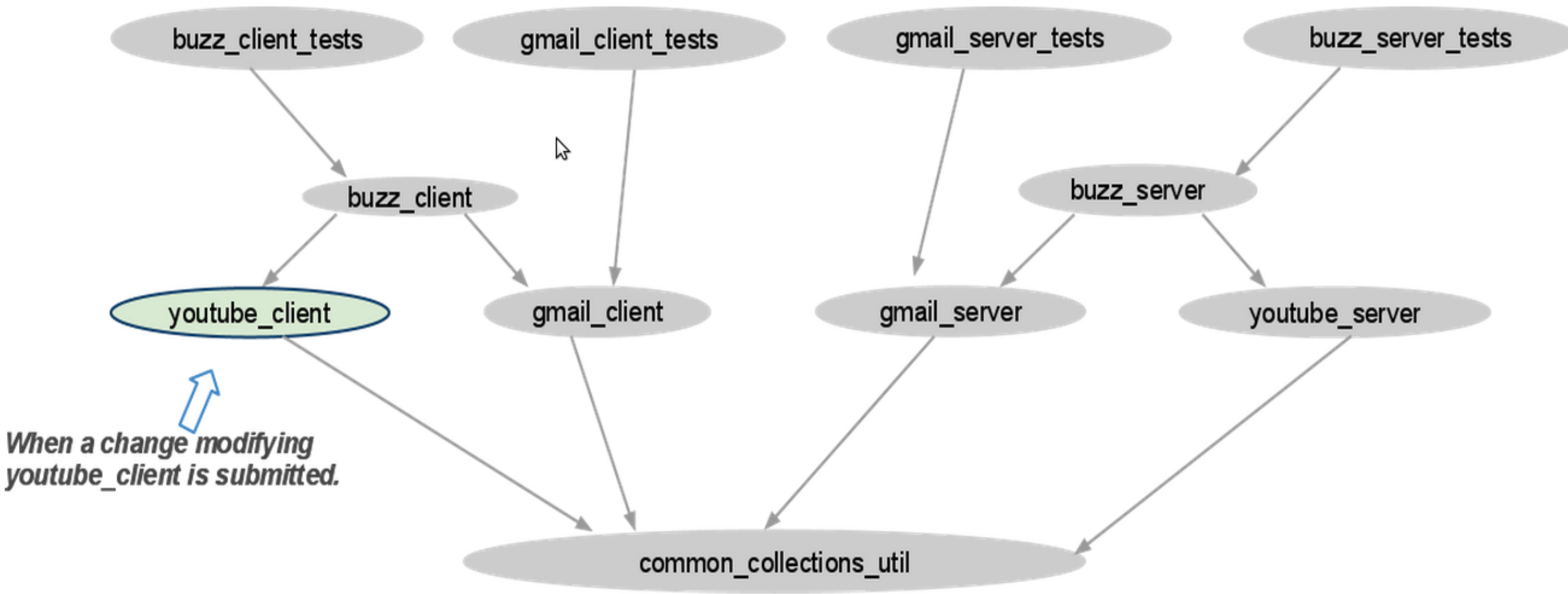
Scenario 1: a change modifies common_collections_util



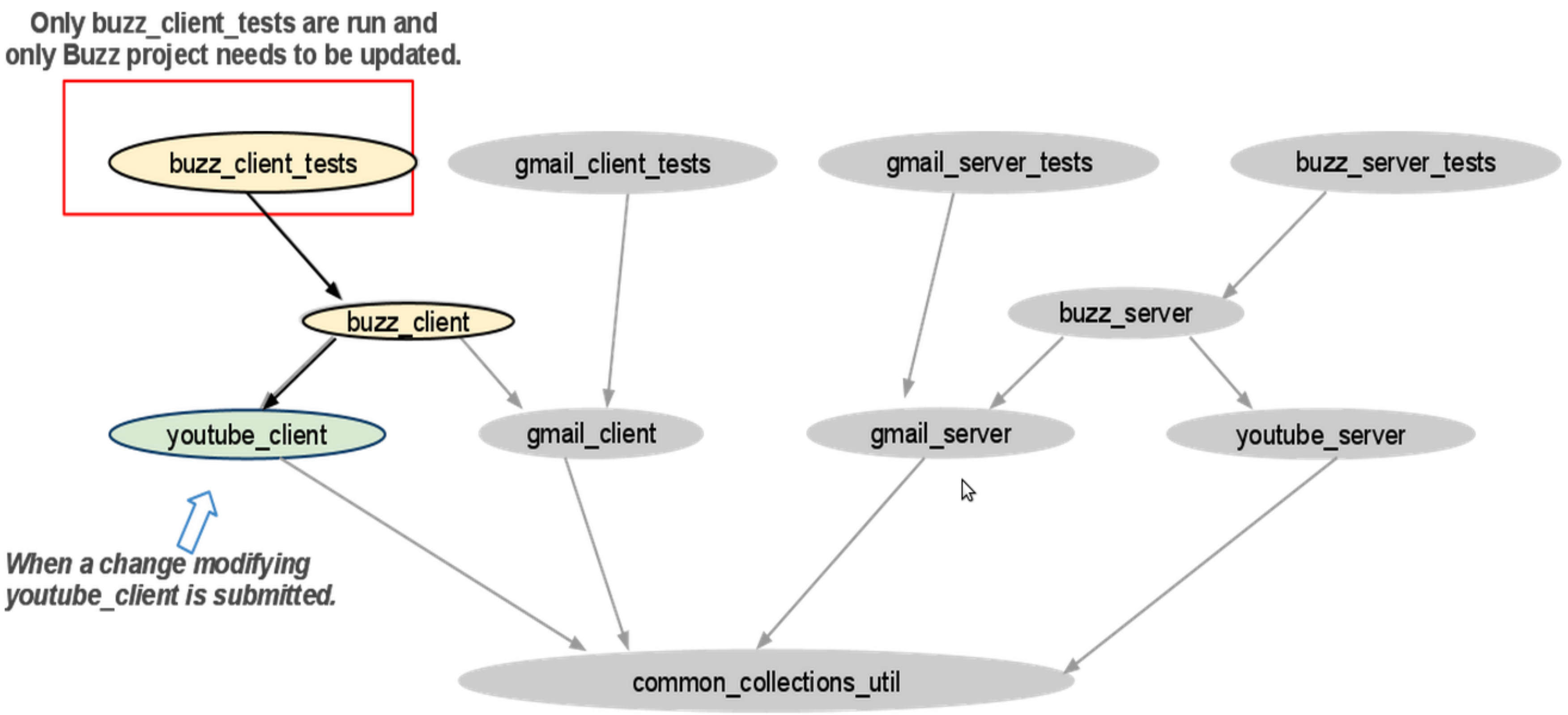
Scenario 1: a change modifies common_collections_util



Scenario 2: a change modifies the youtube_client

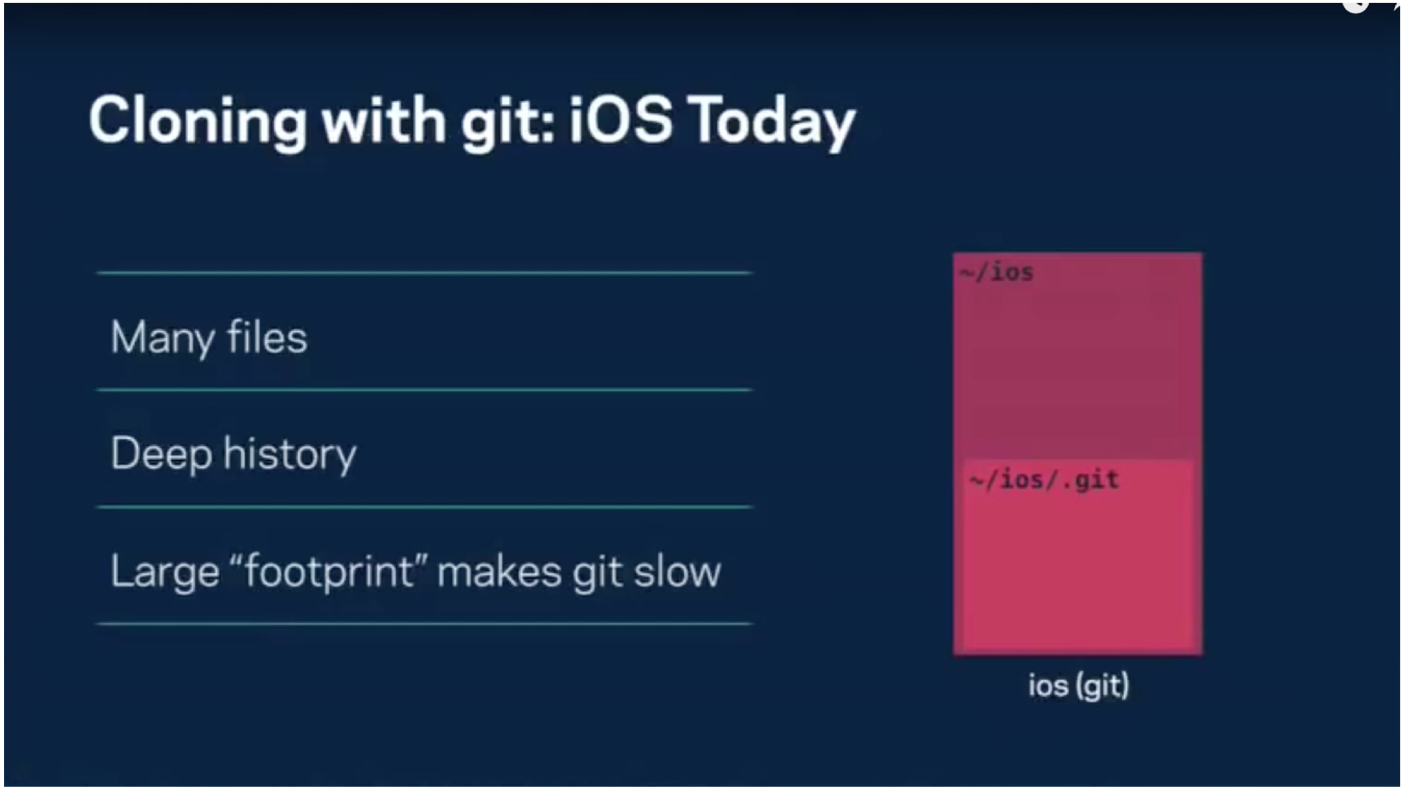


Scenario 2: a change modifies the youtube_client



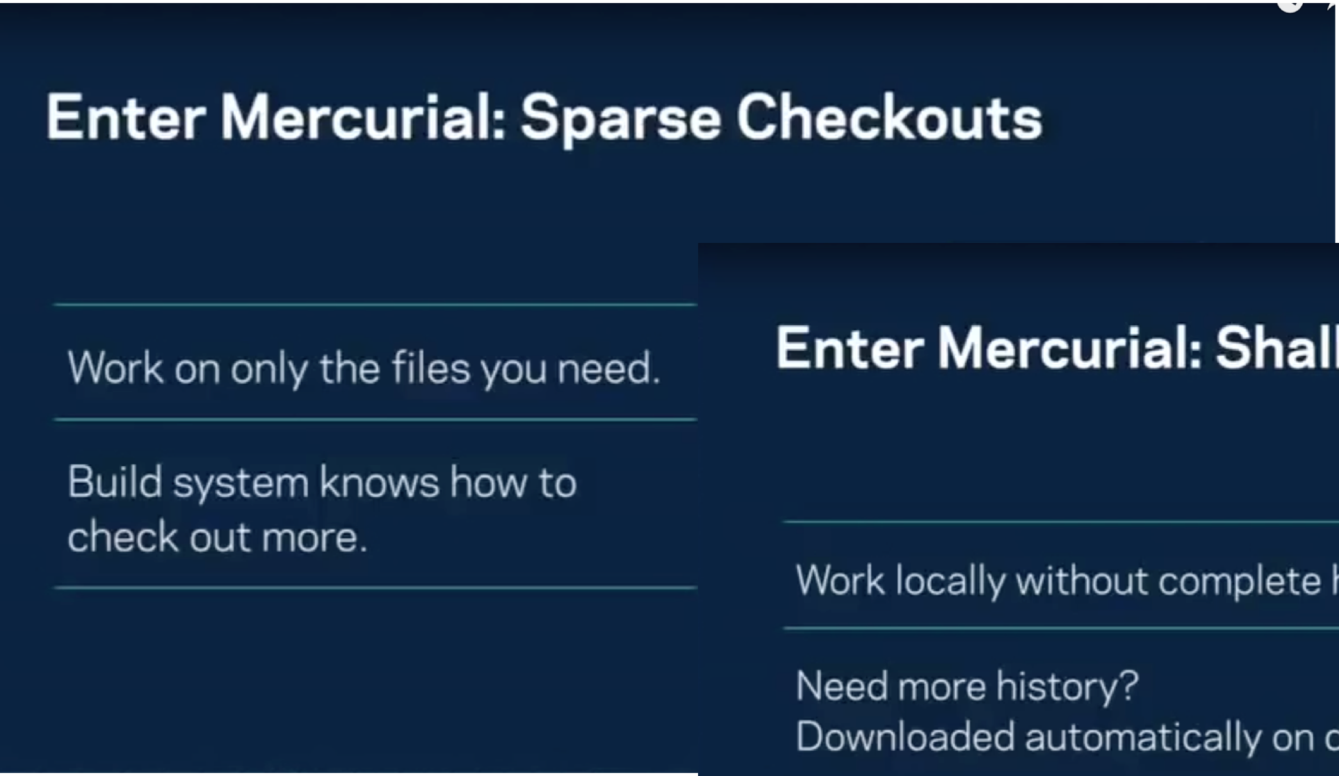
3b. Version control

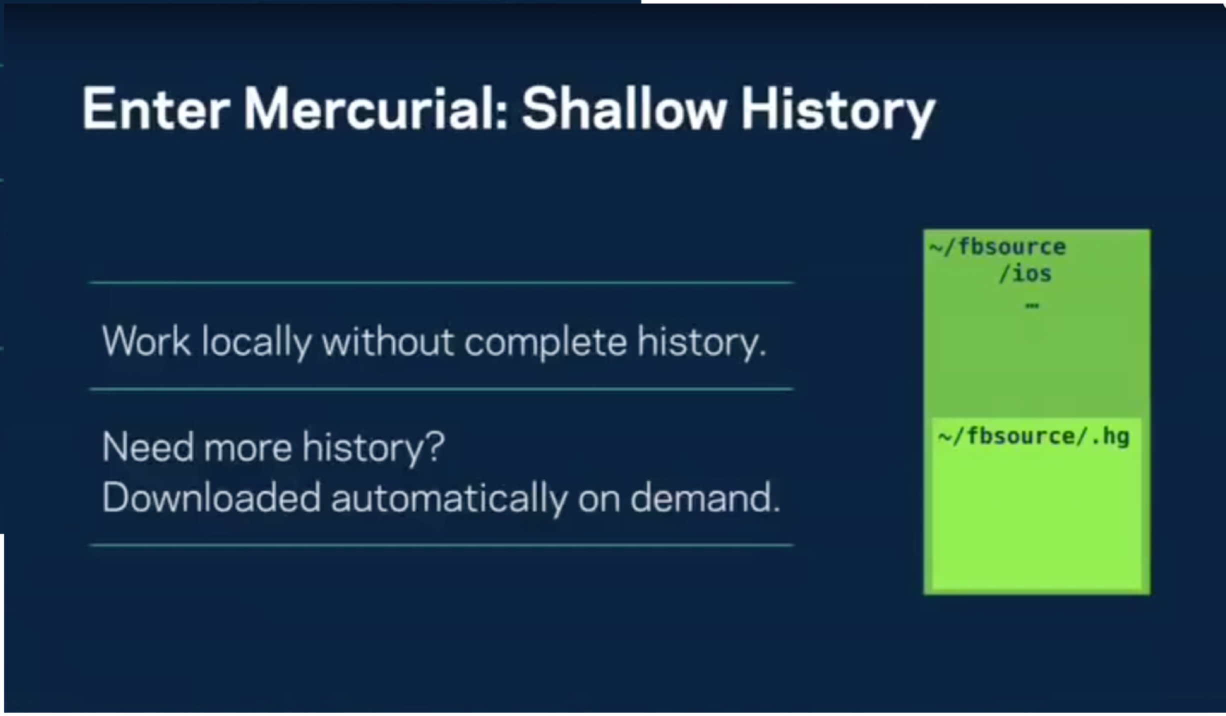
- Problem: even git can get slow at Facebook-like scale
  - 1M+ source control commands run per day
  - 100K+ commits per week



3b. Version control

- Solution: redesign version control



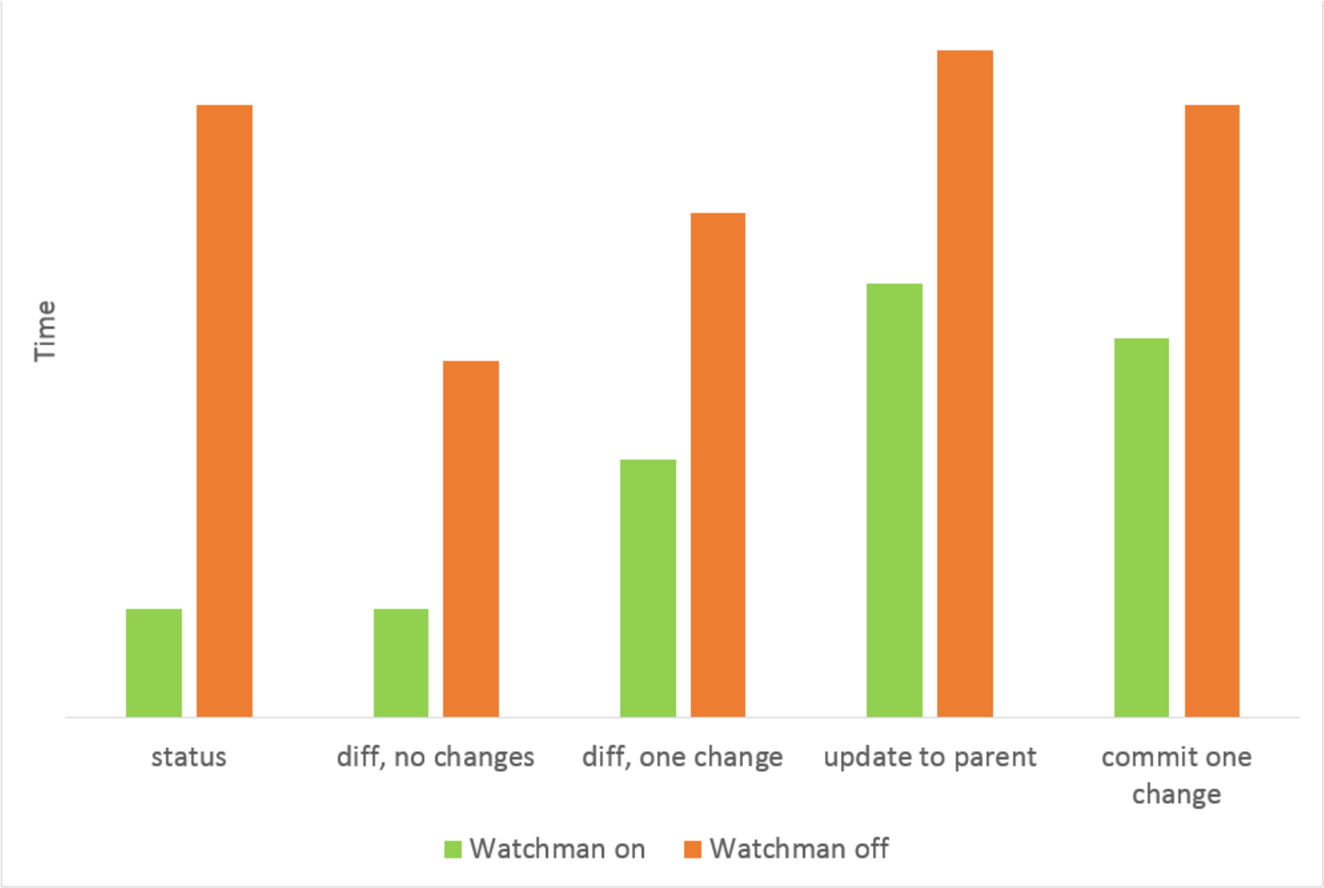


3b. Version control

- Solution: redesign version control
  - Query build system's file monitor, Watchman, to see which files have changed

3b. Version control

- Solution: redesign version control
  - Query build system's file monitor, Watchman, to see which files have changed → **5x faster "status" command**



institute for SOFTWARE RESEARCH

3b. Version control

- Solution: redesign version control
  - Sparse checkouts??? (remember, git is a distributed VCS)

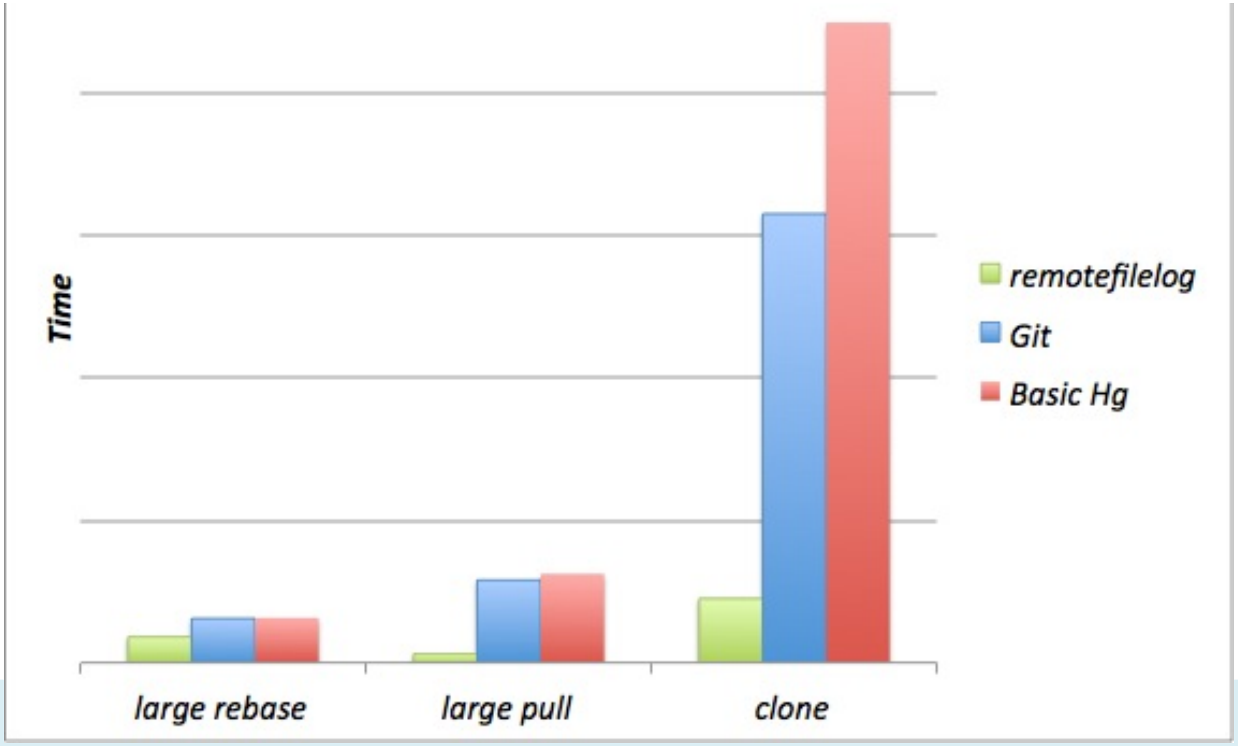
3b. Version control

- Solution: redesign version control
  - Sparse checkouts:
  - Change the clone and pull commands to download only the commit metadata, while omitting all file changes (the bulk of the download)
  - When a user performs an operation that needs the contents of files (such as checkout), download the file contents on demand using existing memcache infrastructure

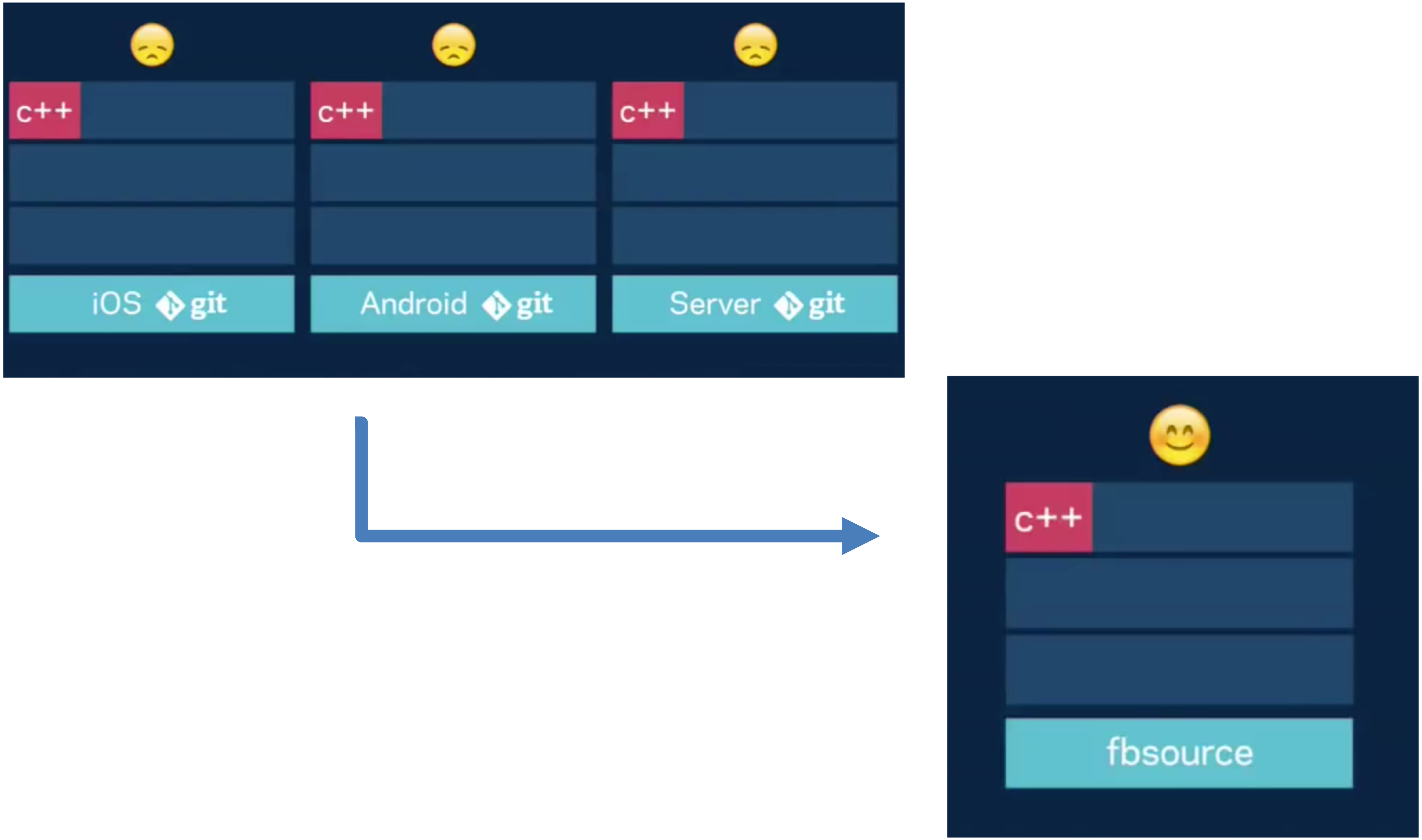
institute for SOFTWARE RESEARCH

3b. Version control

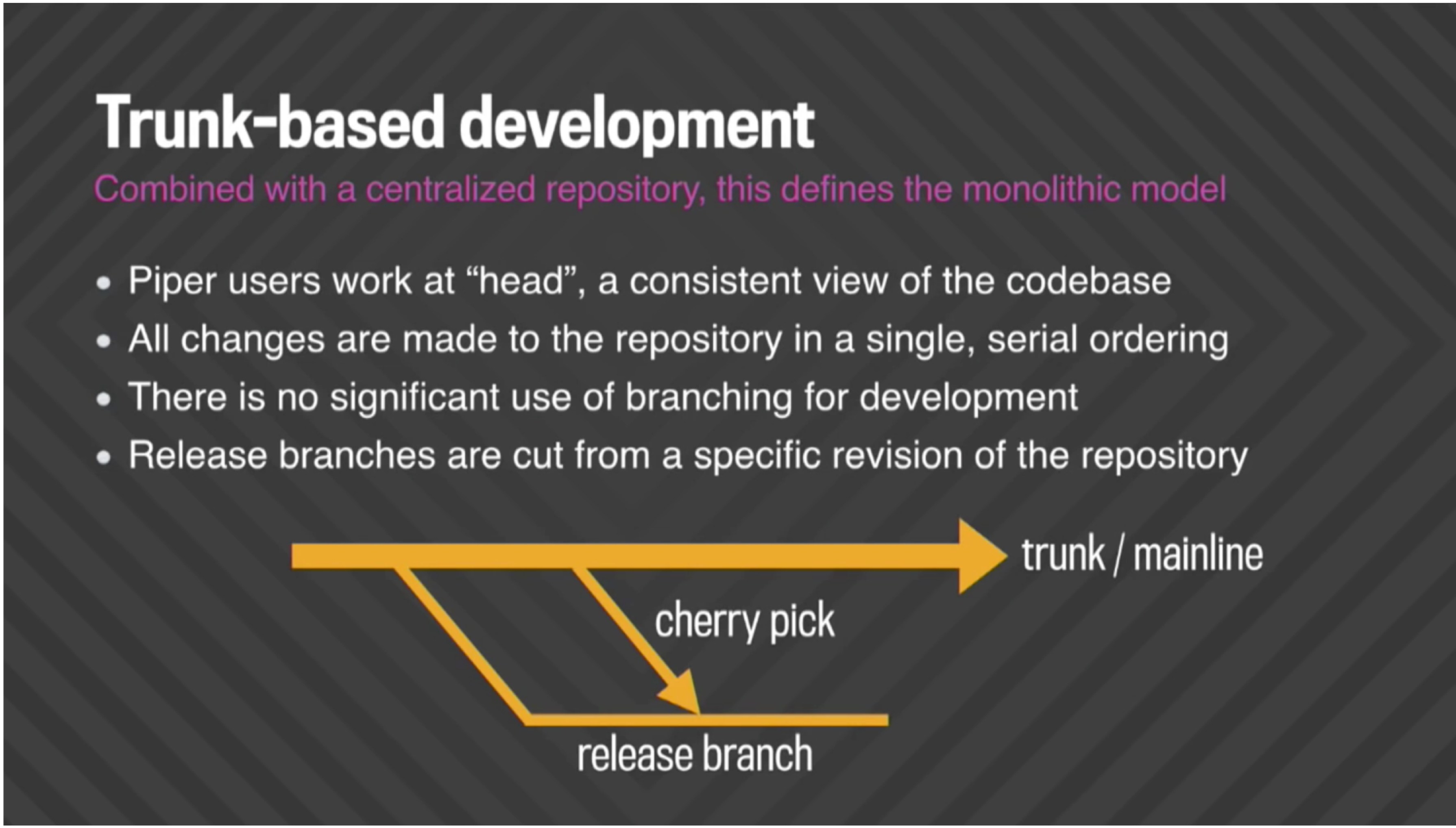
- Solution: redesign version control
  - Sparse checkouts → **10x faster clones and pulls**
  - Change the clone and pull commands to download only the commit metadata, while omitting all file changes (the bulk of the download)
  - When a user performs an operation that needs the contents of files (such as checkout), download the file contents on demand using existing memcache infrastructure



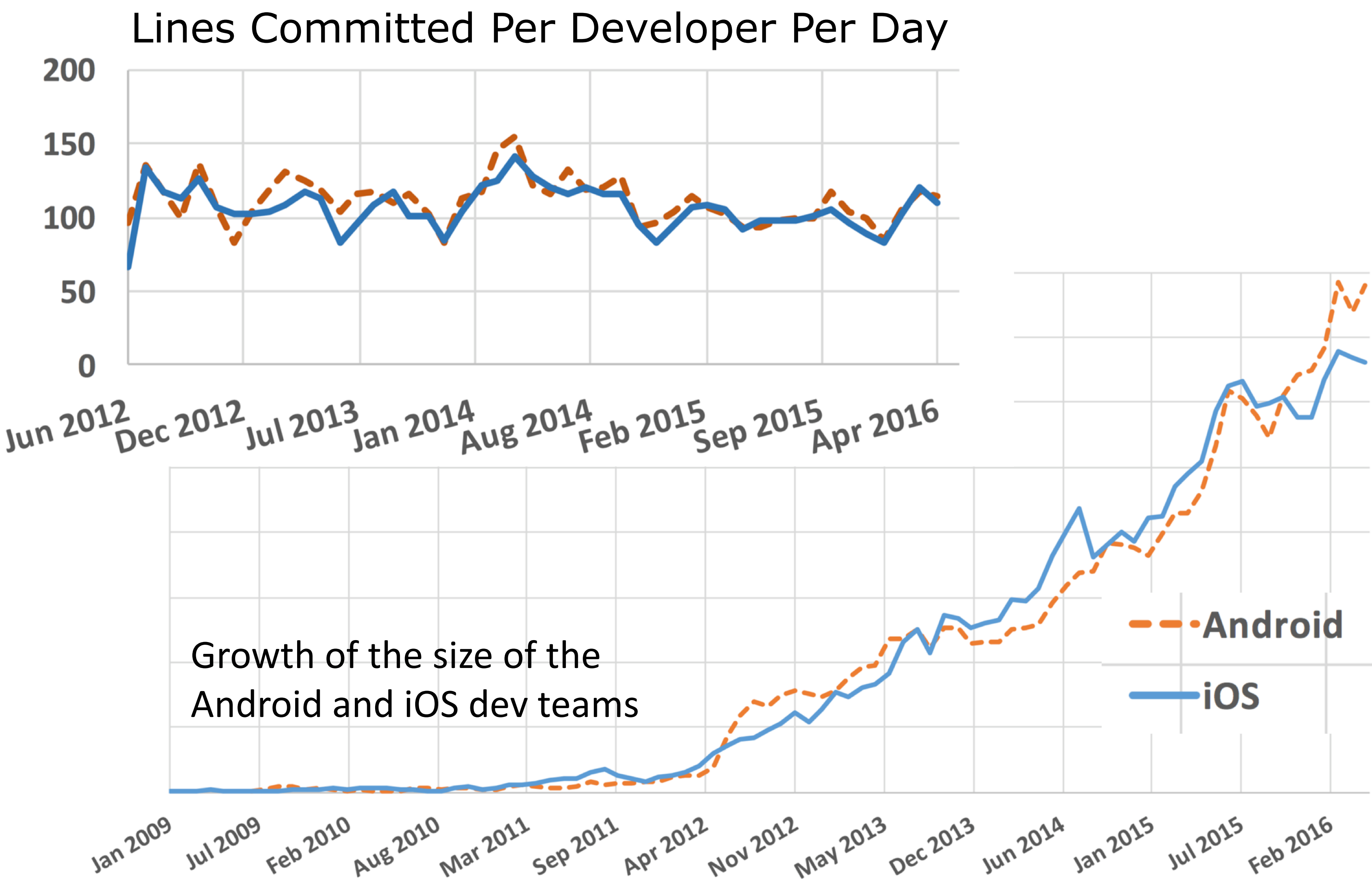
4. Monolithic repository



Monolithic repository – no major use of branches for development



Did it work? Yes. Sustained productivity at Facebook



MONOREPO VS MANY REPOS

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.

Qafoo
passion for software quality

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

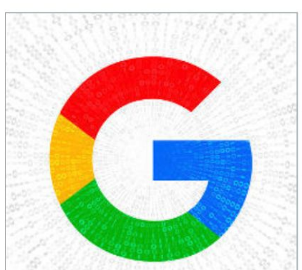
Home / Magazine Archive / July 2016 (Vol. 59, No. 7) / Why Google Stores Billions of Lines of Code in a Single... / Full Text

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

Key Insights

SIGN IN for Full Access

User Name

Password

» Forgot Password?

» Create an ACM Web Account

SIGN IN

ARTICLE CONTENTS:

institute for SOFTWARE RESEARCH

Advantages and Disadvantages of a Monolithic Repository

A case study at Google

Ciera Jaspan, Matthew Jorde,
Andrea Knight, Caitlin Sadowski,
Edward K. Smith, Collin Winter
Google
ciera,majorde,aknight,supertri,edwardsmith,
collinwinter@google.com

Emerson Murphy-Hill*
NC State University
emerson@csc.ncsu.edu

ABSTRACT

Monolithic source code repositories (repos) are used by several large tech companies, but little is known about their advantages or disadvantages compared to multiple per-project repos. This paper investigates the relative tradeoffs by utilizing a mixed-methods approach. Our primary contribution is a survey of engineers who have experience with both monolithic repos and multiple, per-project repos. This paper also backs up the claims made by these engineers with a large-scale analysis of developer tool logs. Our study finds that the visibility of the codebase is a significant advantage of a monolithic repo: it enables engineers to discover APIs to reuse, find examples for using an API, and automatically have dependent code updated as an API migrates to a new version. Engineers also appreciate the centralization of dependency management in the repo. In contrast, multiple-repository (multi-repo) systems afford engineers more flexibility to select their own toolchains and provide significant access control and stability benefits. In both cases, the related tooling is also a significant factor; engineers favor particular tools and are drawn to repo management systems that support their desired toolchain.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**;

1 INTRODUCTION

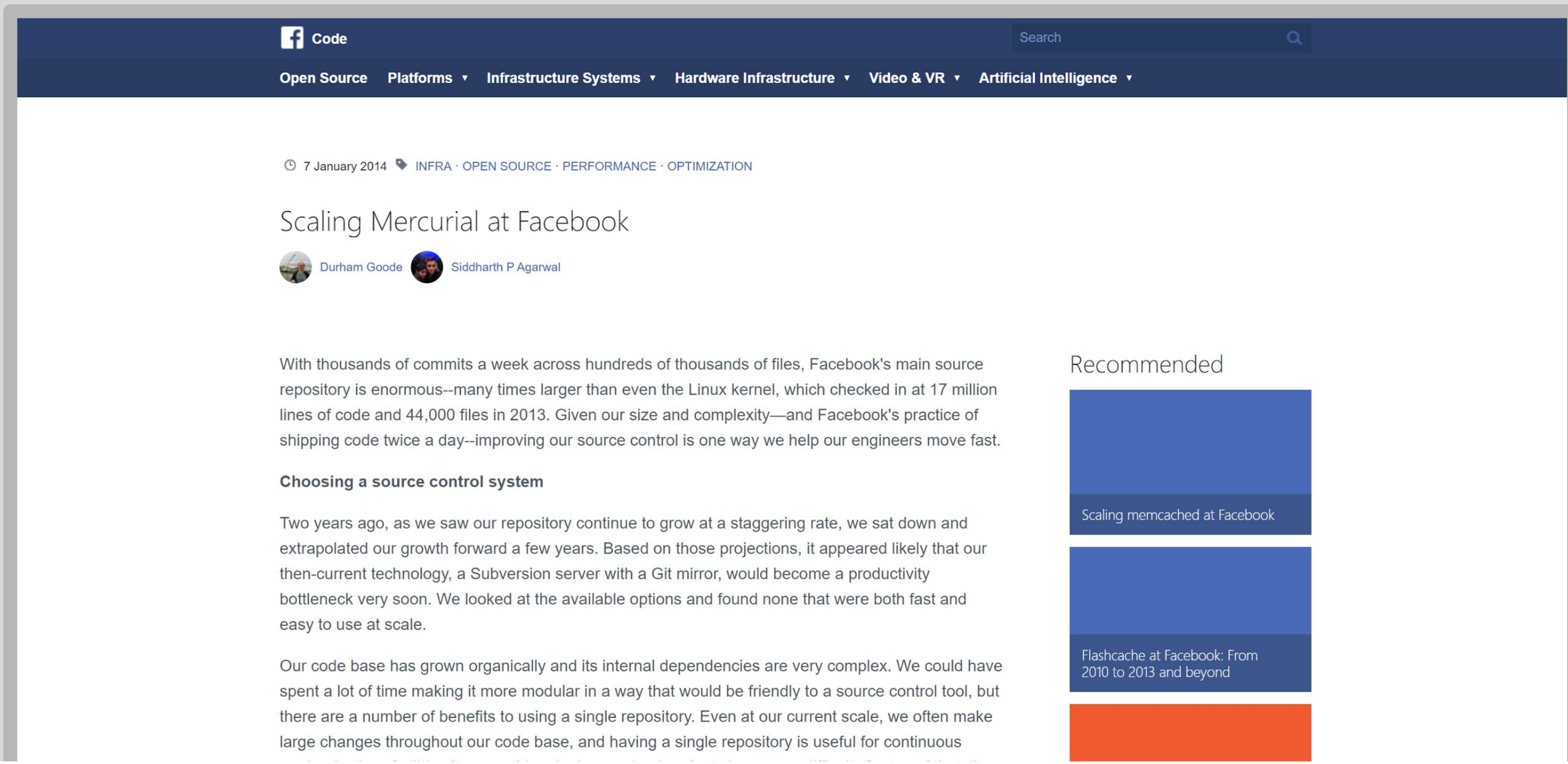
Companies today are producing more source code than ever before. Given the increasingly large codebases involved, it is worth examining the software engineering experience provided by the organization. Successfully organizing these dependencies and frameworks is crucial for development velocity.

One approach to scaling development practices is the monolithic repo, a model of source code organization where engineers have broad access to source code, a shared set of tooling, and a single set of common dependencies. This standardization and level of access is enabled by having a single, shared repo that stores the source code for all the projects in an organization. Several large software companies have already moved to this organizational model, including Facebook, Google, and Microsoft [10, 12, 17, 21]; however, there is little research addressing the possible advantages or disadvantages of such a model. Does broad access to source code let software engineers better understand APIs and libraries, or overwhelm engineers with use cases that aren't theirs? Do projects benefit from shared dependency versioning, or would engineers prefer more stability for their dependencies? How often do engineers take advantage of the workflows that monolithic repos enable? Do engineers prefer having consistent, shared toolchains or the flexibility of selecting a toolchain for their project?

In this paper, we investigate the experience of engineers working within a monolithic repo and the tradeoffs between using a monolithic repo and a multi-repo codebase. Specifically, this paper seeks to answer two research questions:

(1) What do developers perceive as the benefits and drawbacks to working in a monolithic versus multi-repo environment?
(2) To what extent do developers make use of the unique advantages that monolithic repos provide?

Monorepos in industry



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 3213 in 1230

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.

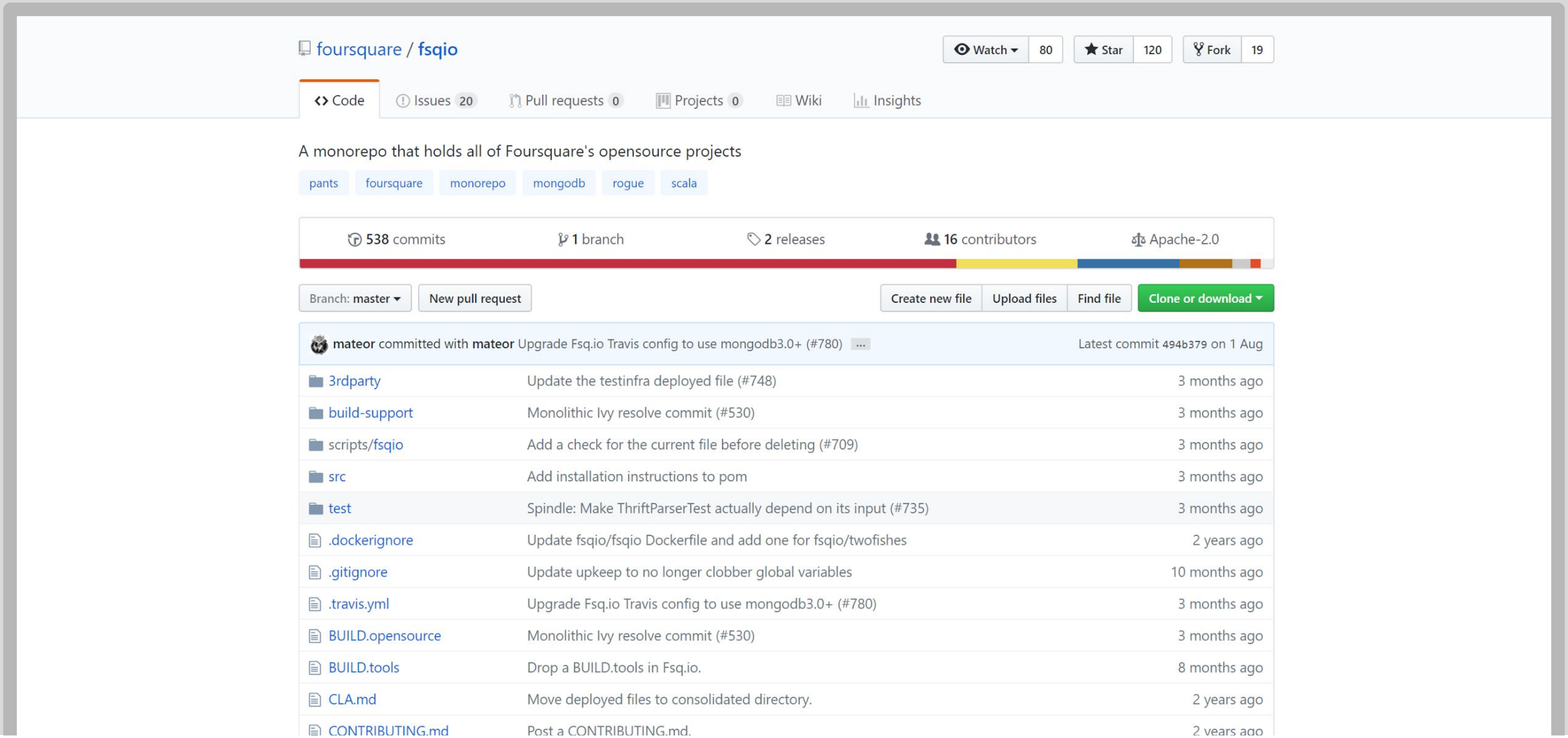
Email Address

Subscribe! Unsubscribe

Back to

Monorepos in open-source

foresquare public monorepo



2016 talk by FABIEN POTENCIER

Monorepos in open-source

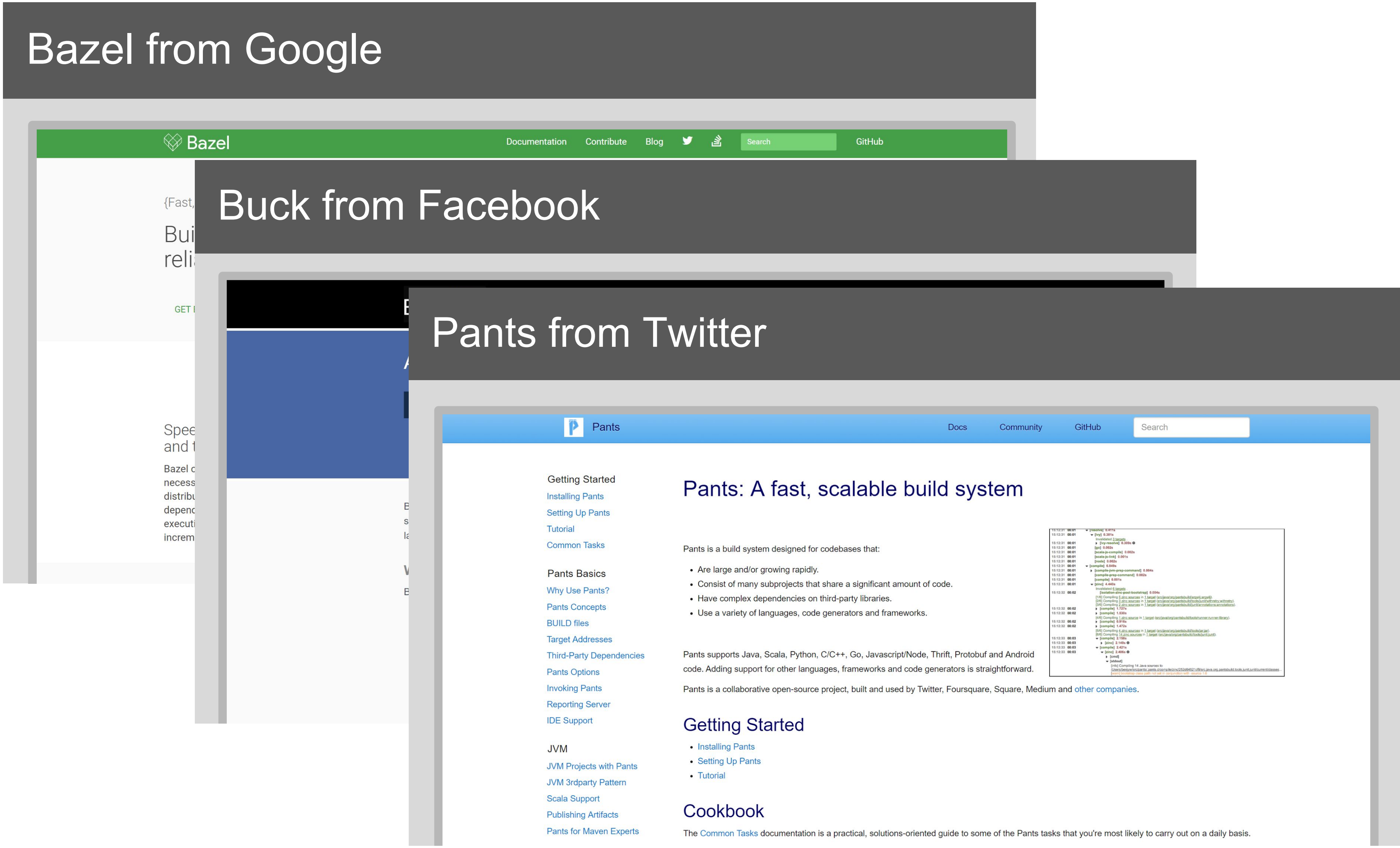
The Symfony monorepo

43 projects, **25 000** commits, and **400 000** LOC

https://github.com/symfony/symfony

- Bridge/
  - 5 sub-projects
- Bundle/
  - 5 sub-projects
- Component/
  - 33 independent sub-projects like Asset, Cache, CssSelector, Finder, Form, HttpKernel, Ldap, Routing, Security, Serializer, Templating, Translation, Yaml, ...

Common build system



Some advantages of monorepos

High Discoverability For Developers

- Developers can read and explore the whole codebase
- `grep`, IDEs and other tools can search the whole codebase
- IDEs can offer auto-completion for the whole codebase
- Code Browsers can links between all artifacts in the codebase

Code-Reuse is cheap

Almost zero cost in introducing a new library

- Extract library code into a new directory/component
- Use library in other components
- Profit!

Refactorings in one commit

Allow large scale refactorings with one single, atomic, history-preserving commit

- Extract Library/Component
- Rename Functions/Methods/Components
- Housekeeping (phpcs-fixer, Namespacing, ...)

Another refactoring example

- Make large backward incompatible changes easily... especially if they span different parts of the project
- For example, old APIs can be removed with confidence
  - Change an API endpoint code **and** all its usages in **all** projects in **one** pull request

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

Some downsides

- Require collective responsibility for team and developers
- Require trunk-based development
  - Feature toggles are technical debt (recall financial services example)
- Force you to have only one version of everything
- Scalability requirements for the repository
- Can be hard to deal with updates around things like security issues
- Build and test bloat without very smart build system
- Slow VCS without very smart system
- Permissions?

Summary

- Software development at scale requires a lot of infrastructure
  - Version control, build managers, testing, continuous integration, 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