Continuous Integration and Cloud Resources

Advanced Software Engineering

Spring 2023

Software Testing in the News

Aviation

After Alaska Airlines planes bump runway while taking off from Seattle, a scramble to 'pull the plug'

By Dominic Gates, The Seattle Times
Updated: February 20, 2023
Published: February 20, 2023

"That morning, a software bug in an update to the DynamicSource tool caused it to provide seriously undervalued weights for the airplanes.

The Alaska 737 captain said the data was on the order of 20,000 to 30,000 pounds light. With the total weight of those jets at 150,000 to 170,000 pounds, the error was enough to skew the engine thrust and speed settings.

Both planes headed down the runway with less power and at lower speed than they should have. And with the jets judged lighter than they actually were, the pilots rotated too early

Both the Max 9 and 737-900ER have long passenger cabins, which makes them more vulnerable to a tail strike when the nose comes up too soon." …

… "A quick interim fix proved easy: When operations staff turned off the automatic uplink of the data to the aircraft and switched to manual requests "we didn't have the bug anymore."

Peyton said his team also checked the integrity of the calculation itself before lifting the stoppage. All that was accomplished in 20 minutes.

The software code was permanently repaired about five hours later.

Peyton added that even though the update to the DynamicSource software had been tested over an extended period, the bug was missed because it only presented when many aircraft at the same time were using the system.

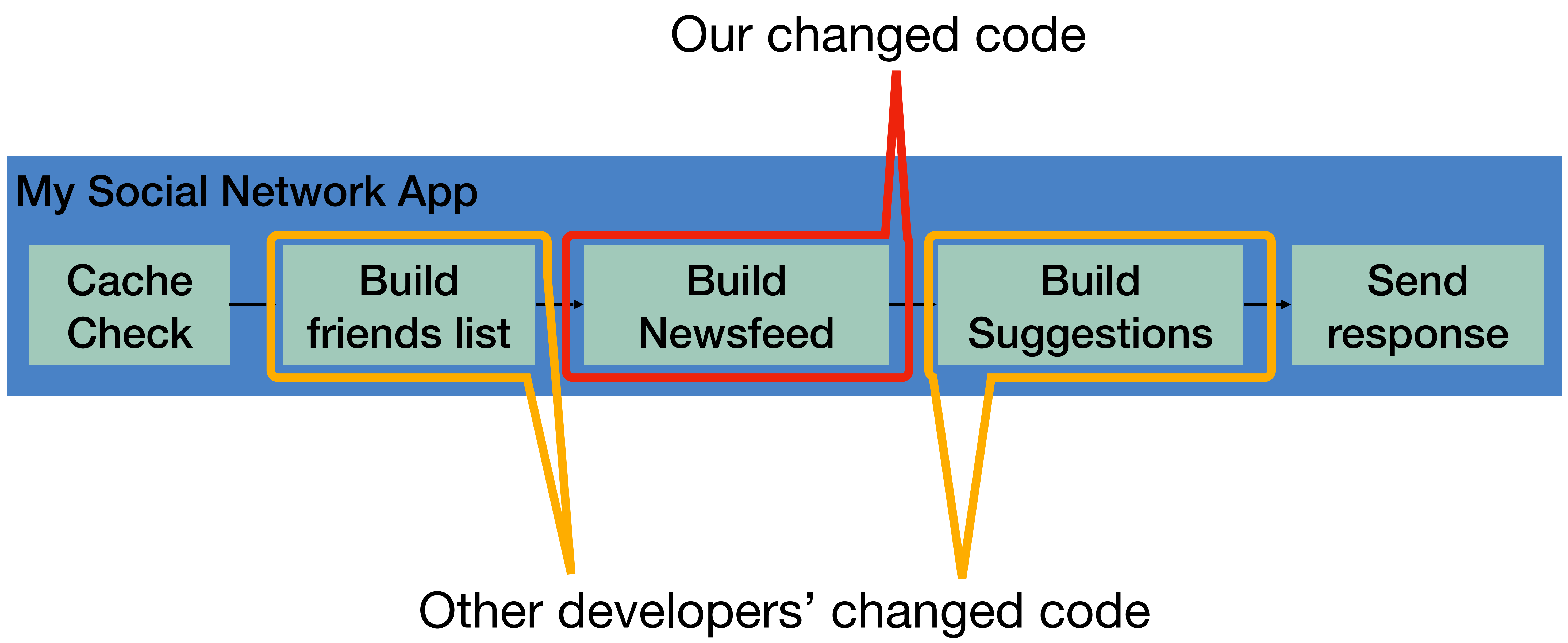
Subsequently, a test of the software under high demand was developed."

https://www.adn.com/alaska-news/aviation/2023/02/20/after-alaska-airlines-planes-bump-runway-a-scramble-to-pull-the-plug/

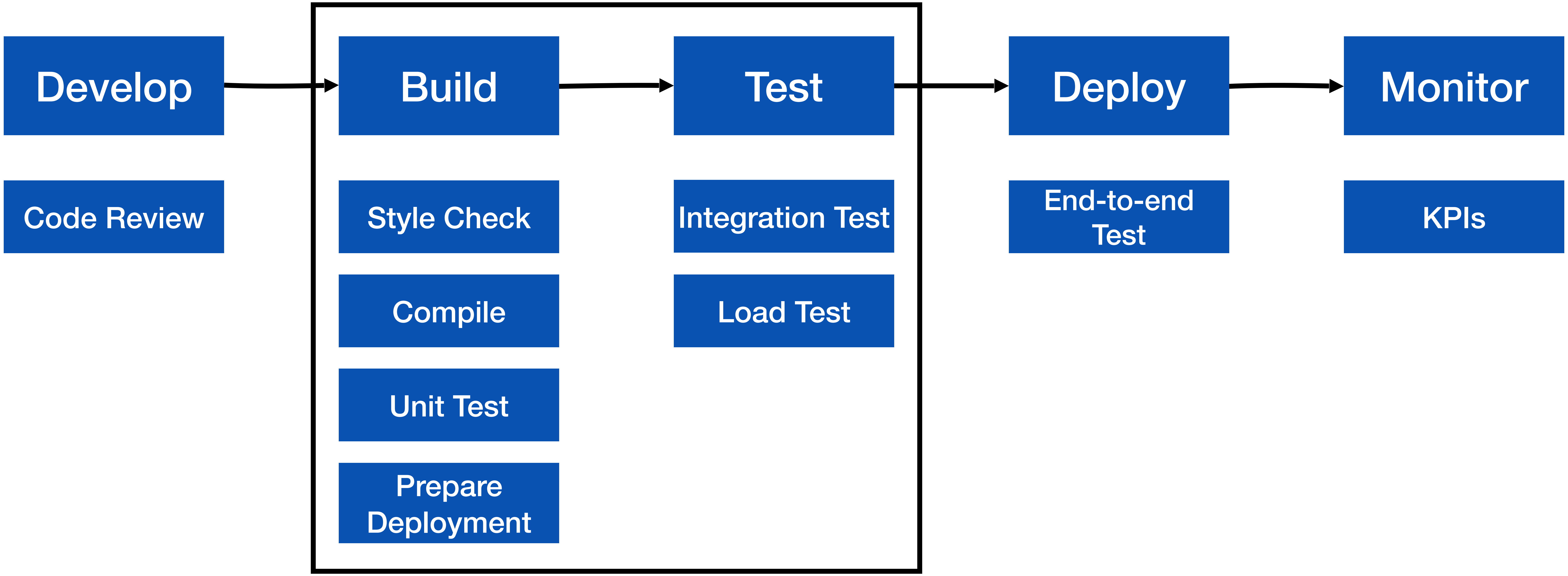
Continuous Integration

Motivation

- Our systems involve many components, some of which might even be in different version control repositories
- How does a developer get feedback on their (local) change?



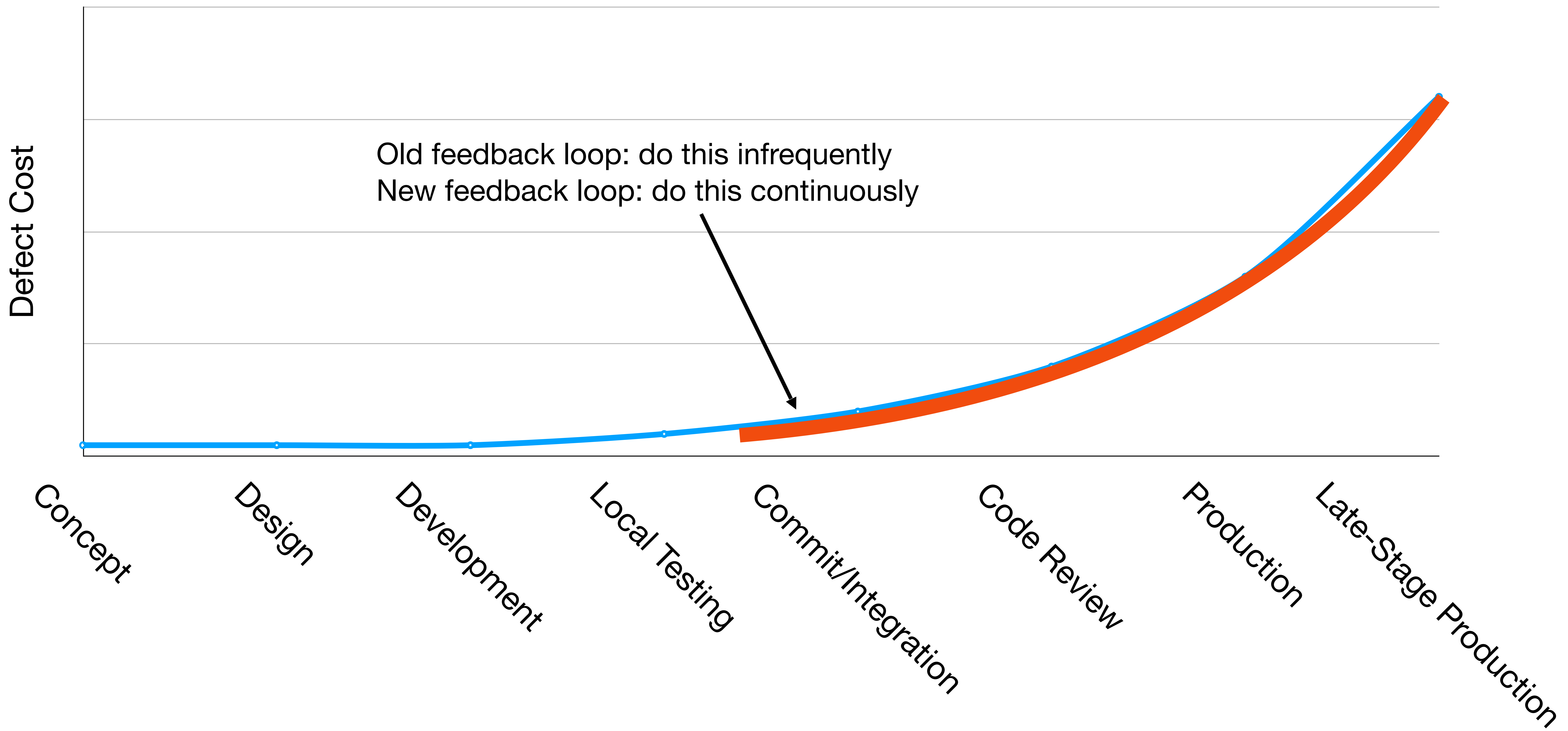
Continuous Integration is a Software Pipeline



Automate this centrally, provide a central record of results

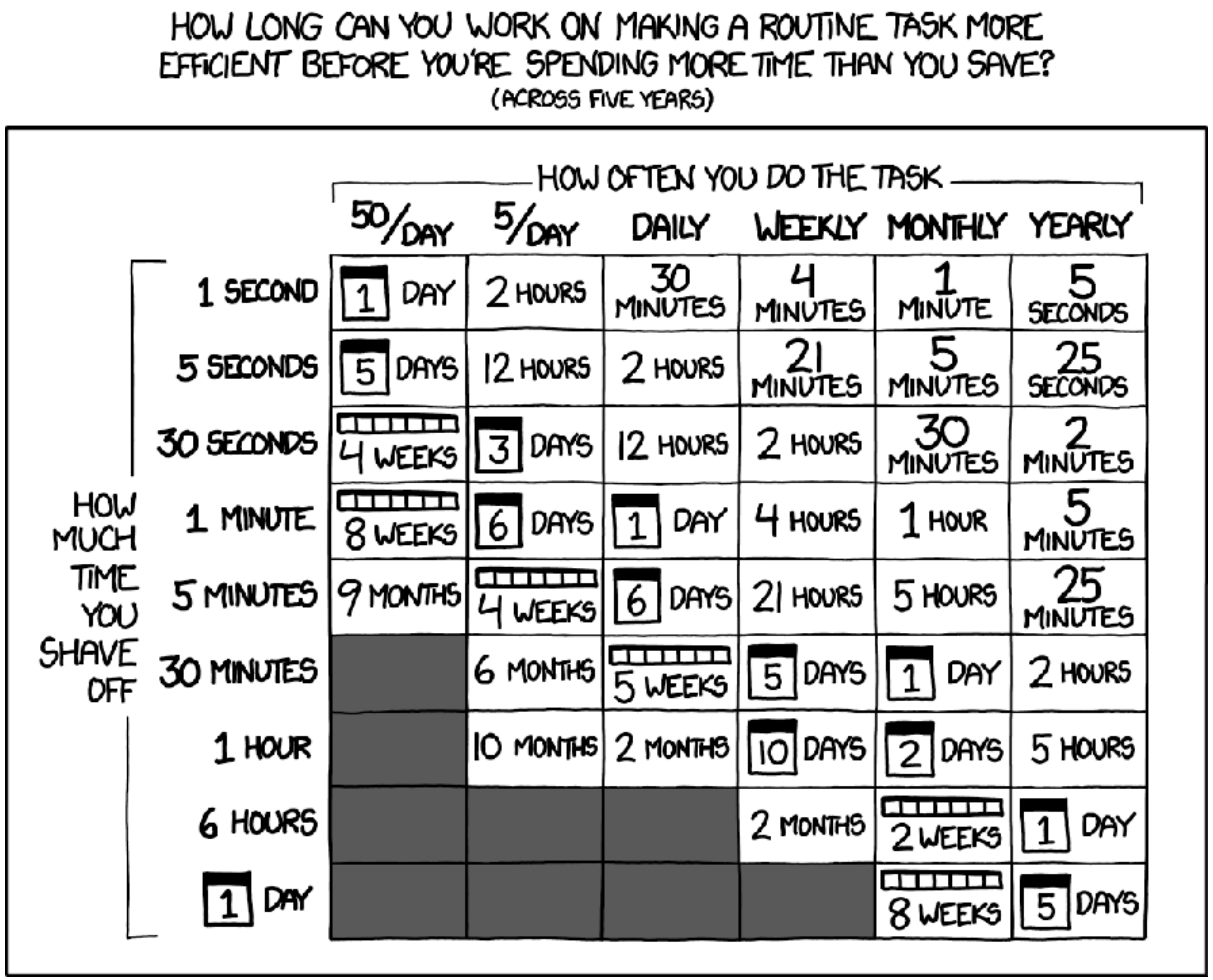
Agile Values Fast Quality Feedback Loops

Faster feedback = lower cost to fix bugs



The Power of Automating Feedback Loops

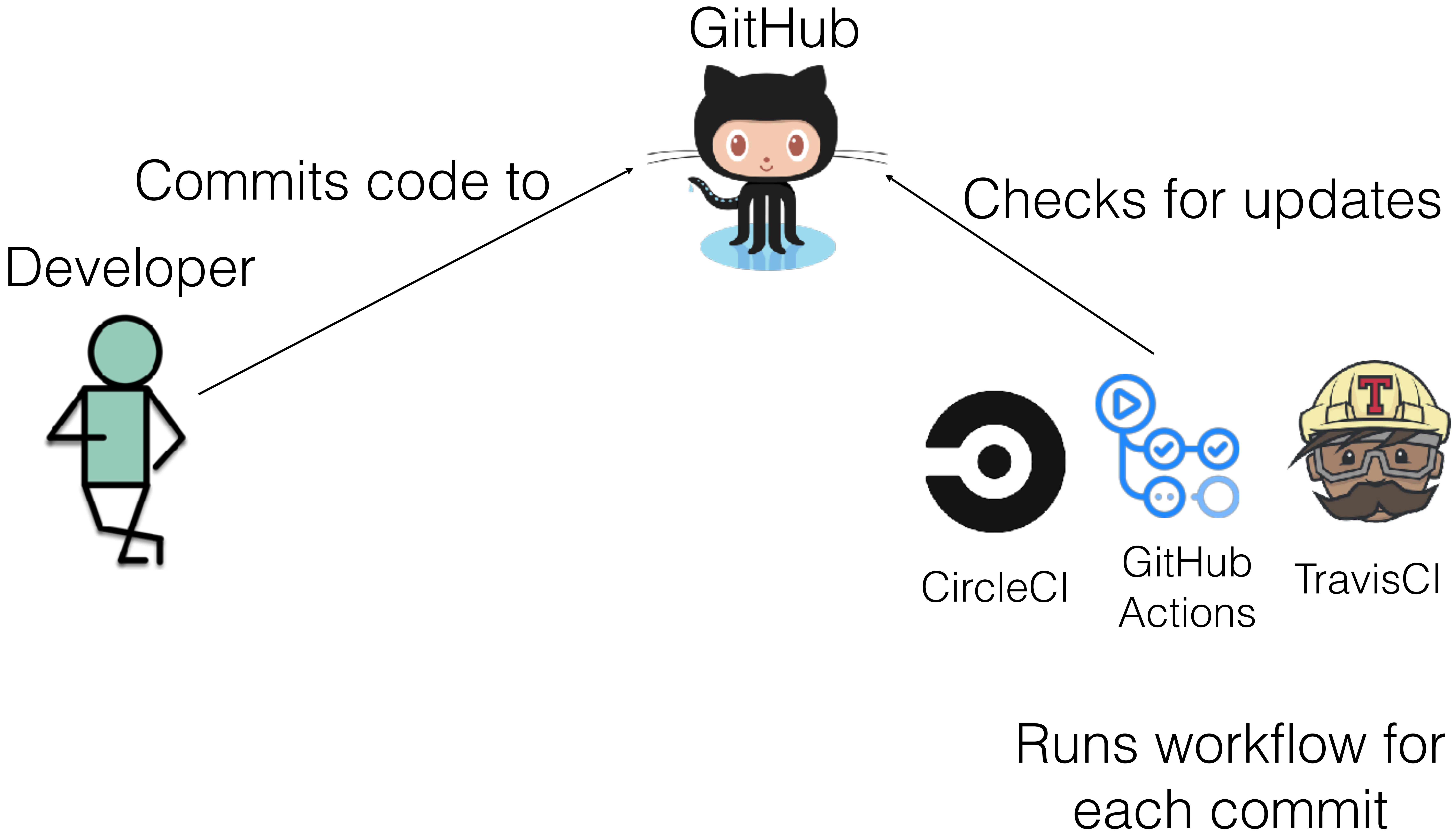
Consider tasks that are done by *dozens* of developers



https://xkcd.com/1205/

Continuous Integration in Practice

Small scale, with a service like CircleCI, GitHub Actions or TravisCI



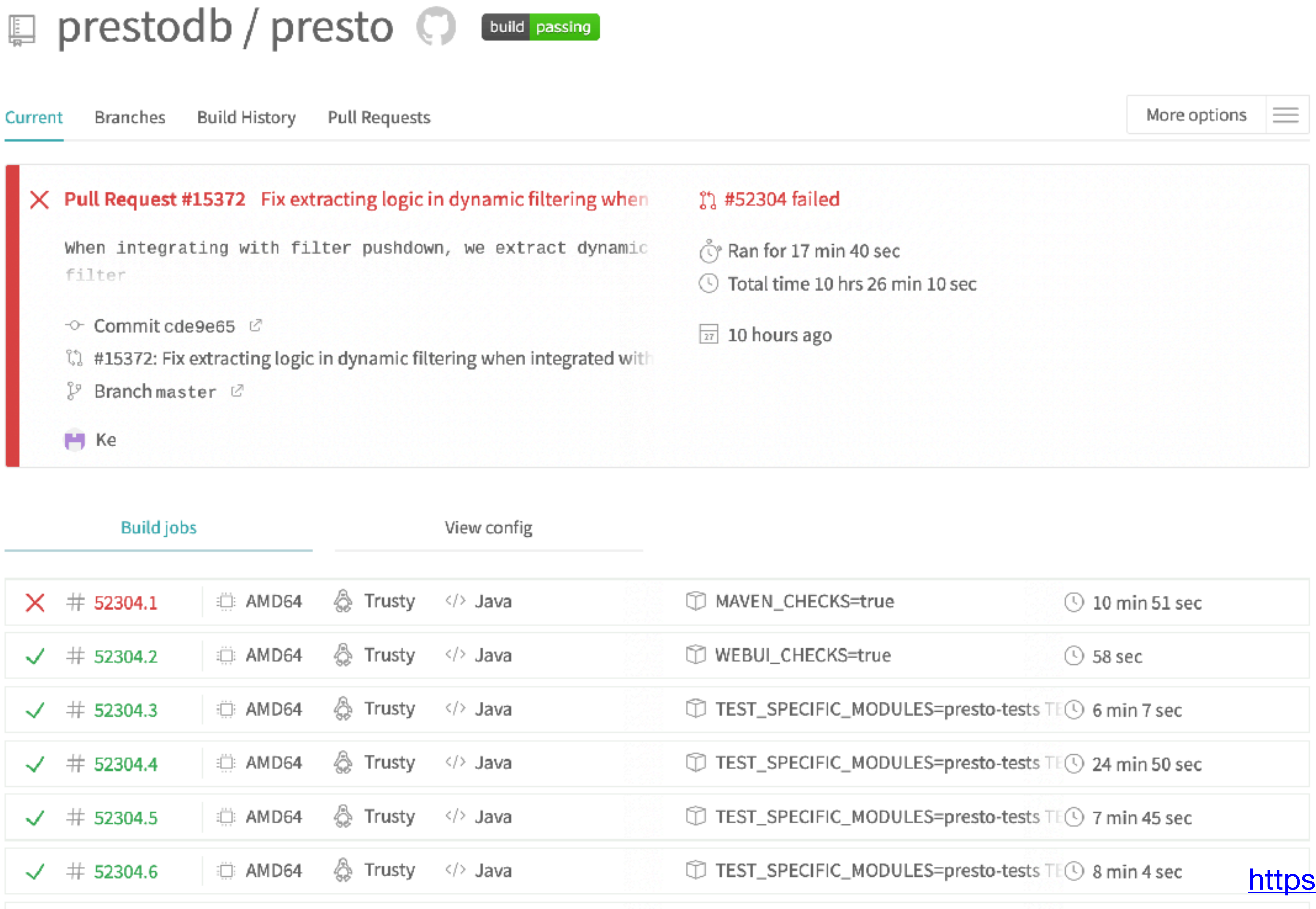
Brainstorm: What could we check in CI for Google Docs?

Consider all scopes of testing, from unit to system-level

- Brainstorming notes:
- Run unit tests
- Run some localization tests
- Validate infrastructure deployment
- Do regression testing on user scenarios - ensure that old/new look the same
- Compress images, other artifacts before deployment
- Update documentation, internal screenshots
- Build software, lint, etc
- Check interoperability with other/existing packages
- Accessibility testing - ensure that components can be accessed through screen readers
- Check/gate on test quality metrics
- Do security audit
- Design review? (Code review fits somewhere in the workflow)

Example CI Pipeline

Open source project: PrestoDB



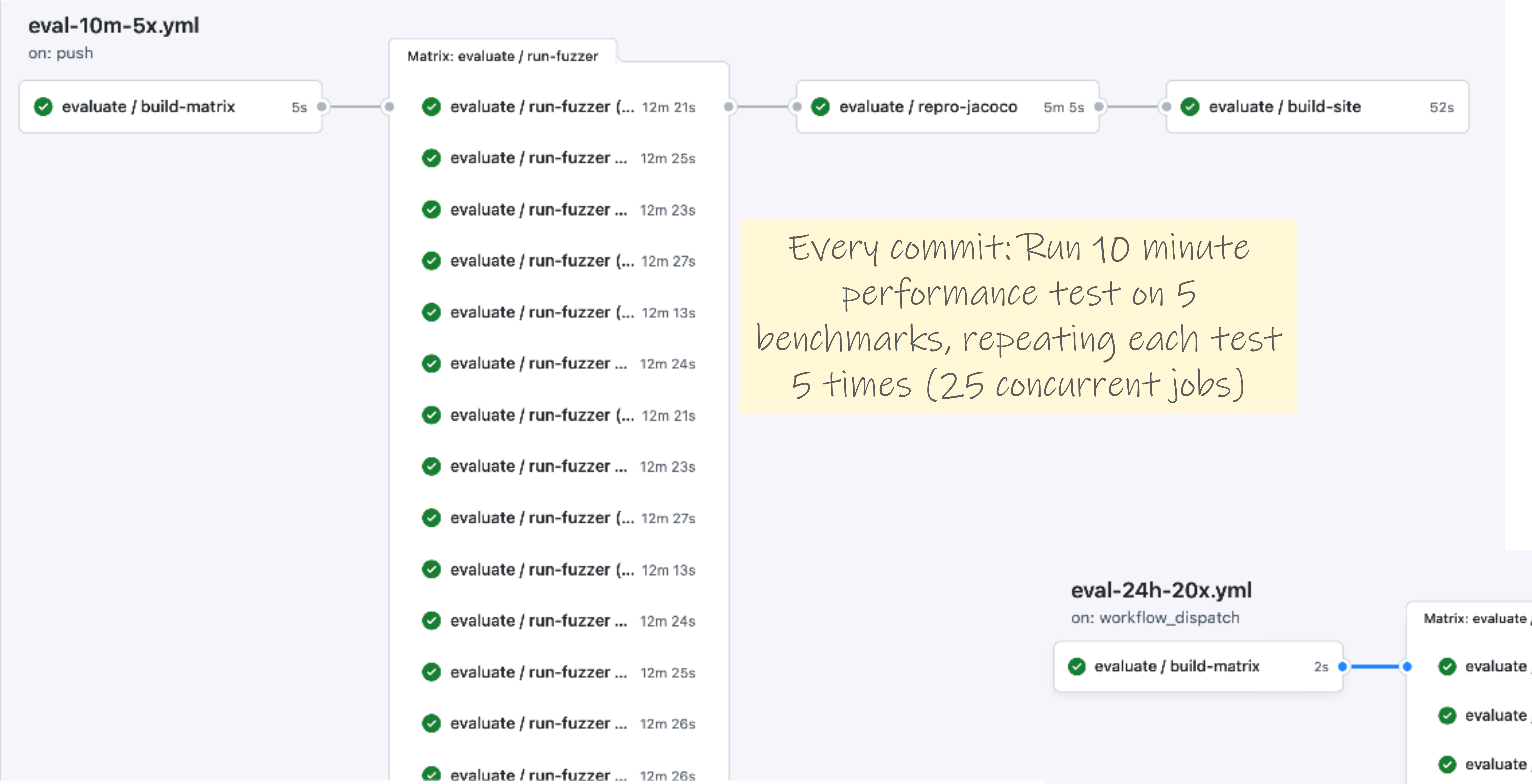
https://travis-ci.com/github/prestodb/presto

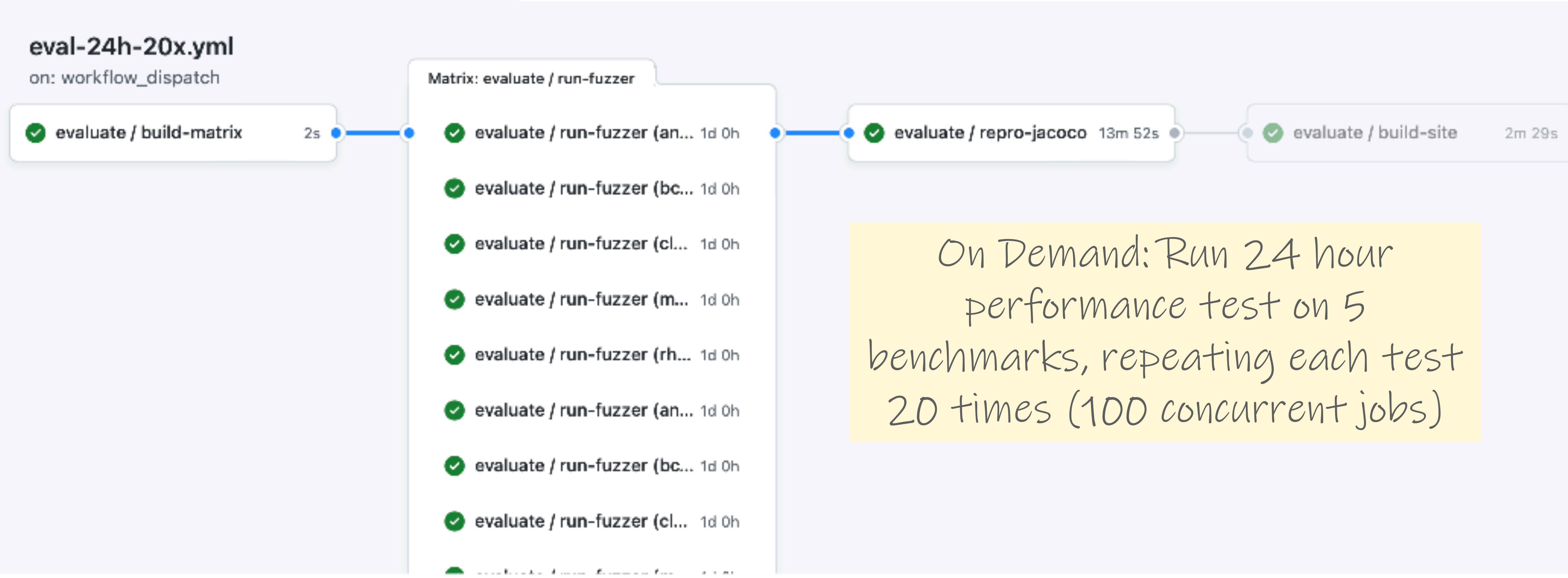
Use Scalable Cloud Resources for CI

Example: Developing a Fuzzer

- "Fuzzers" are automated testing systems that aim to automatically generate inputs to programs that cover code and reveal bugs
- Fuzzers are non-deterministic: to evaluate with confidence, need repeated, long-running trials
- Evaluating fuzzers is time consuming, determining which changes impact performance is confusing
- How to run experiments in the cloud?

CI Pipelines Automate Otherwise Manual Testing



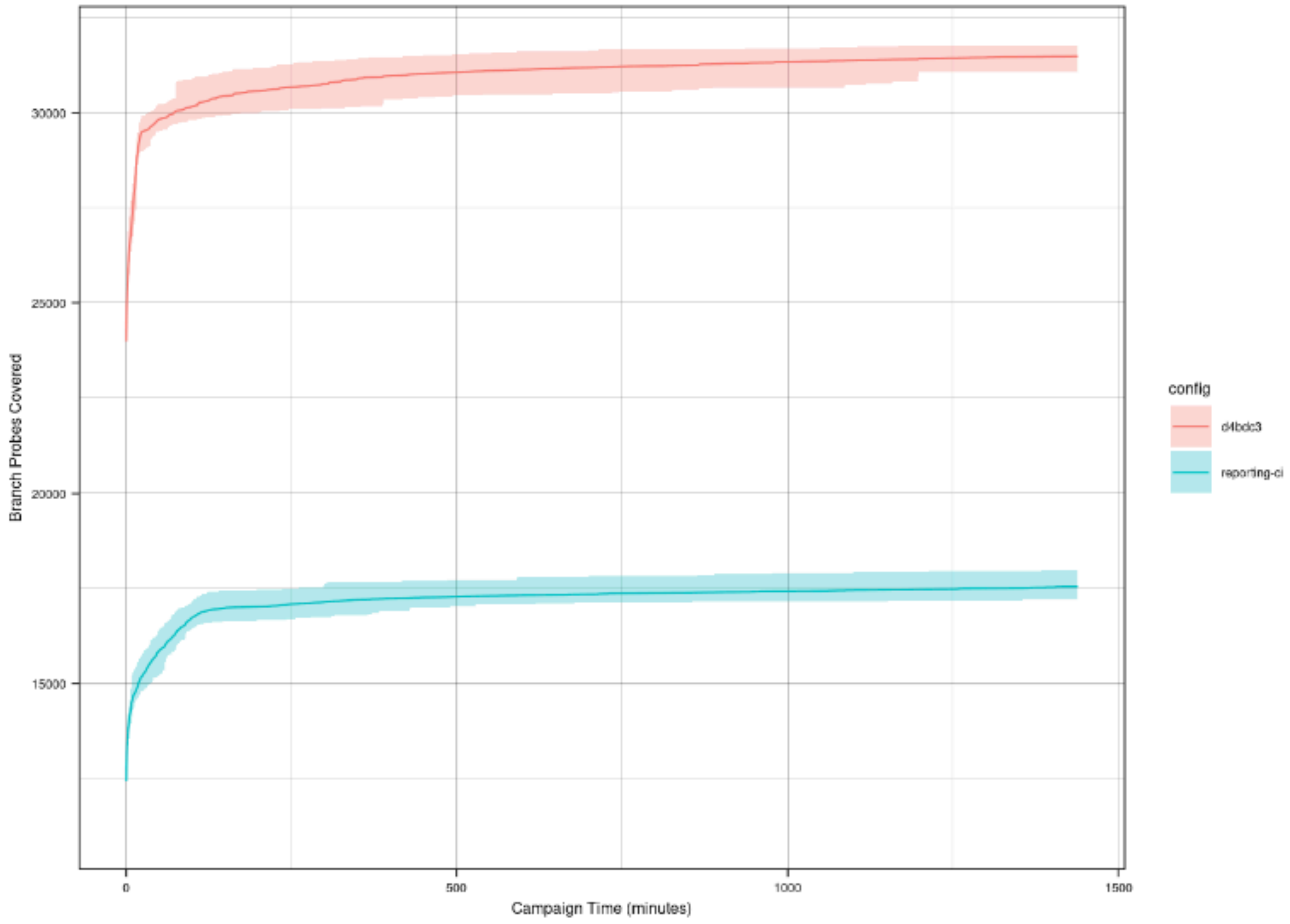


https://github.com/neu-se/CONFETTI/actions

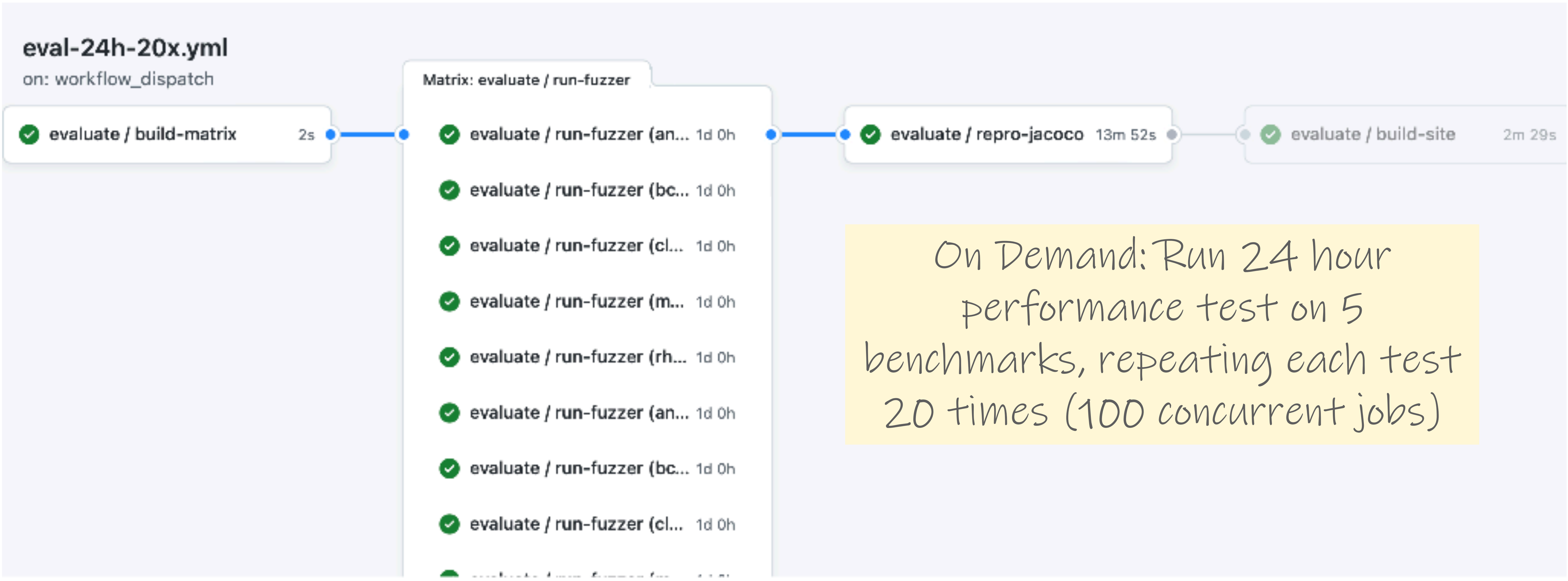
CI Pipelines Automate Otherwise Manual Testing

closure

Branch Probes Over Time



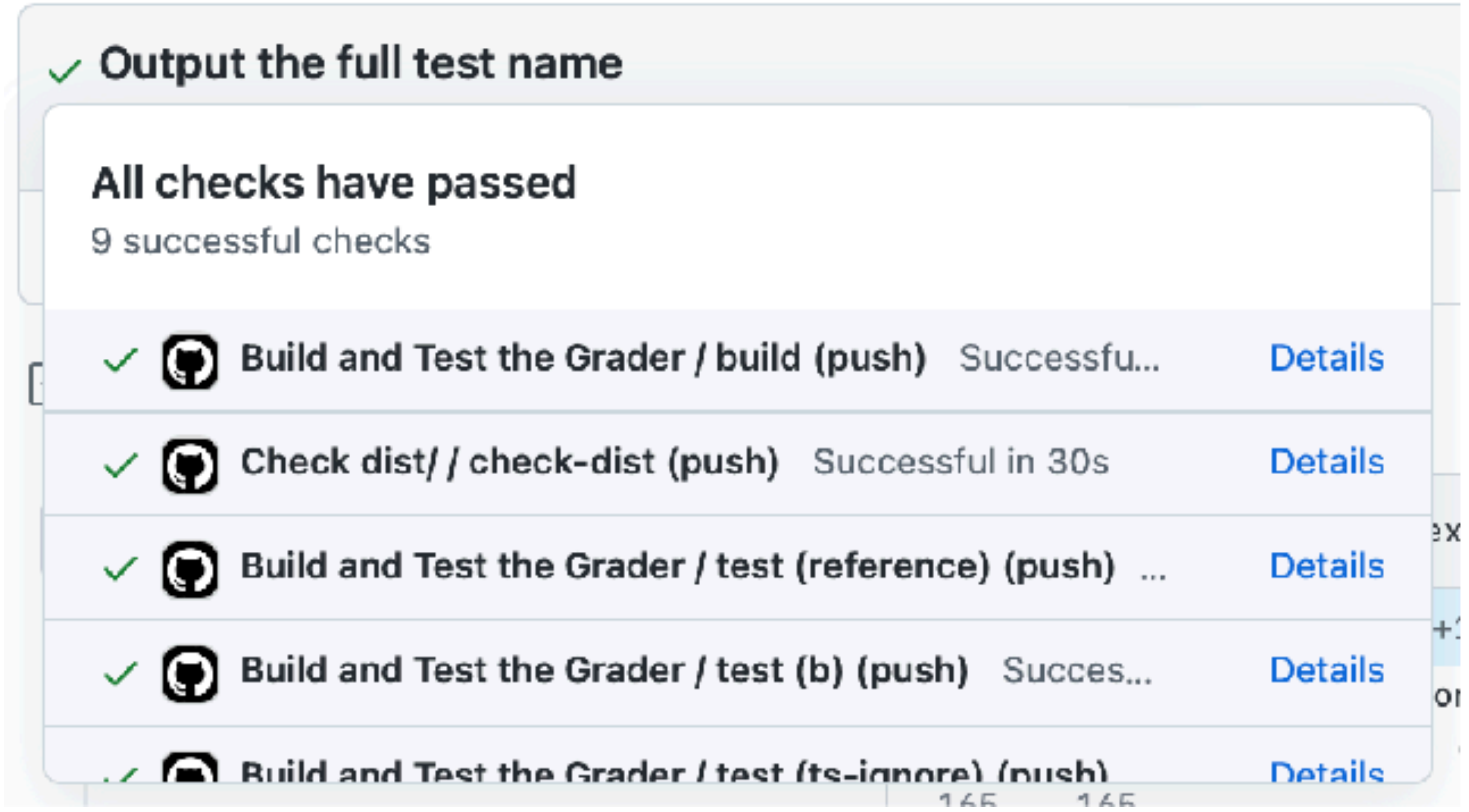
Download this graph as PDF

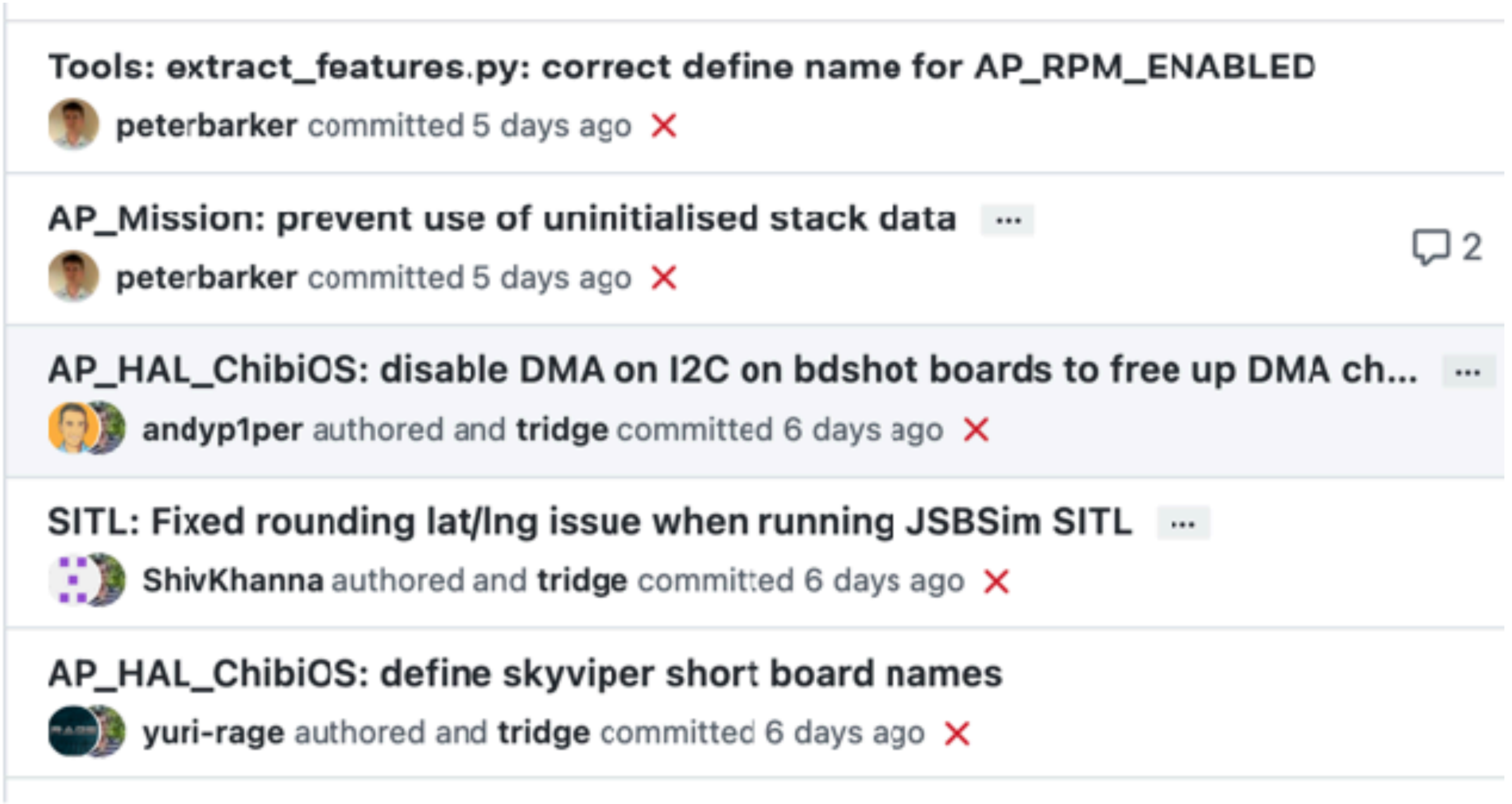


https://github.com/neu-se/CONFETTI/actions

Attributes of Effective CI Processes

- Infrastructure:
  - CI should be repeatable (deterministic)
  - CI should be *fast*, providing feedback within minutes or hours
- Policies:
  - Do not allow builds to remain broken for a long time
  - CI should run for every change
  - CI should not completely replace pre-commit testing





Brainstorm: Why might CI not be repeatable?

- Flaky tests
- If dependency server is down
- The infrastructure that we are using is under-provisioned
- Generally unmaintained - some dependencies may have changed, requiring more complex upgrade
- If not everything is automated

Challenges and Solutions for Repeatable Builds

- Which commands to run to produce an executable? (build systems)
- How to link third-party libraries? (dependency managers)
- How to specify system-level software requirements? (containers)
- How to specify infrastructure requirements? (Infrastructure as code)

Build Systems Orchestrate Software Engineering Tasks

Early build tools (e.g. “make”) are scripting tools with special support for commands that transform “source files” to “target files”

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Build Systems Orchestrate Software Engineering Tasks

“Orchestrate” -> Execute in the right order, ideally with concurrency

- Example tasks:
    - Installing dependencies
    - Compiling the code
    - Running static analysis
    - Generating documentation
    - Running tests
    - Creating artifacts for customers
    - Deploying Code
- Example build systems: xMake, ant, maven, gradle, npm…

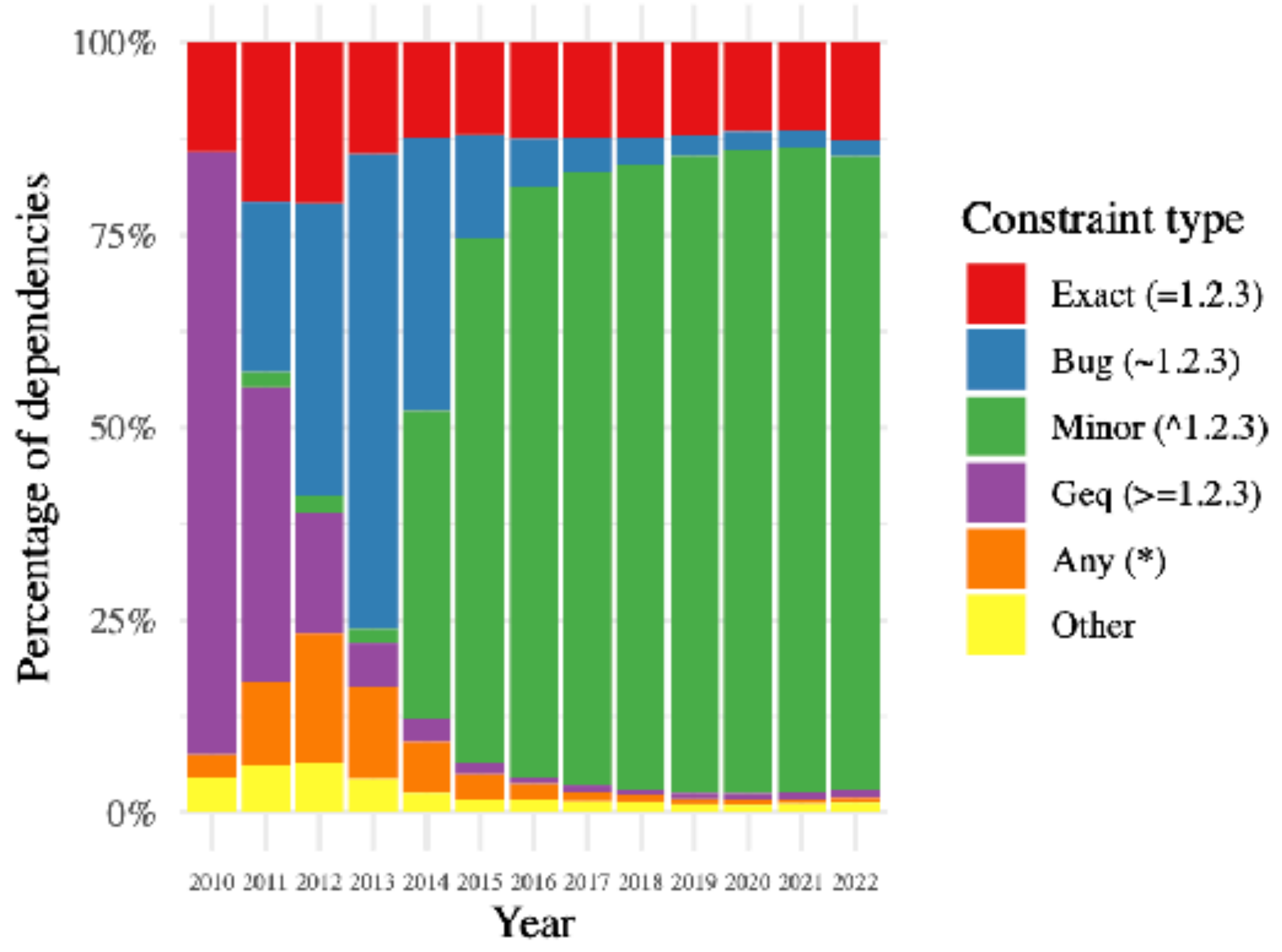
Dependency Managers Organize External Dependencies

- Addresses this problem: “Before you compile this code, install commons-lang from the apache website”
- Declare a dependency using coordinates (unique ID of a package plus version)
- Packages are archived in common repositories; fetched/linked by dependency manager
- Dependency managers handle transitive dependencies 🐉
- Examples: Maven, NPM, pip, cargo, apt

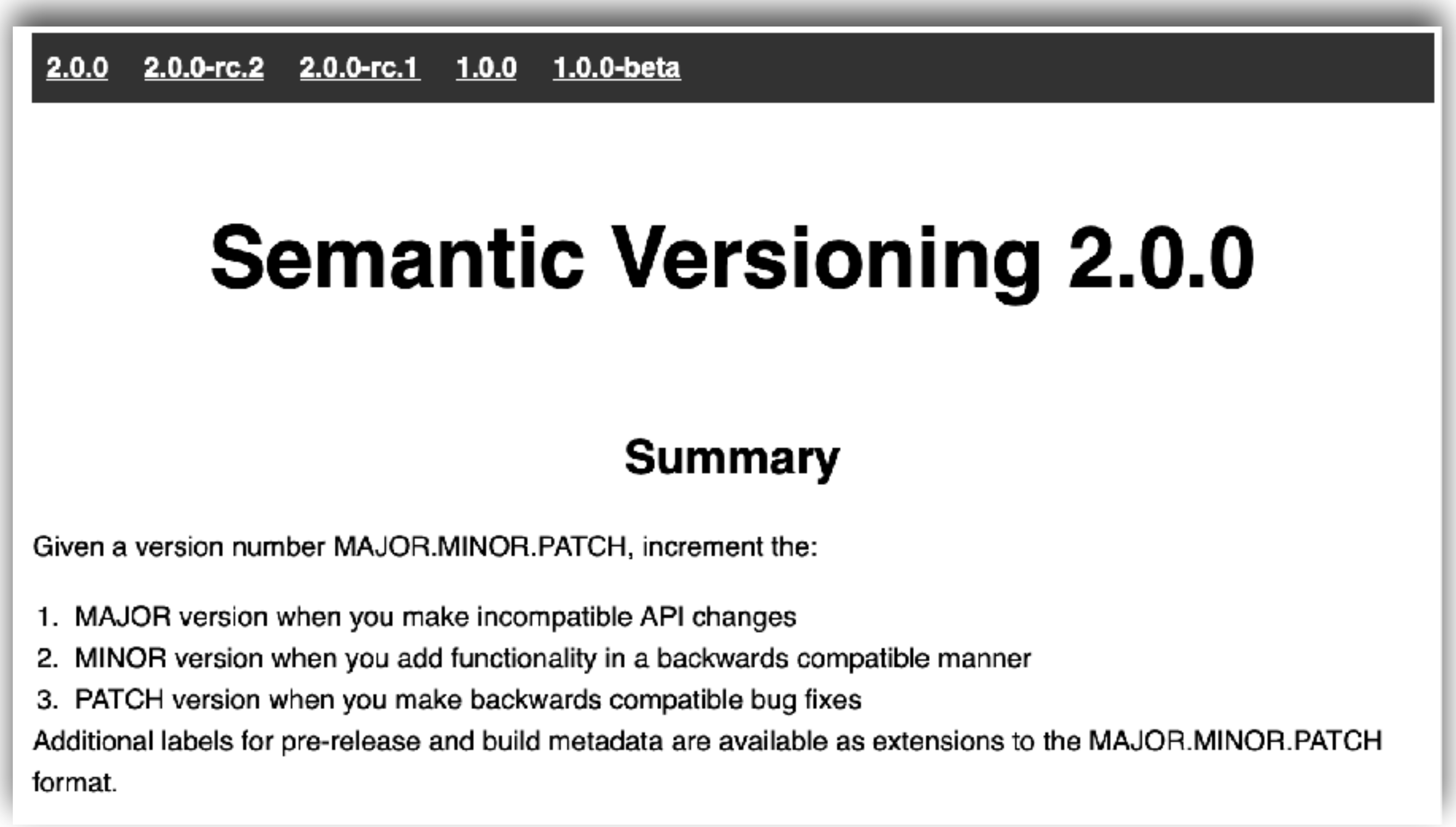
Specify and Depend on Package Versions with Care

Semantic Versioning is Often Expected

- Library maintainers expected to indicate breaking changes with version numbers
- Dependency consumers can specify *constraints* on versions (e.g. accept 2.0.x)



Distribution of dependencies of all packages in NPM over time (2023, Pinckney et al)



2.0.0 2.0.0-rc.2 2.0.0-rc.1 1.0.0 1.0.0-beta

Semantic Versioning 2.0.0

Summary

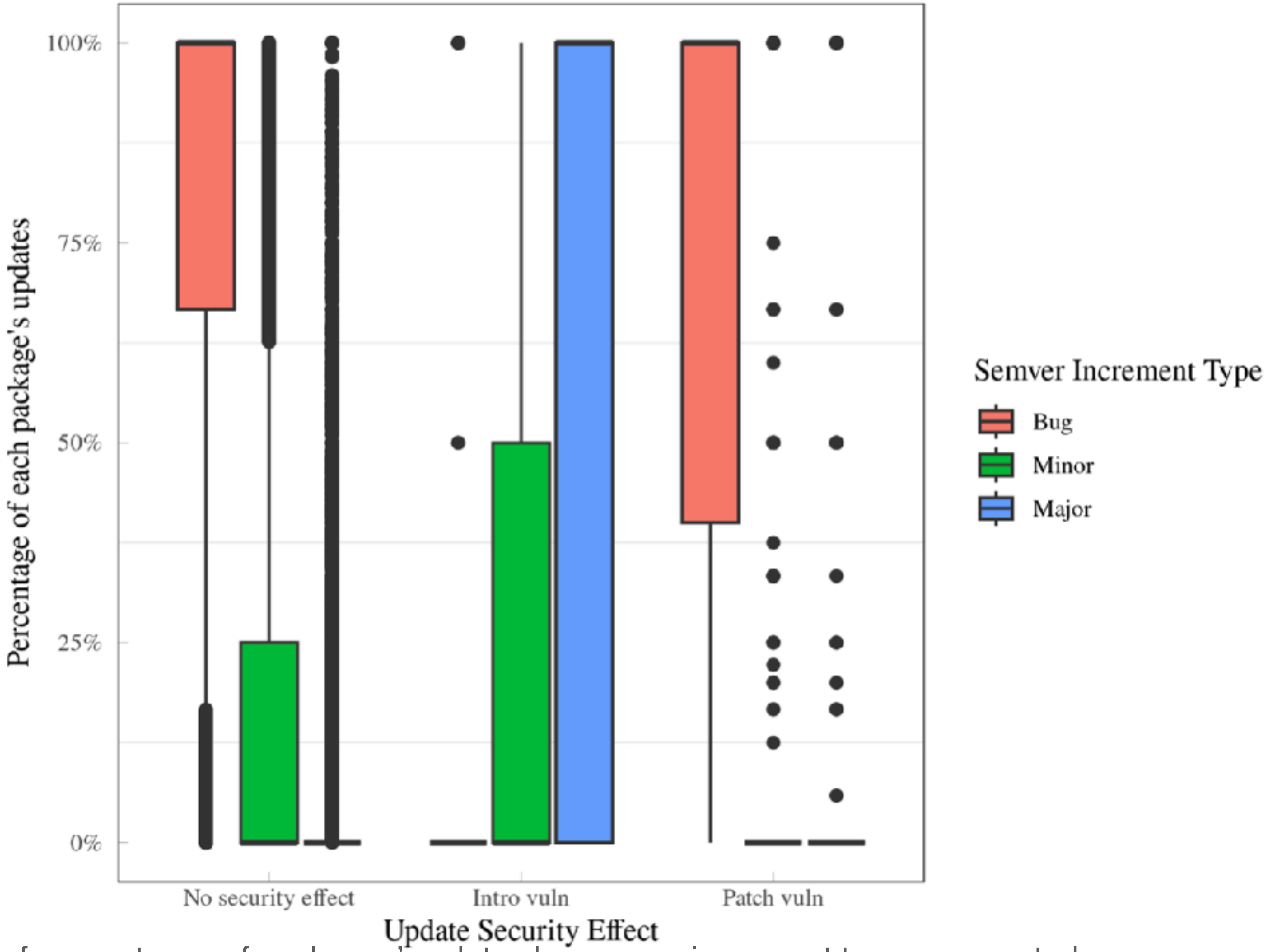
Given a version number MAJOR.MINOR.PATCH, increment the:

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

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Few Bug-Fix Updates Create Vulnerabilities

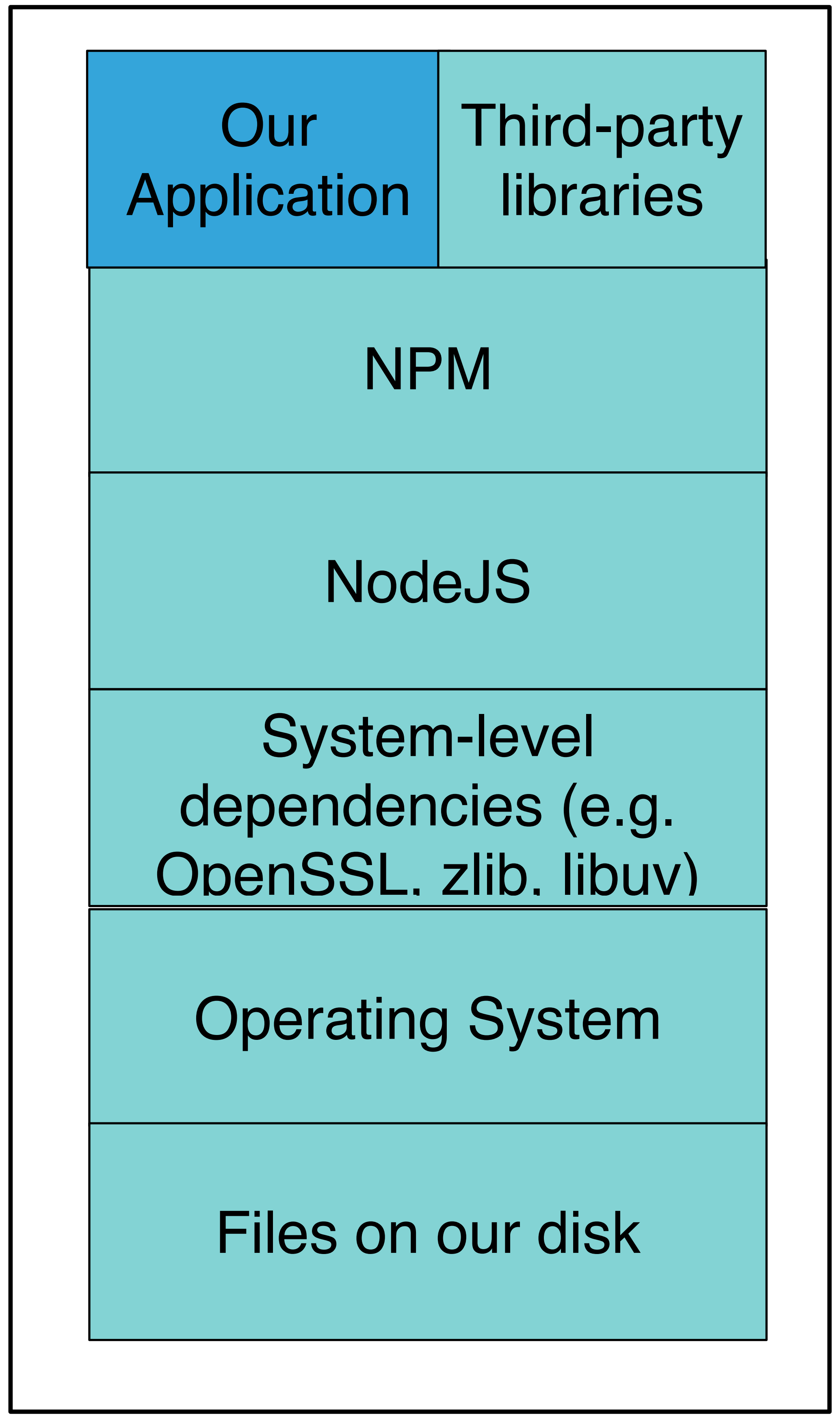
(Most vulnerabilities are *patched* in them!)



Distribution of percentages of packages' updates by semver increment type, segmented across security effects. Within each security effect the percentages across semver increment types are normalized. (2023, Pinckney et al)

Containers Include System-Level Dependencies

- Common problems:
  - Incorrect or conflicting version of system dependency
  - Different OS with files in the “wrong” place
  - Dependencies no longer available
- Containers can include the entire stack



“Container image”

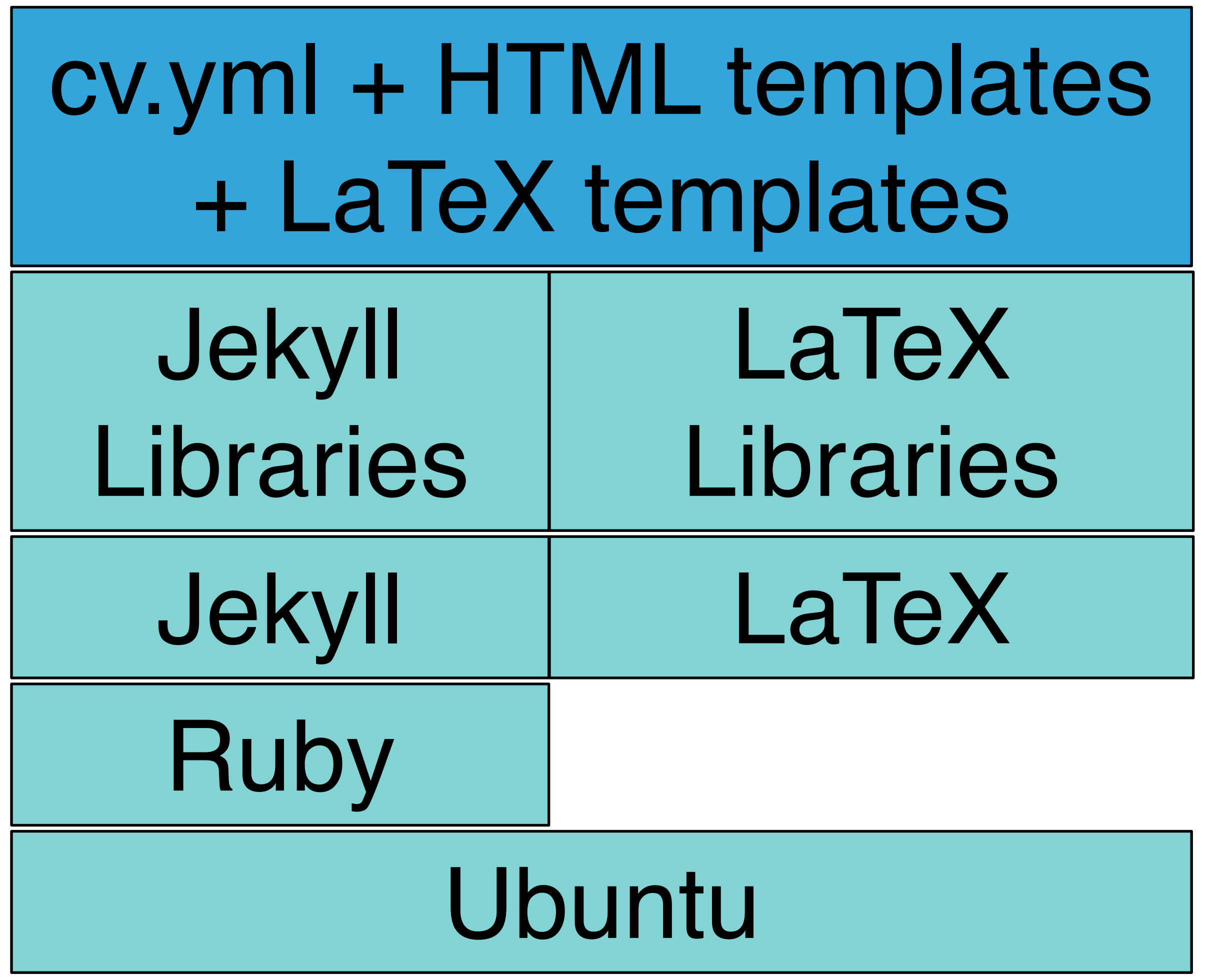
Containers are Deployed From Images

- A container *image* is an archive with a complete filesystem
- Images are defined in terms of layers
- Ideally: include *all* dependencies in image (do not fetch at runtime)
- Publish container images to registries
- A container is a set of processes running within a copy of that filesystem
- OS can impose restrictions on memory limits, access to CPU, I/O devices, etc

Example Containers: Building jonbell.net

Example snippet cv.yml

```yaml
- group: Conference Technical Program Committee Membership
  items:
    - name: Automated Software Engineering (ASE)
      years: [2018, 2019, 2020, 2021, 2022, 2023]
    - name: Foundations of Software Engineering (ESEC/FSE)
      years: [2022, 2023]
   …
```



Screenshot of generated section of website:

Service Activities in 2023:

Conference/Professional Organization Leadership

- ISSTA Tools Track Co-Chair
- Workshop on Software Engineering Education for the Next Generation at ICSE Workshop Co-Organizer

Conference Technical Program Committee Membership

- Automated Software Engineering (ASE)
- Foundations of Software Engineering (ESEC/FSE)
- International Conference on Program Comprehension (ICPC)
- International Symposium on Software Testing and Analysis (ISSTA)

Screenshot of generated section of CV:

Conference Technical Program Committee Membership

Automated Software Engineering (ASE) 2018, 2019, 2020, 2021, 2022, 2023
Foundations of Software Engineering (ESEC/FSE) 2022, 2023
IEEE Secure Development Conference 2021
International Conference on Mining Software Repositories (MSR) 2020
International Conference on Program Comprehension (ICPC) 2023
International Conference on Software Engineering (ICSE) 2019, 2020, 20
2024

Example Containers: Building jonbell.net

Base container: built only when I want to update dependencies

```
FROM ubuntu:focal
ARG DEBIAN_FRONTEND=noninteractive

RUN curl -sL https://deb.nodesource.com/setup_16.x  | bash
RUN apt-get update
RUN apt-get -y install ruby
RUN apt-get -y install texlive-latex-base texlive-fonts-recommended \
    texlive-latex-extra texlive-fonts-extra texlive-bibtex-extra \
    ruby-full build-essential zlib1g-dev locales curl nodejs
RUN gem install jekyll bundler jekyll-sitemap jekyll-seo-tag \
    jekyll-coffeescript jekyll-scholar coffee-script coffee-script-source \
    bibtex-ruby citeproc-ruby csl-styles rexml execjs latex-decode \
    citeproc csl namae
```

Website container: built on each website update

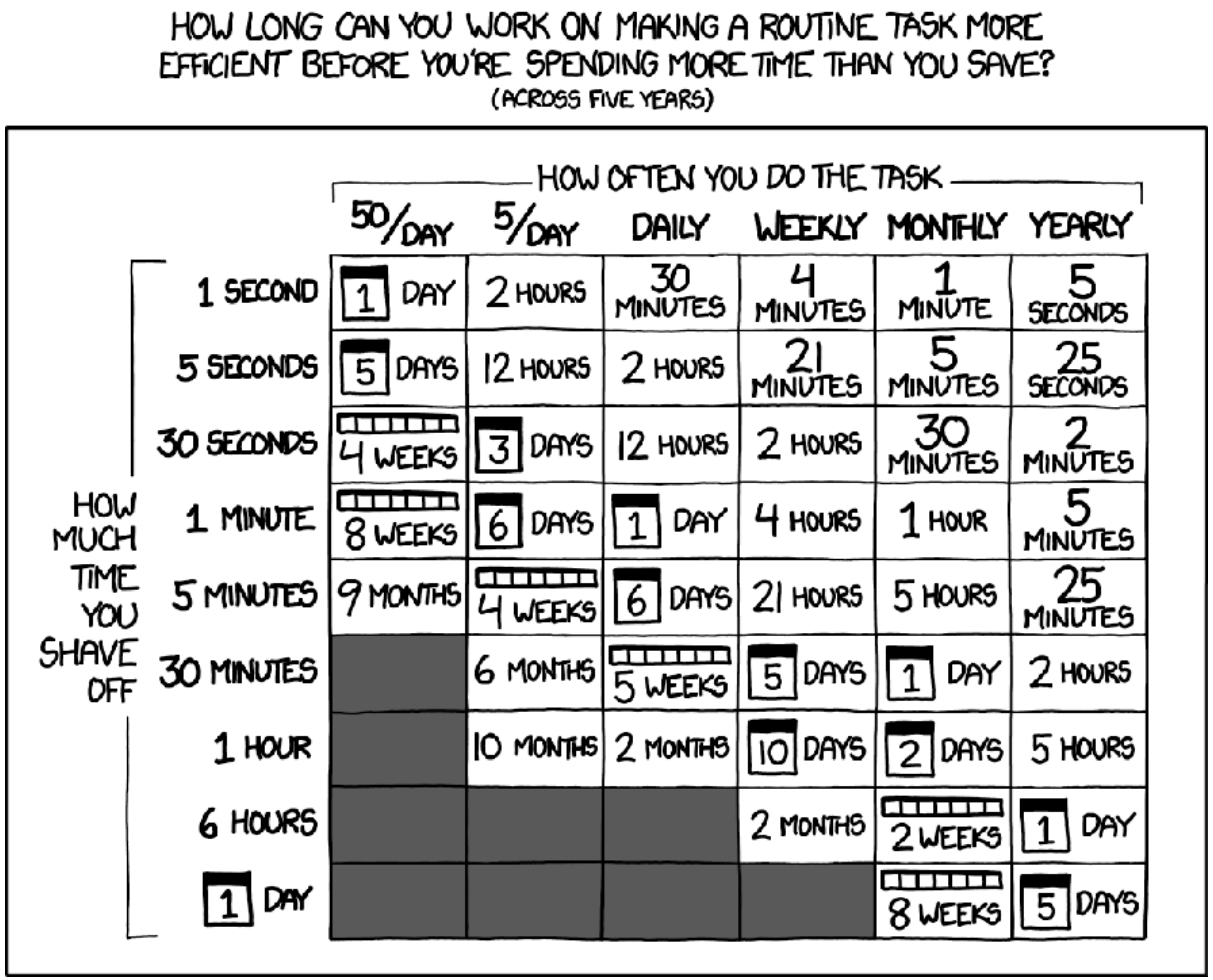
```
FROM jonbell/website-builder
# Copy site directory
COPY . /site
WORKDIR /site
RUN bundle install
RUN mkdir _cv/generated/
# Build Site
RUN bundle exec jekyll build
# Build CV
WORKDIR /site/_cv
RUN pdflatex jbell_cv
RUN bibtex jbell_cv
RUN pdflatex jbell_cv
RUN pdflatex jbell_cv

RUN cp jbell_cv.pdf ../_site/cv.pdf

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

What is the payoff of this website/CV mess?

Estimated 8 hours to migrate out of WordPress to this containerized build



	50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

https://xkcd.com/1205/

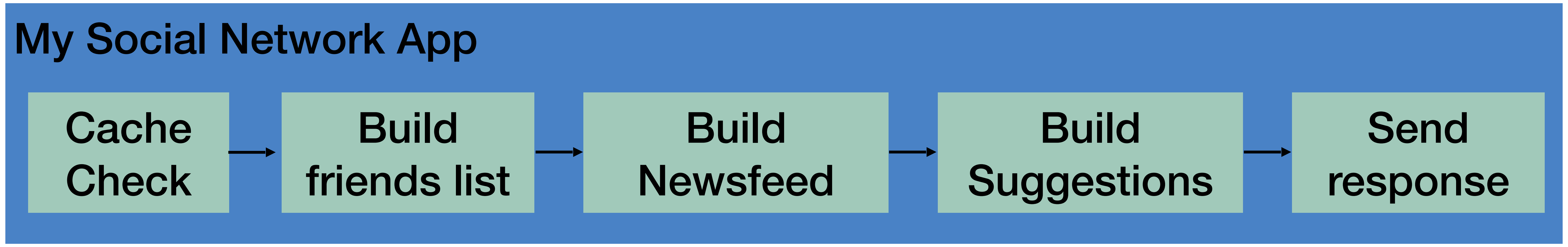
Infrastructure as Code

Common metaphor “Infrastructure as Pets vs Cattle”

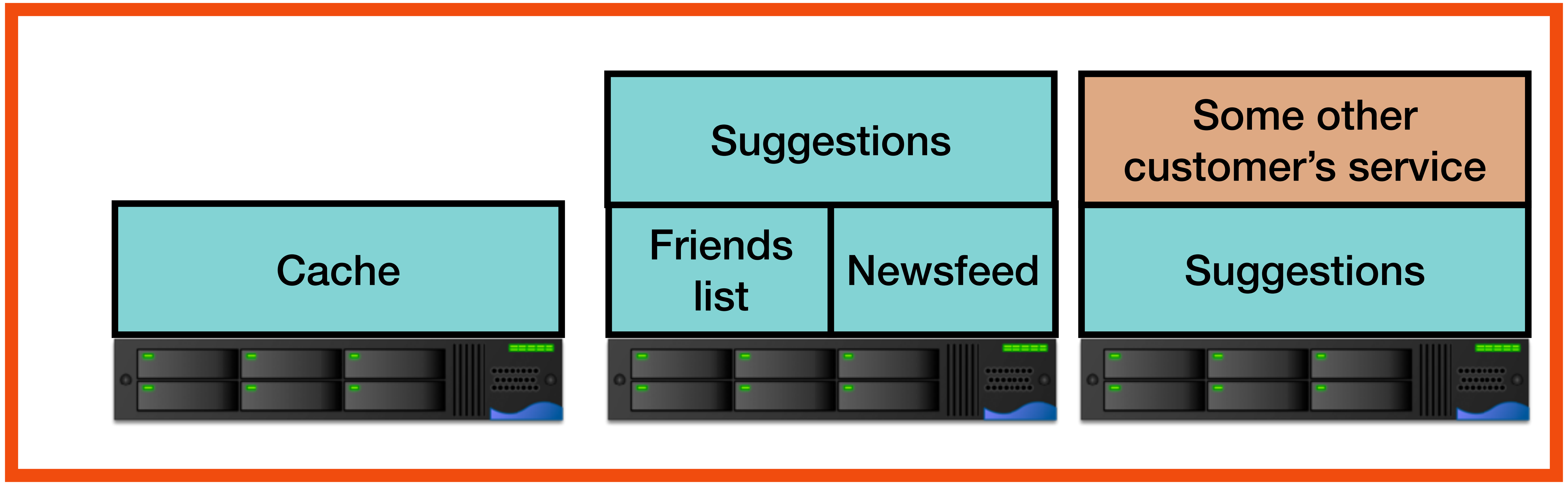
- Traditional approach to run a server: install dependencies, configure them, maintain the entire system
  - Recover from crash: Manually re-install/configure
  - Share with others: Write a blog post
  - Creating a test environment: Manual
- IaC: Specify the docker container(s) to run, along with their network configuration
  - Recover from crash: Re-deploy containers
  - Share with others: Share a configuration file
  - Creating a test environment: Automatic

Infrastructure as Code

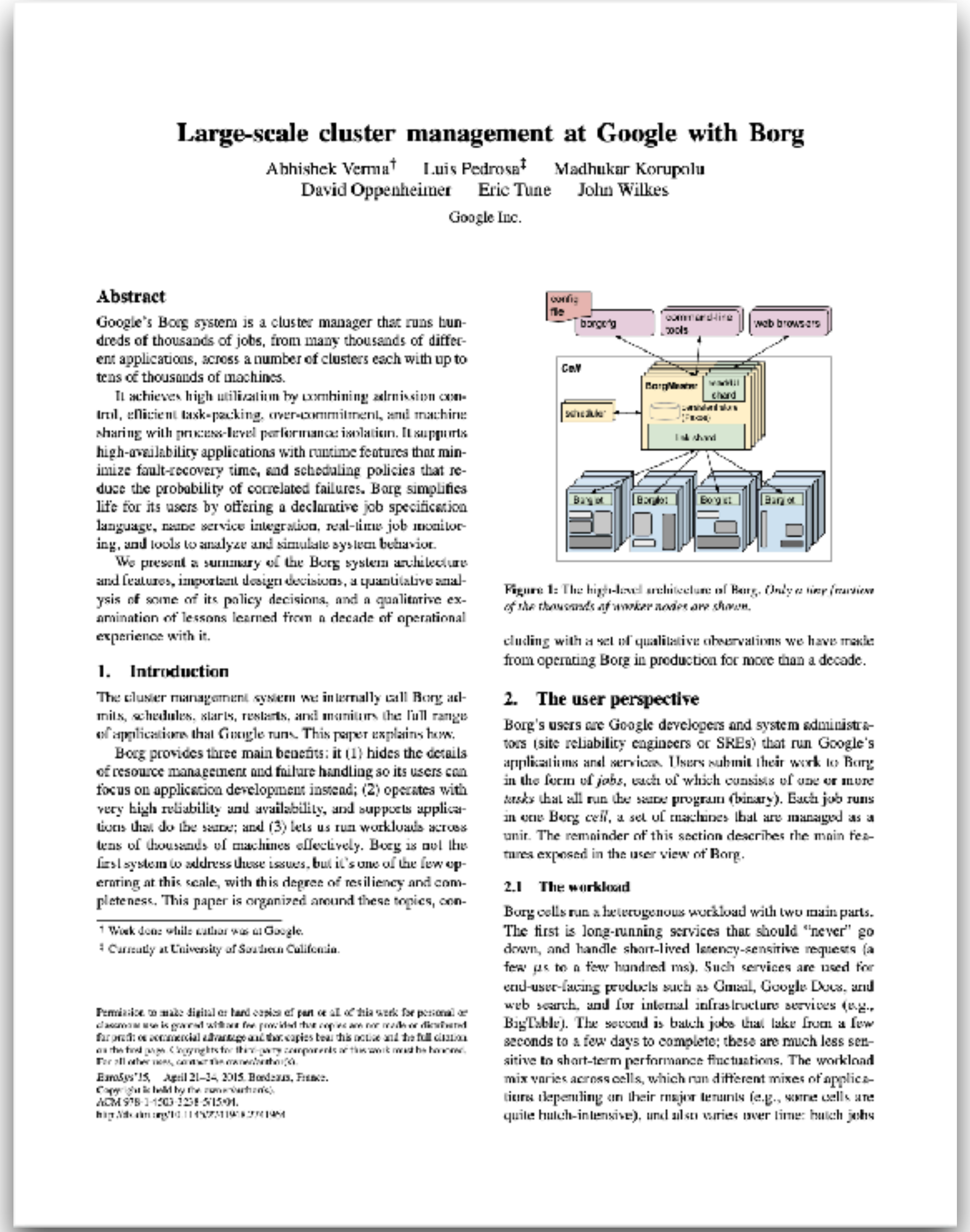
Managing Container Deployments: Kubernetes



"Give me at least 1 of each of these app services in their own docker containers, and if the load gets above a threshold, spin up more of them"



Managed by Kubernetes



Large-scale cluster management at Google with Borg

Abhishek Verma† Luis Pedrosa‡ Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes

Google Inc.

Abstract

Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.

It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. It supports high-availability applications with runtime features that minimize fault-recovery time, and scheduling policies that reduce the probability of correlated failures. Borg simplifies life for its users by offering a declarative job specification language, name service integration, real-time job monitoring, and tools to analyze and simulate system behavior.

We present a summary of the Borg system architecture and features, important design decisions, a quantitative analysis of some of its policy decisions, and a qualitative examination of lessons learned from a decade of operational experience with it.

1. Introduction

The cluster management system we internally call Borg admits, schedules, starts, restarts, and monitors the full range of applications that Google runs. This paper explains how.

Borg provides three main benefits: it (1) hides the details of resource management and failure handling so its users can focus on application development instead; (2) operates with very high reliability and availability, and supports applications that do the same; and (3) lets us run workloads across tens of thousands of machines effectively. Borg is not the first system to address these issues, but it's one of the few operating at this scale, with this degree of resiliency and completeness. This paper is organized around these topics, concluding with a set of qualitative observations we have made from operating Borg in production for more than a decade.

† Work done while author was at Google.
‡ Currently at University of Southern California.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3238-5/15/04.

Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

2. The user perspective

Borg's users are Google developers and system administrators (site reliability engineers or SREs) that run Google's applications and services. Users submit their work to Borg in the form of *jobs*, each of which consists of one or more *tasks* that all run the same program (binary). Each job runs in one Borg *cell*, a set of machines that are managed as a unit. The remainder of this section describes the main features exposed in the user view of Borg.

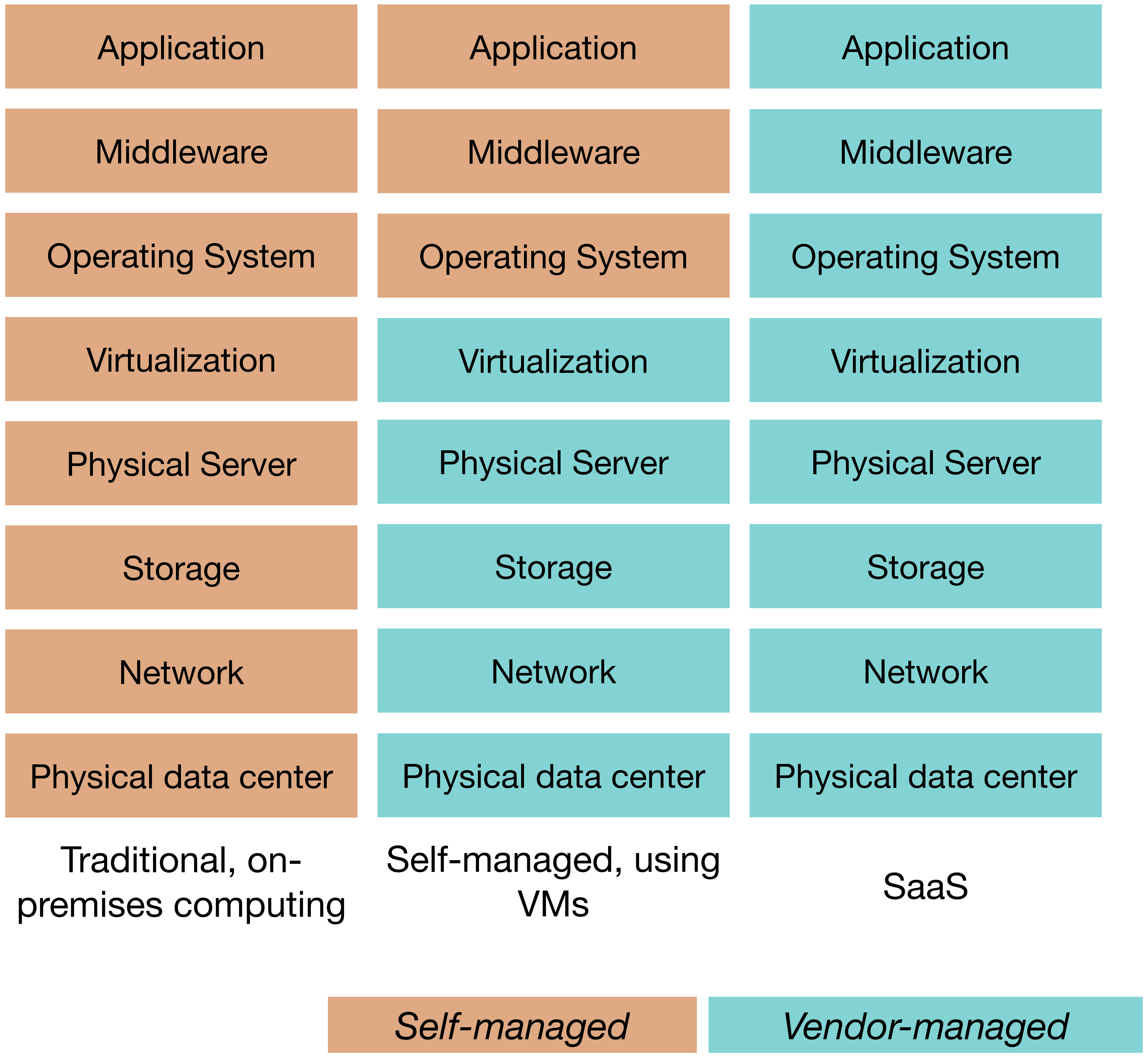
2.1 The workload

Borg cells run a heterogenous workload with two main parts. The first is long-running services that should "never" go down, and handle short-lived latency-sensitive requests (a few µs to a few hundred ms). Such services are used for end-user-facing products such as Gmail, Google Docs, and web search, and for internal infrastructure services (e.g., BigTable). The second is batch jobs that take from a few seconds to a few days to complete; these are much less sensitive to short-term performance fluctuations. The workload mix varies across cells, which run different mixes of applications depending on their major tenants (e.g., some cells are quite batch-intensive), and also varies over time: batch jobs

https://research.google/pubs/pub43438/

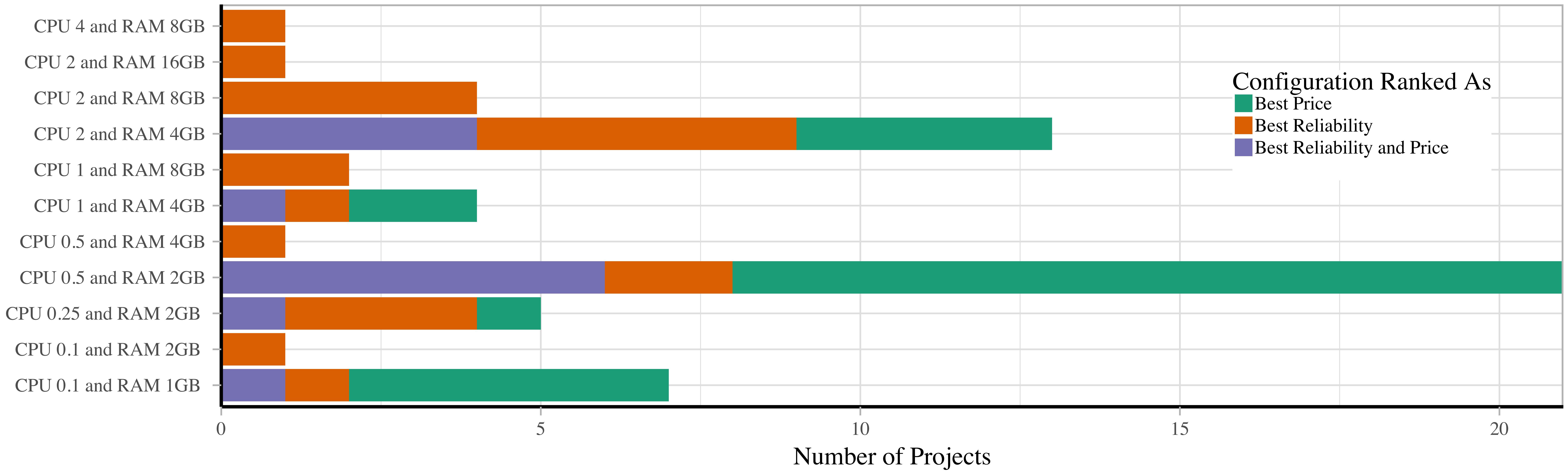
Continuous Integration Service Models

- Self-hosted/managed on-premises or in cloud
  - Jenkins
- Fully cloud managed
  - GitHub Actions, CircleCI, Travis, many more…
  - Billing model: pay per-build-minute running on SaaS infrastructure
  - “Self-hosted runners” run builds on your own infrastructure, usually “free”



Allocate Enough Resources to Avoid Flaky Tests

Study of 30 open-source projects written in Java



Cloud Infrastructure is Best Suited for Variable Workloads

- Consider: Does your workload benefit from ability to scale up/down?
- Example: need to run 300 VMs, each with 4 vCPUs, 16GB RAM
- Private cloud: Dell PowerEdge Pricing (AMD EPYC 64 core CPUs)
  - 7 servers, each with 128 cores/256 threads, 512GB RAM, 3 TB storage = $162,104
- Public cloud: Amazon EC2 Pricing (M5.xlarge instances, $0.121/VM-hour)
  - 10 VMs for 1 year + 290 VMs for 1 month: $36,215.30
  - 300 VMs for 1 year: $317,988