

CI/CD and Deployment Challenges

17-313: Foundations of Software Engineering
<https://cmu-313.github.io>

Michael Hilton and Josh Sunshine
Fall 2026

Thanks to Jon Bell for slide inspiration:
<https://neu-se.github.io/CS4530-Spring-2024/>

Administrivia

- **P3 - Interviews & Requirements.** Due Thursday, October 29 @ 11:59pm
- Guest lecture on Thursday
- No class Tuesday Nov 3rd
- Guest lecture Thursday Nov 5th (there will be a pre-activity due on Wed Nov 4th)

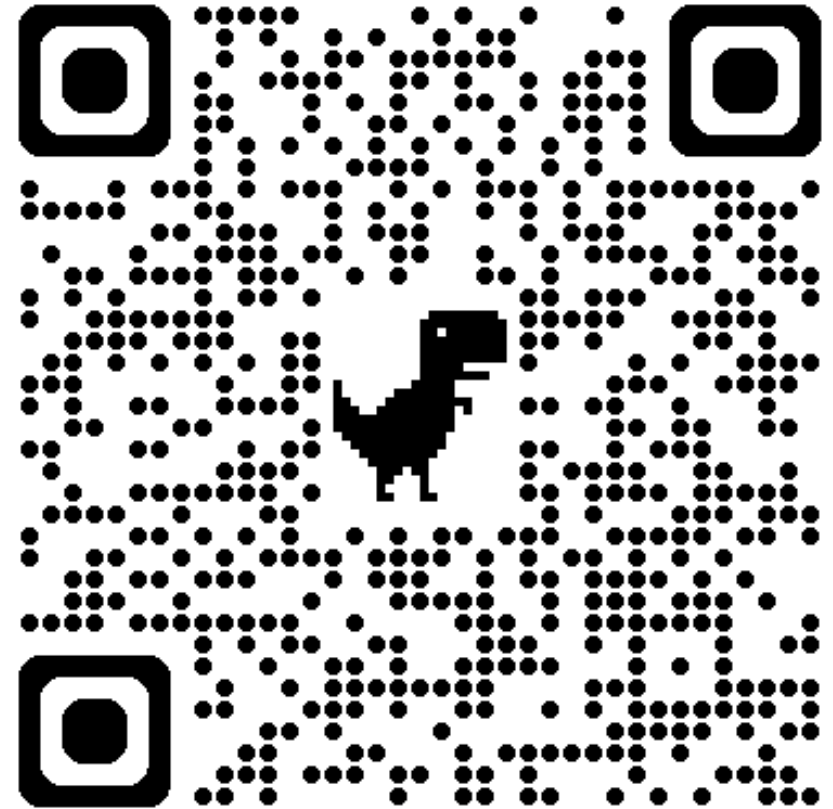
Learning objectives for this lesson

- By the end of this lesson, you should be able to...
 - Describe how continuous integration catches errors sooner, and what makes a CI process effective
 - Explain how containers, infrastructure as code, and orchestration (Kubernetes) make deployments repeatable
 - Compare deployment strategies (big bang, rolling, blue/green, canary, feature flags) and the risks each one reduces
 - Explain why monitoring is what decides whether a release is healthy
 - Compare continuous delivery with test-driven development as quality assurance strategies

Retrospectives

- What should we start doing?
- What should we stop doing?
- What should we keep doing?

Helpful to take time to stop and reflect on how things are going.



CrowdStrike Failure



78 min

From update being sent until the update is reverted.

~8.5M

Windows machines crash and fail to reboot (Microsoft estimate)

Airlines grounded flights, hospitals postponed procedures, banks and broadcasters went offline. Fix involved plugging in a flash drive and physically rebooting machine.

The update had passed automated validation. **So what should have stopped it?**

Sources: CrowdStrike, External Technical Root Cause Analysis — Channel File 291 (Aug 2024); Microsoft (Jul 20, 2024)

Validation bug, Process failure

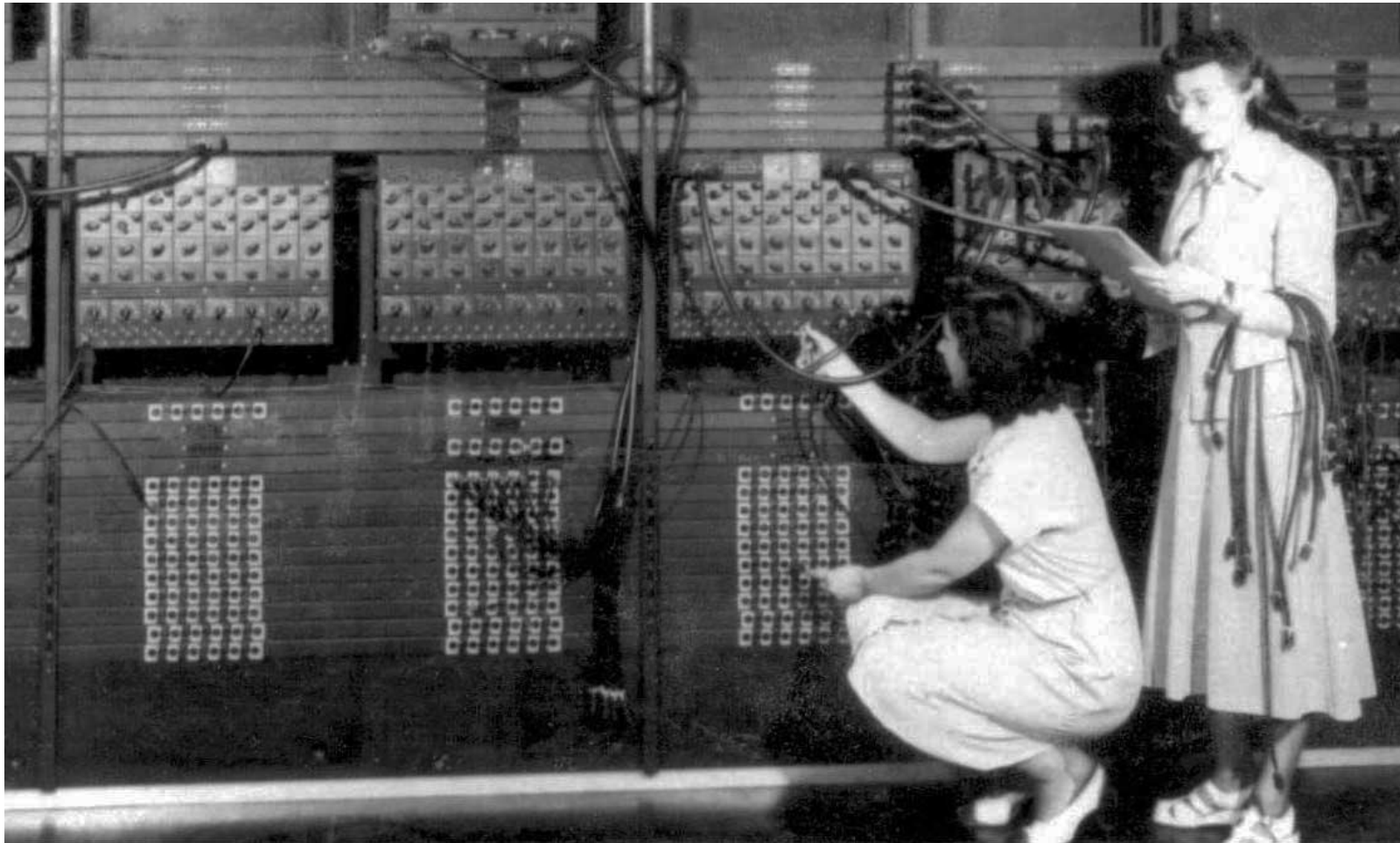
What happened

- A content validator approved the update, but the validator itself had a bug
- It shipped to every online client at once
- Customers had no control over when content updates arrived
- Crashed machines couldn't receive the fix, so rollback meant physical reboot

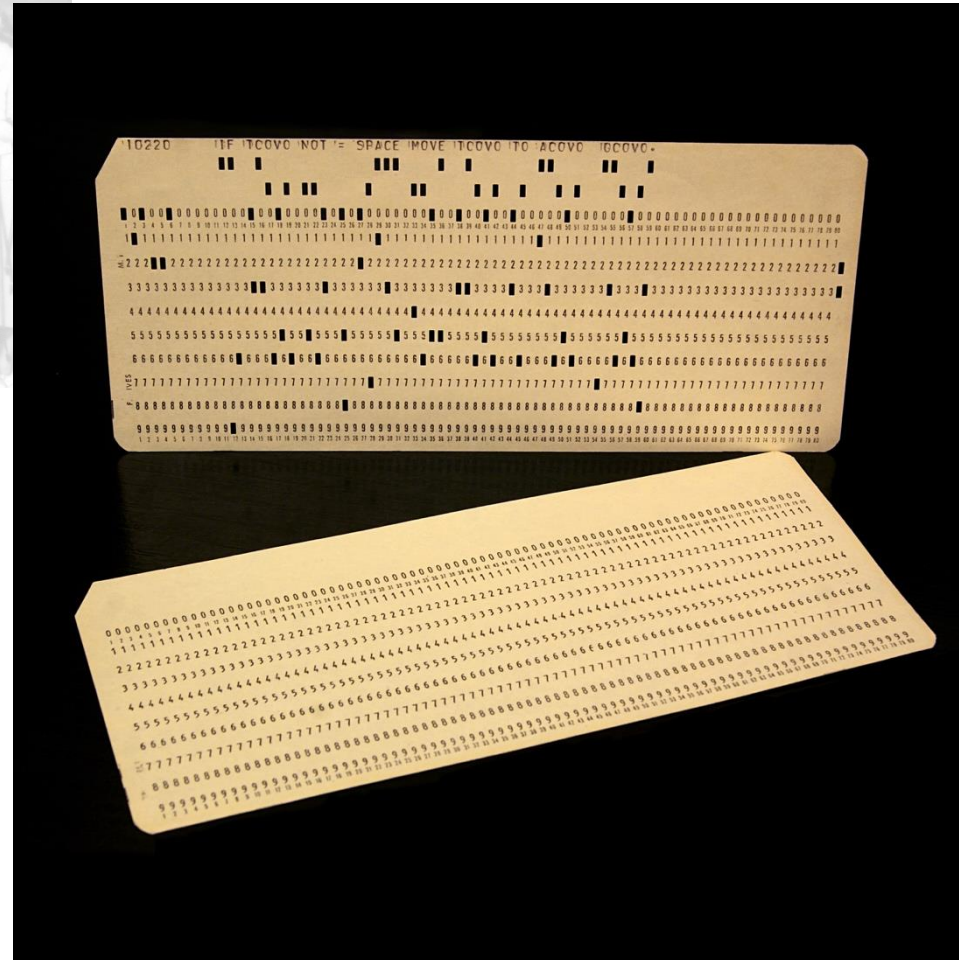
What CrowdStrike committed to

- Staggered deployment, “starting with a canary deployment”
- More testing: local, content-update, and rollback testing
- Monitoring sensor and system health during rollouts
- Letting customers control when updates are delivered

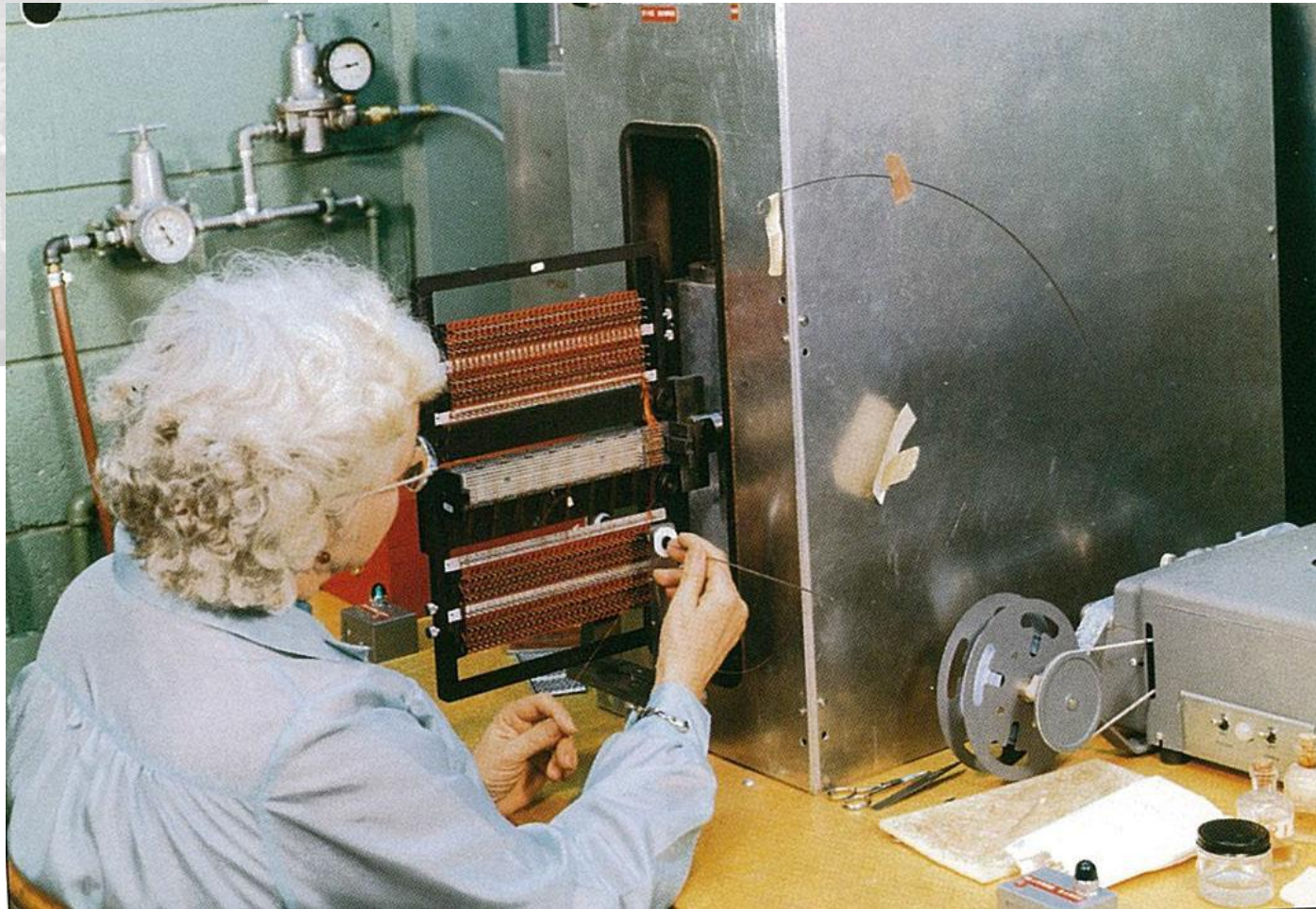
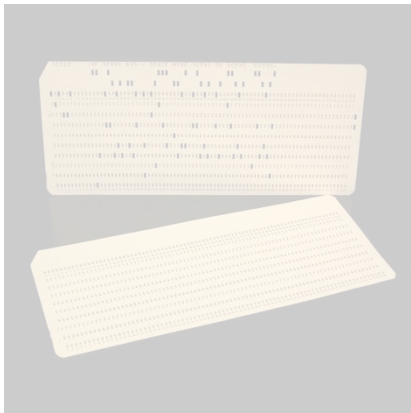
History of Software Distribution



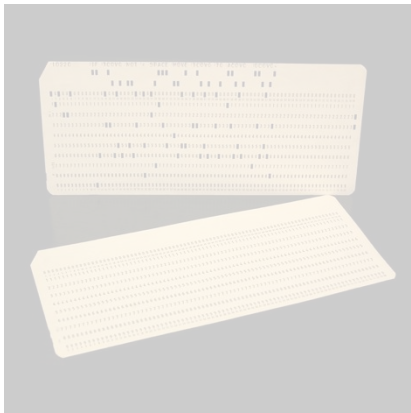
History of Software Distribution



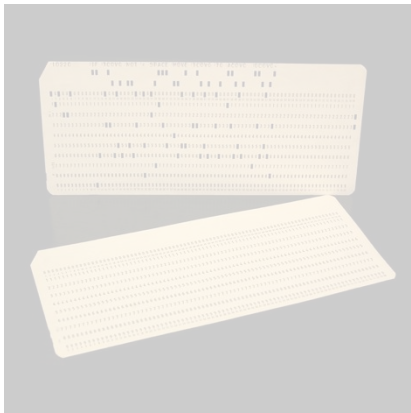
History of Software Distribution



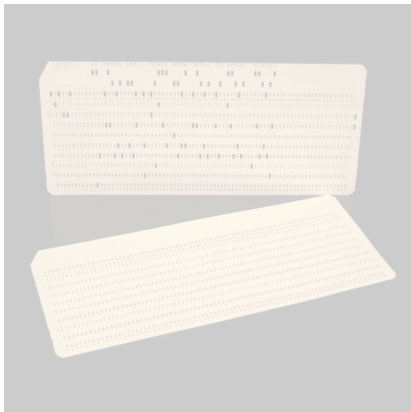
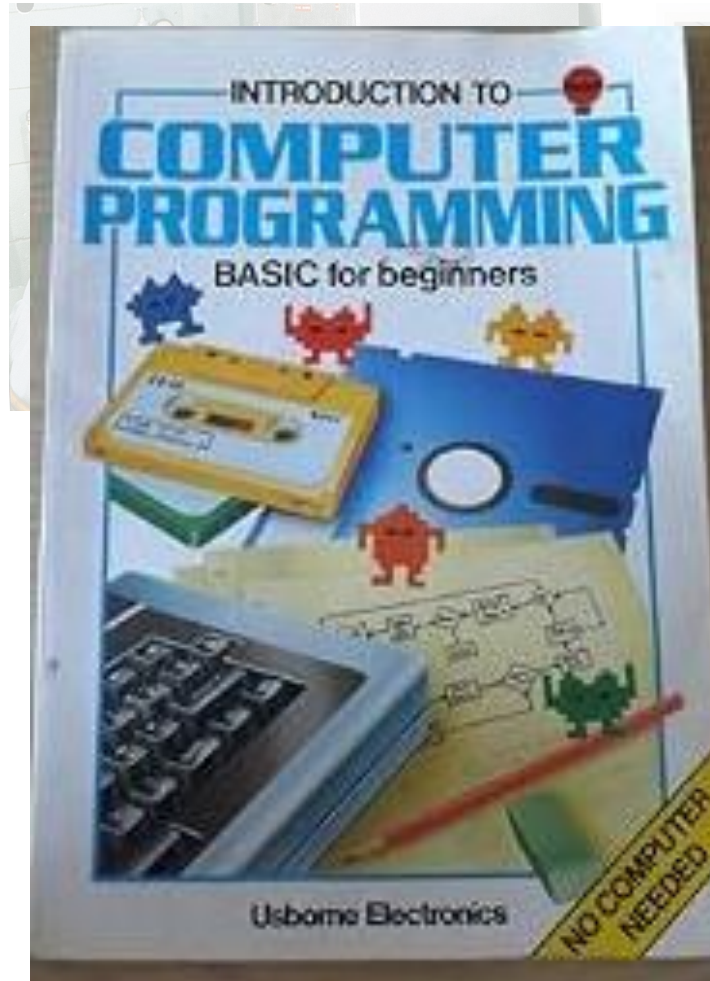
History of Software Distribution



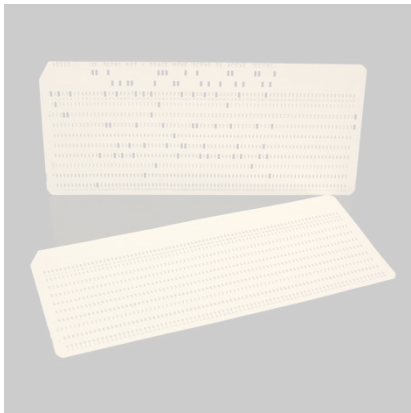
History of Software Distribution



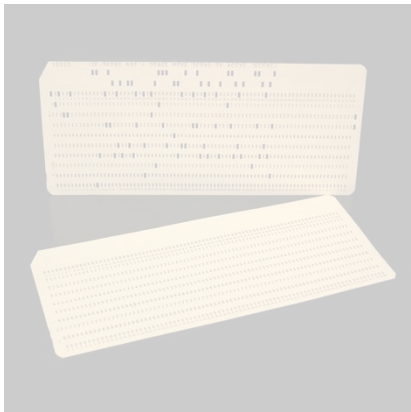
History of Software Distribution



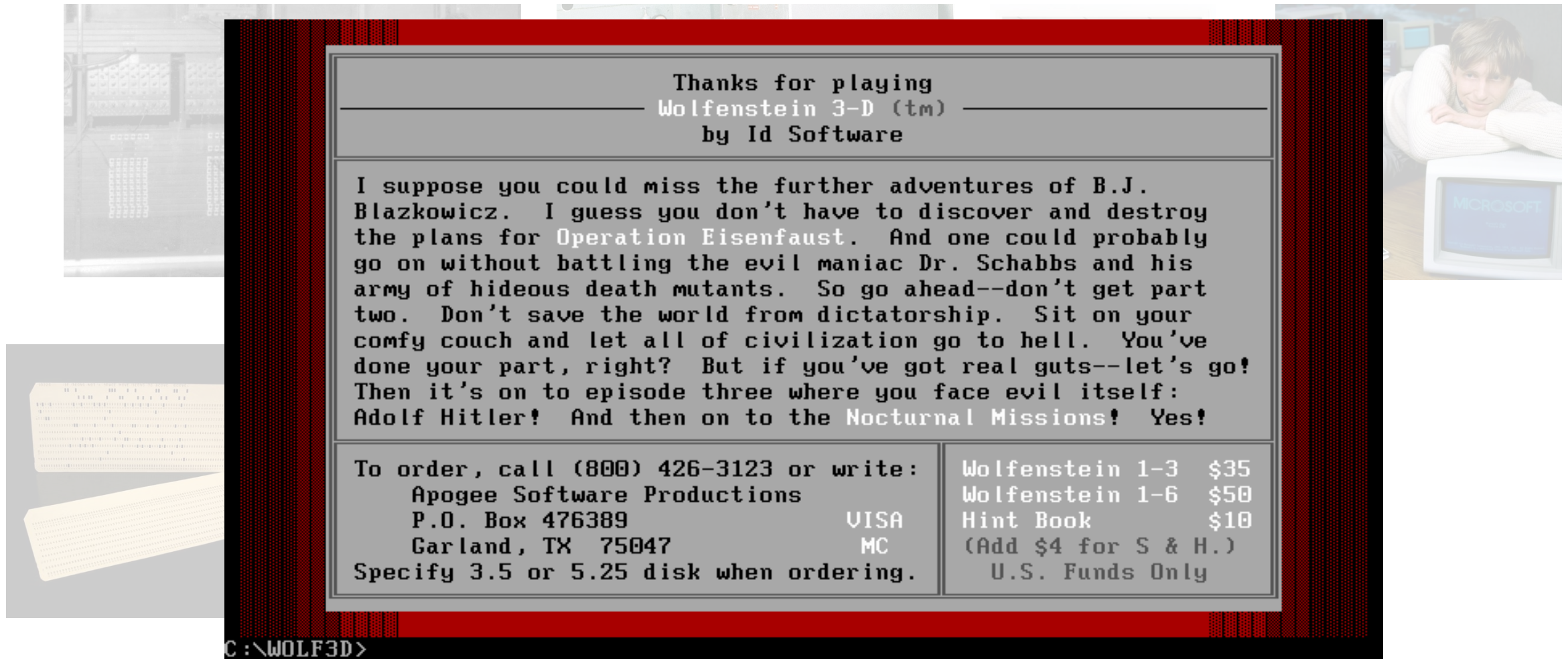
History of Software Distribution



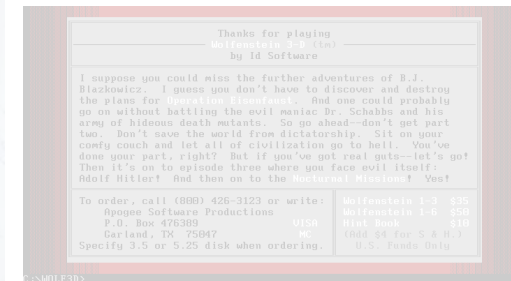
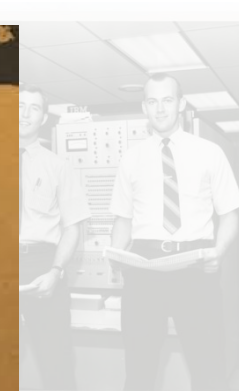
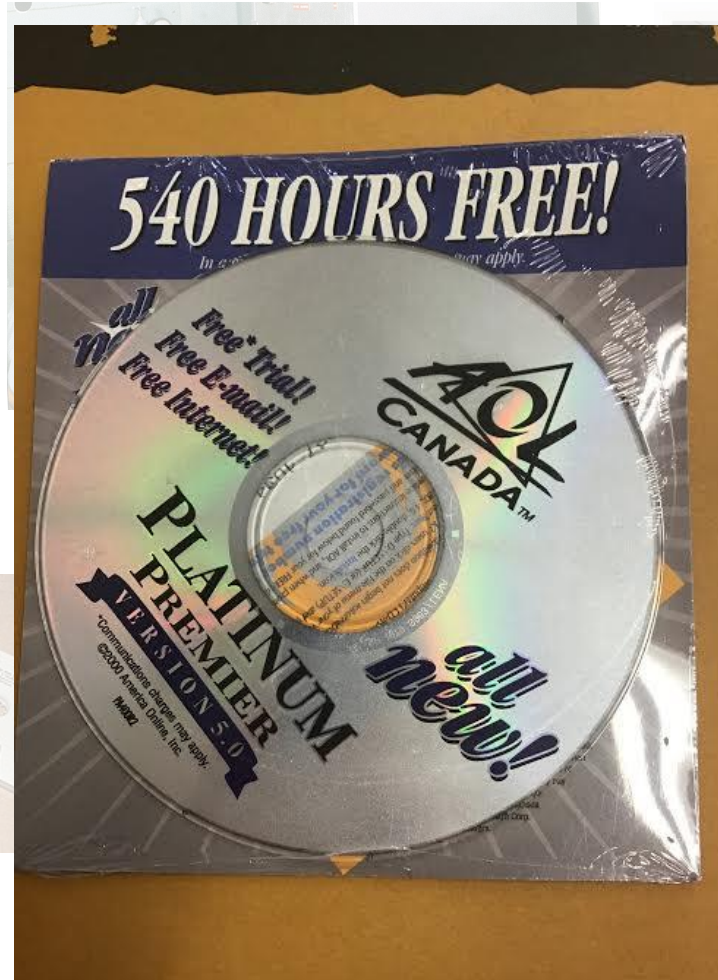
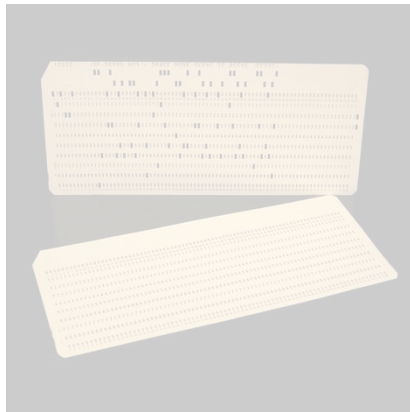
History of Software Distribution



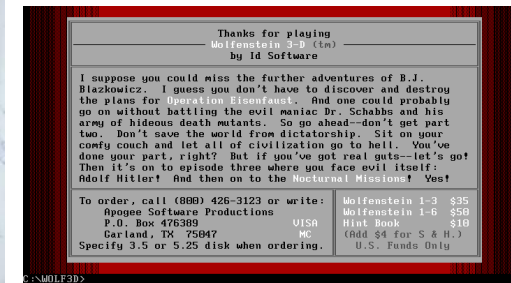
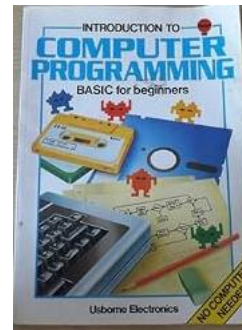
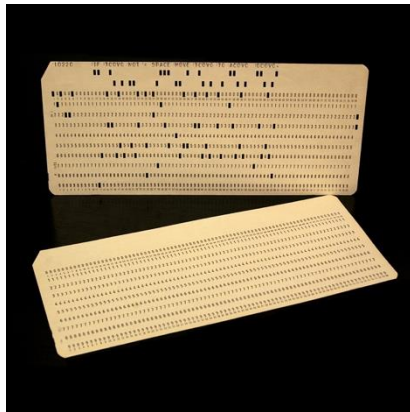
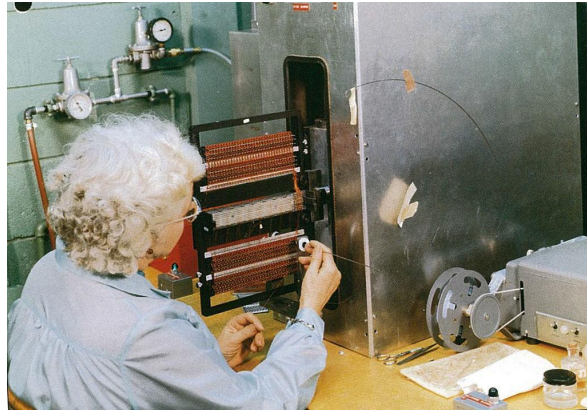
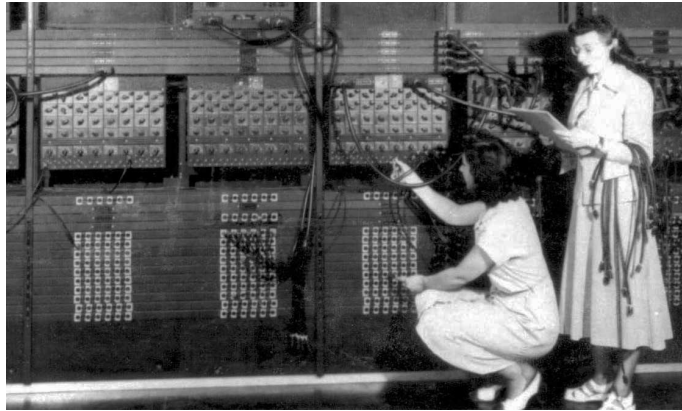
History of Software Distribution



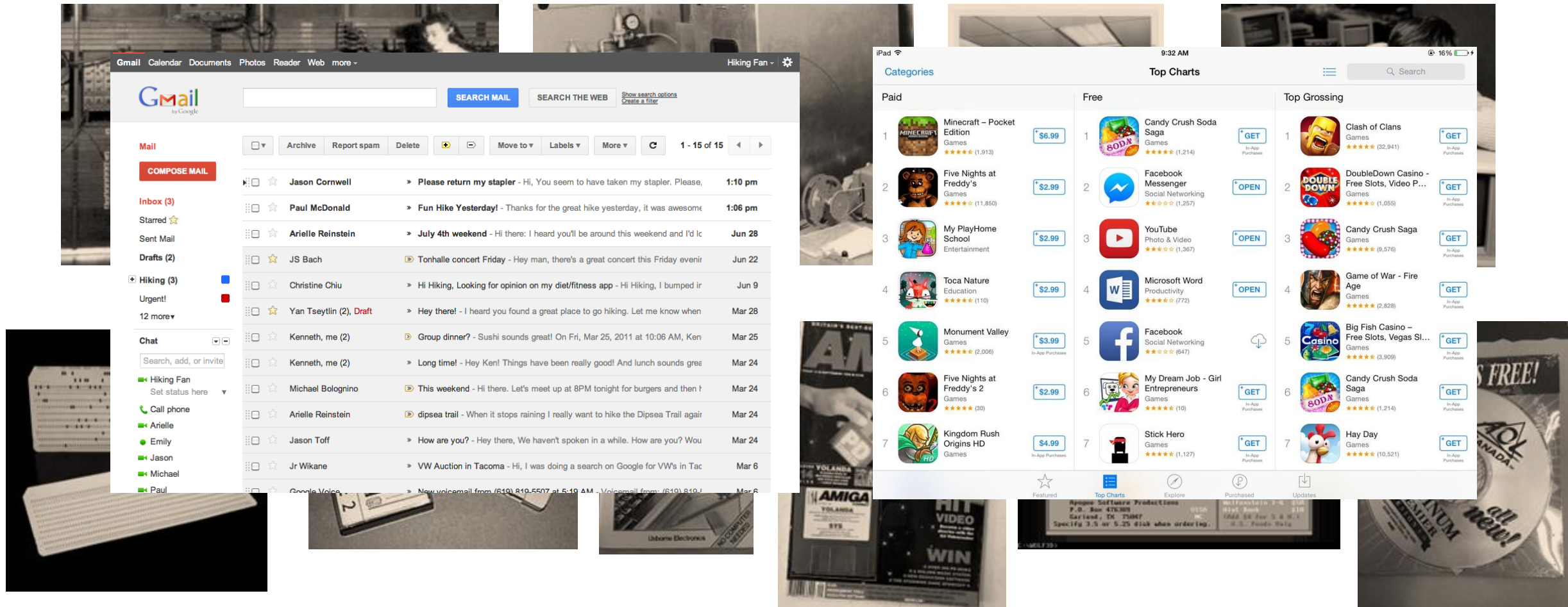
History of Software Distribution



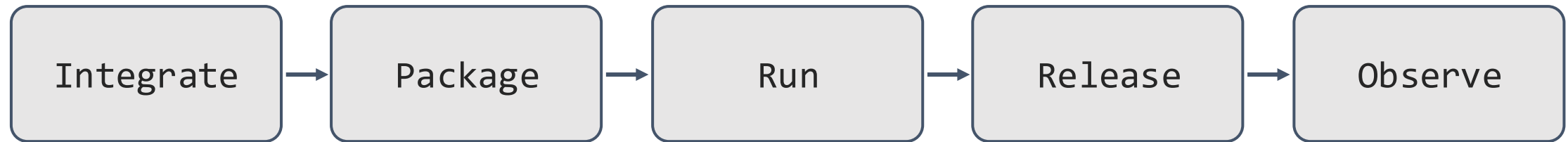
History of Software Distribution



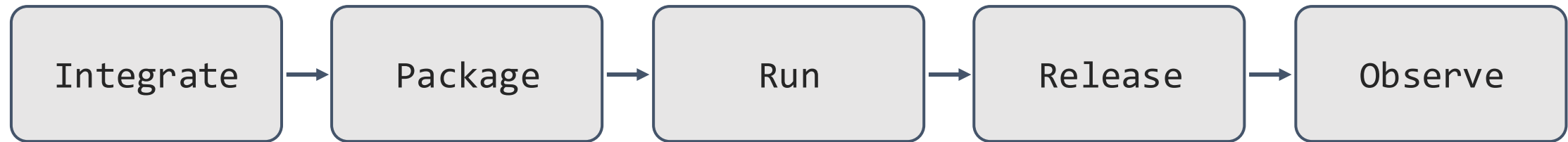
History of Software Distribution



A commit's trip to production, and what goes wrong



A commit's trip to production, and what goes wrong

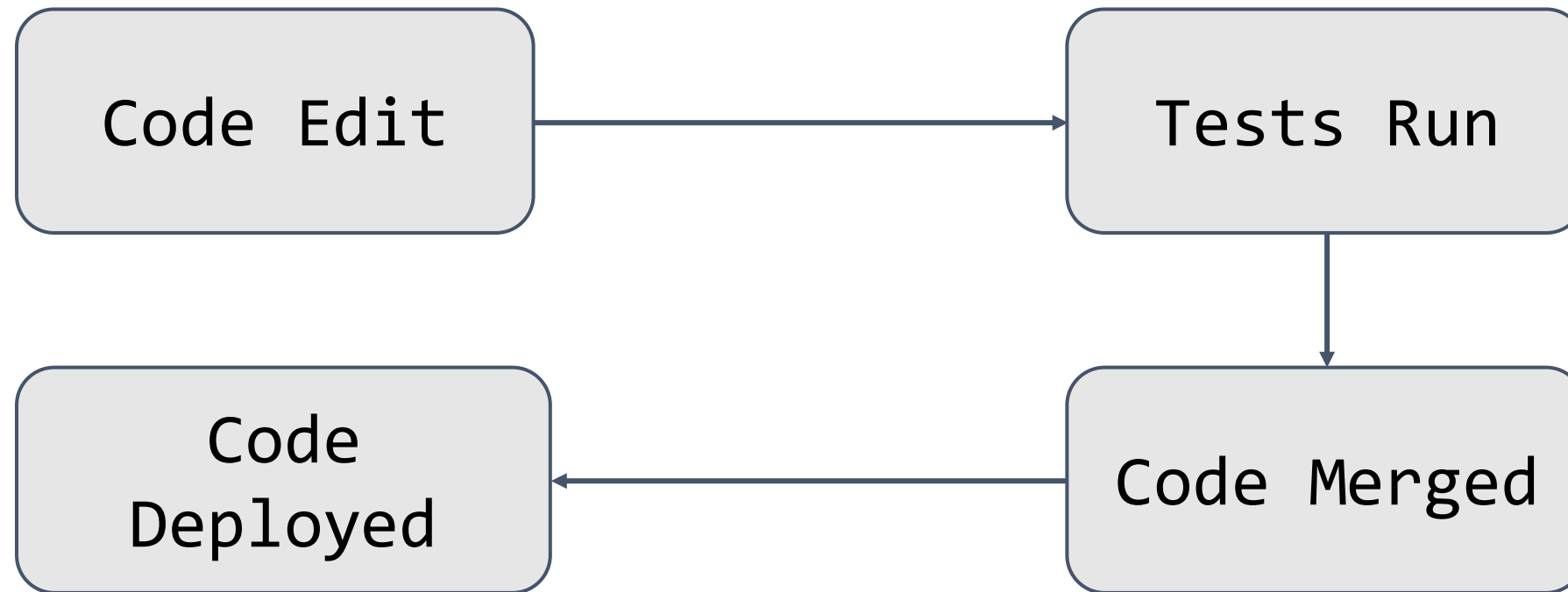


Challenge:	Merging everyone's changes	"Works on my machine"	Hundreds of containers	Every release is a bet	Is it actually working?
Solution:	CI	Cloud, Docker, IaC	Kubernetes	CD, canaries, flags	Monitoring

Challenge 1: Merging everyone's changes

Continuous Integration

CI/CD Pipeline overview



History of CI



(1999) Extreme Programming (XP) rule: “Integrate Often”



(2000) Martin Fowler posts “Continuous Integration” blog



(2001) First CI tool



Jenkins

(2005) Hudson/Jenkins



Travis CI

(2011) Travis CI



GitHub Actions

(2019) GitHub Actions

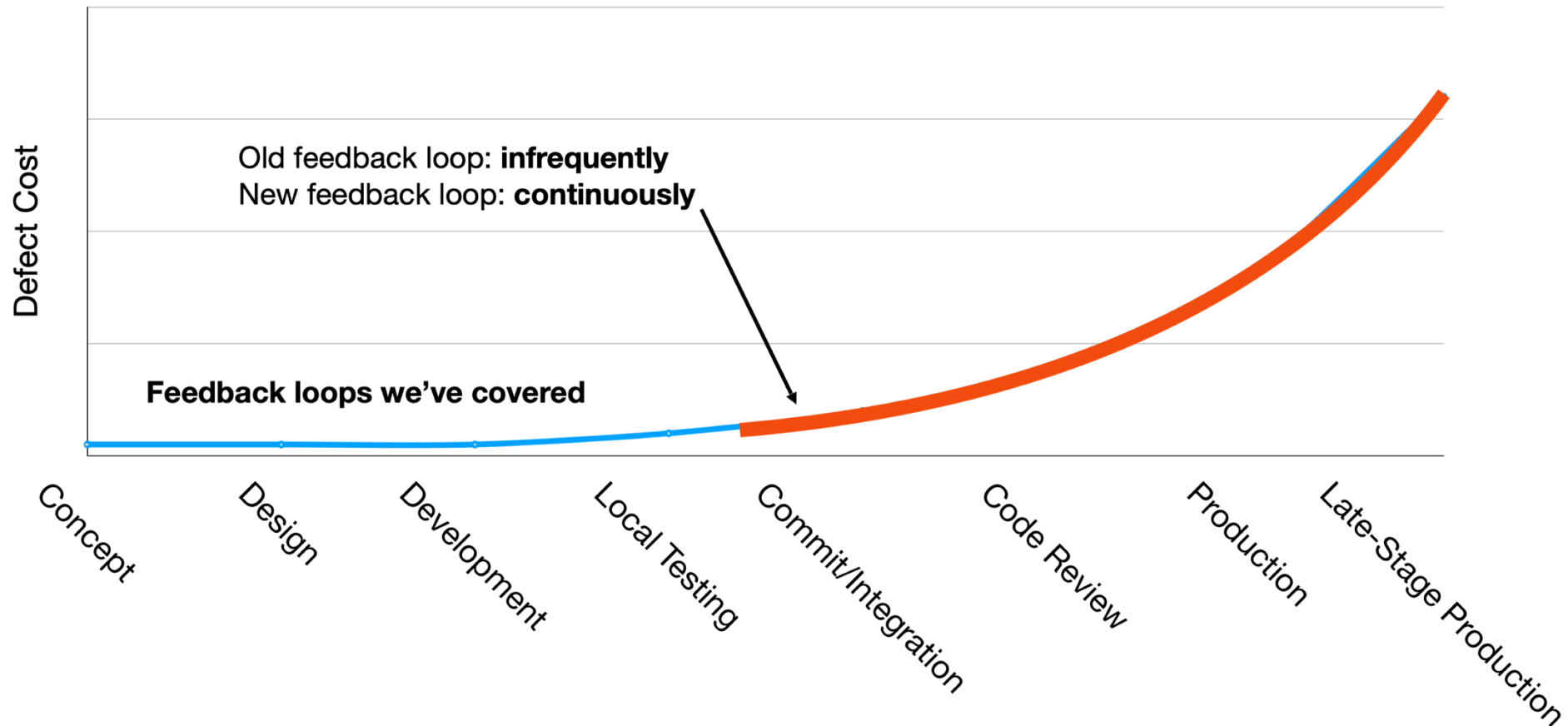
Observation

**CI helps us catch errors
before others see them**

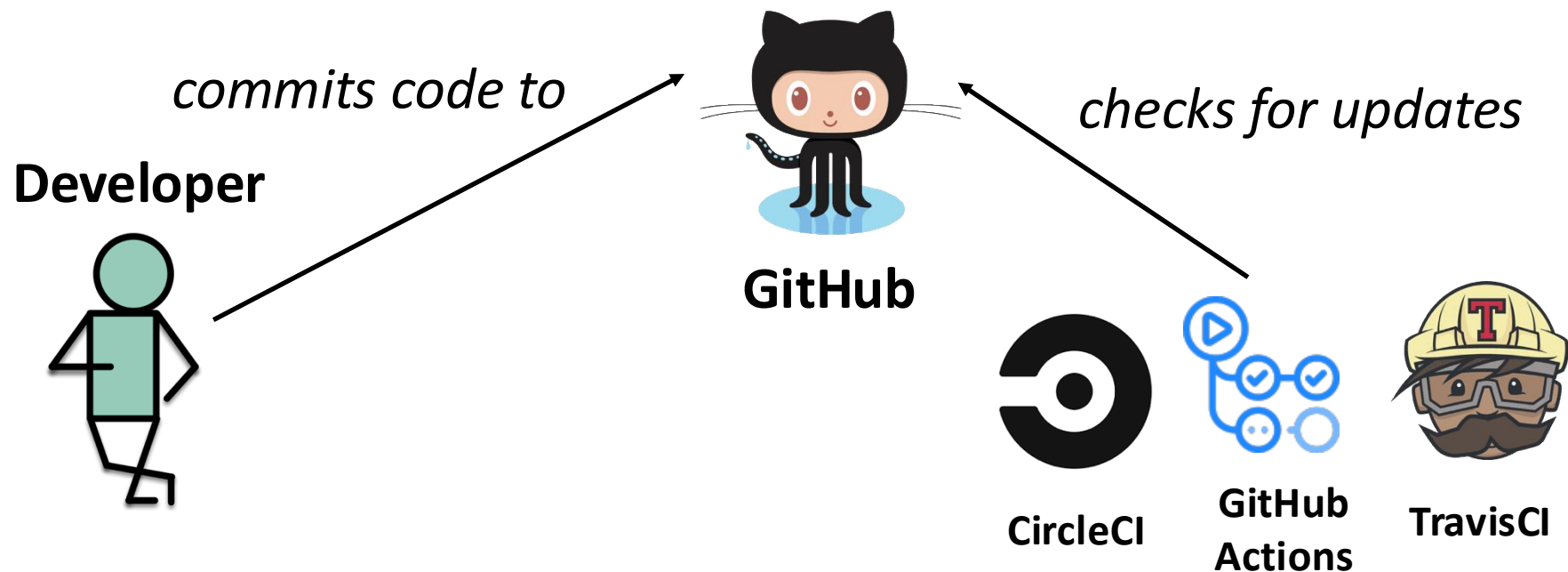
24

Agile values fast quality feedback loops

- Faster feedback = lower cost to fix bugs



CI is triggered by commits, pull requests, and other actions



Runs build for each commit

Automating Feedback Loops is Powerful

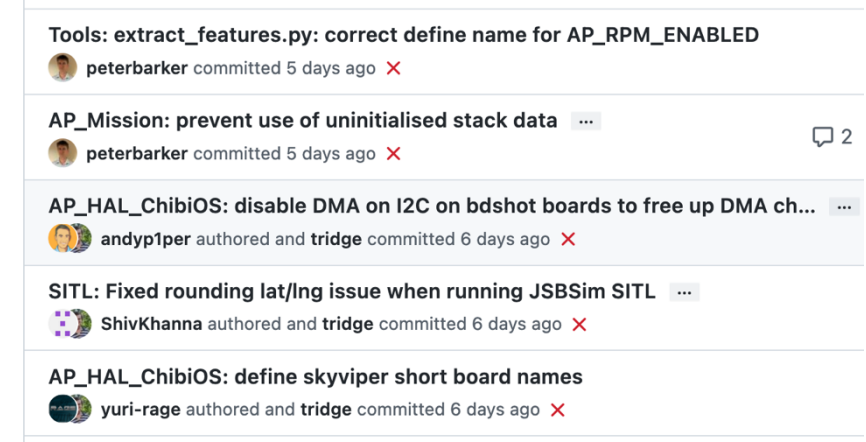
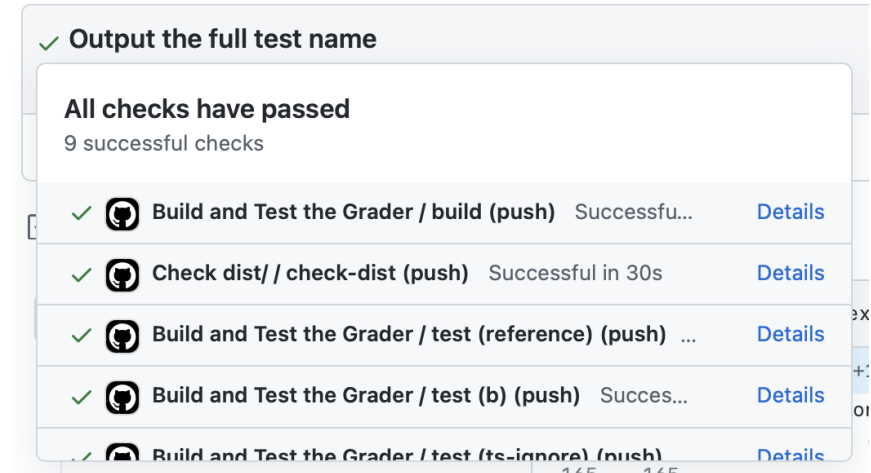
Consider tasks that are done by *dozens* of developers (e.g. testing/deployment)

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	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

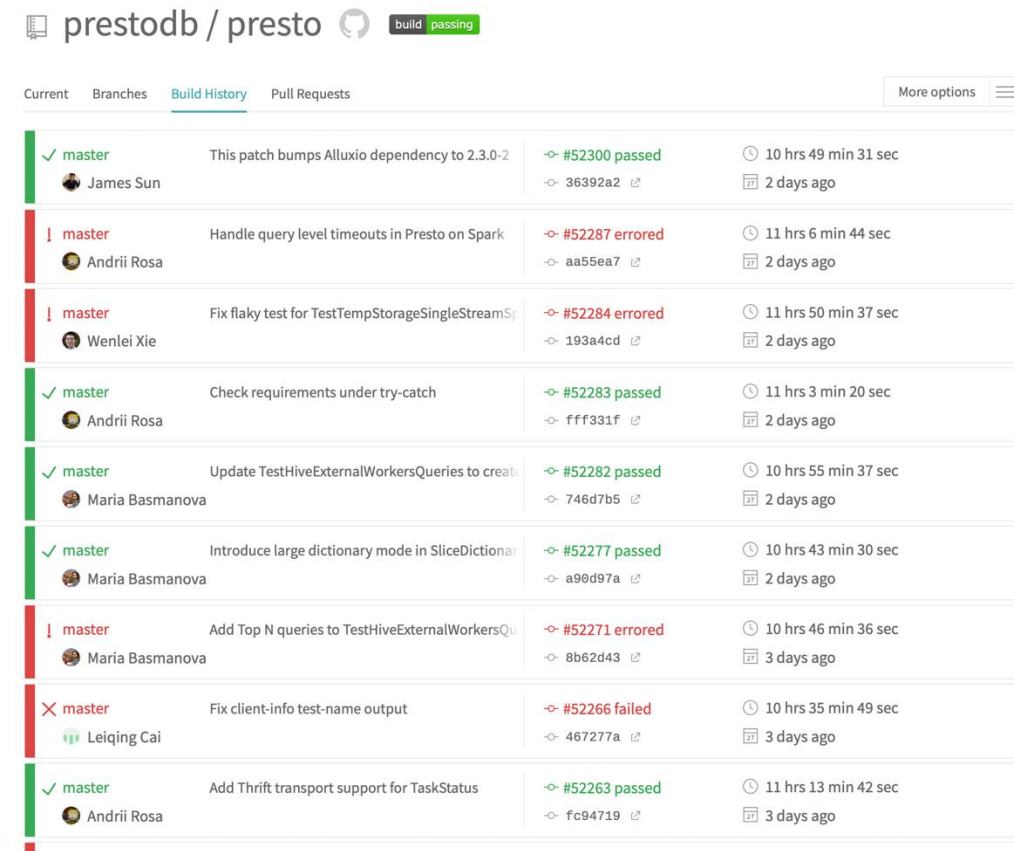
Attributes of effective CI processes

- 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
- Infrastructure:
 - CI should be fast, providing feedback within minutes or hours
 - CI should be repeatable (deterministic)



Effective CI processes are run often enough to reduce debugging effort

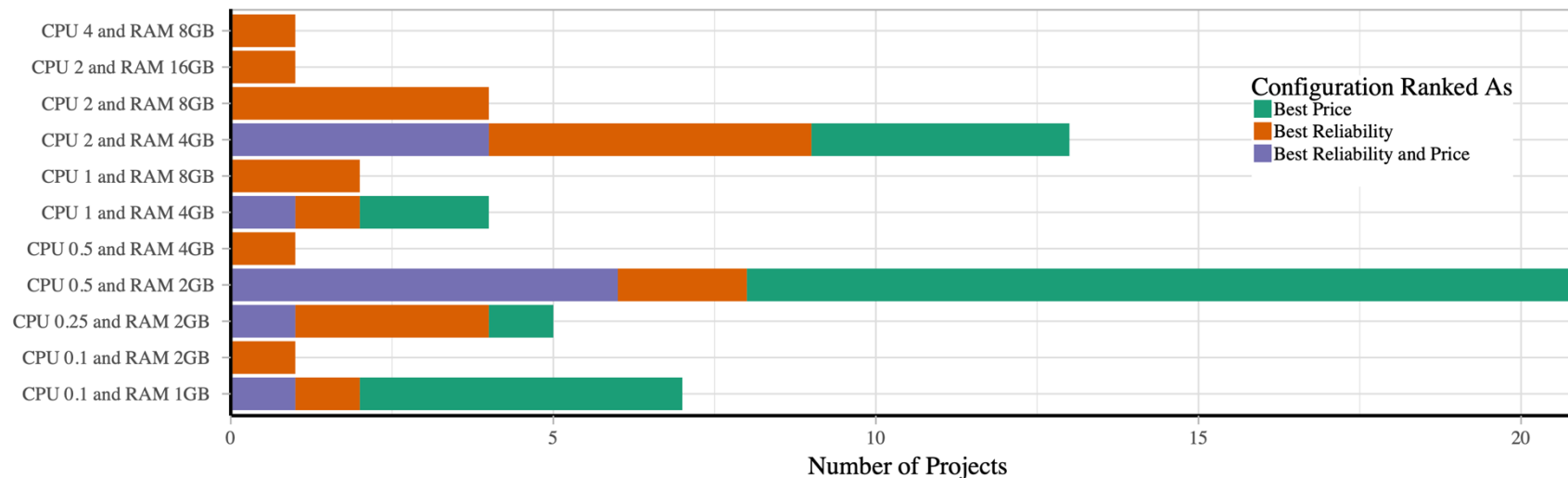
- Failed CI runs indicate a bug was introduced, and caught in that run
- More changes per-CI run require more manual debugging effort to assign blame
- A single change per-CI run pinpoints the culprit



prestodb / presto		build passing		
Current	Branches	Build History	Pull Requests	More options
✓	master	This patch bumps Alluxio dependency to 2.3.0-2	✓ #52300 passed	10 hrs 49 min 31 sec 2 days ago
	James Sun		36392a2	
!	master	Handle query level timeouts in Presto on Spark	✗ #52287 errored	11 hrs 6 min 44 sec 2 days ago
	Andrii Rosa		aa55ea7	
!	master	Fix flaky test for TestTempStorageSingleStreamSp	✗ #52284 errored	11 hrs 50 min 37 sec 2 days ago
	Wenlei Xie		193a4cd	
✓	master	Check requirements under try-catch	✓ #52283 passed	11 hrs 3 min 20 sec 2 days ago
	Andrii Rosa		fff331f	
✓	master	Update TestHiveExternalWorkersQueries to creat	✓ #52282 passed	10 hrs 55 min 37 sec 2 days ago
	Maria Basmanova		746d7b5	
✓	master	Introduce large dictionary mode in SliceDictionar	✓ #52277 passed	10 hrs 43 min 30 sec 2 days ago
	Maria Basmanova		a9d97a	
!	master	Add Top N queries to TestHiveExternalWorkersQu	✗ #52271 errored	10 hrs 46 min 36 sec 3 days ago
	Maria Basmanova		8b62d43	
✗	master	Fix client-info test-name output	✗ #52266 failed	10 hrs 35 min 49 sec 3 days ago
	Leiqing Cai		467277a	
✓	master	Add Thrift transport support for TaskStatus	✓ #52263 passed	11 hrs 13 min 42 sec 3 days ago
	Andrii Rosa		fc94719	

Effective CI processes allocate enough resources to mitigate flaky tests

- *Flaky* tests might be dependent on timing (failing due to timeouts)
- Running tests without enough CPU/RAM can result in increased flaky failure rates and unreliable builds



CI in practice at Google

- Large scale example: Google TAP
 - 50,000 unique changes per-day, 4 billion test cases per-day
 - Pre-submit optimization: run fast tests for each individual change (before code review).
Block merge if they fail.
 - Then: run all affected tests; “build cop” monitors and acts immediately to roll-back or fix
 - Build cop monitors integration test runs
 - Average wait time to submit a change: 11 minutes

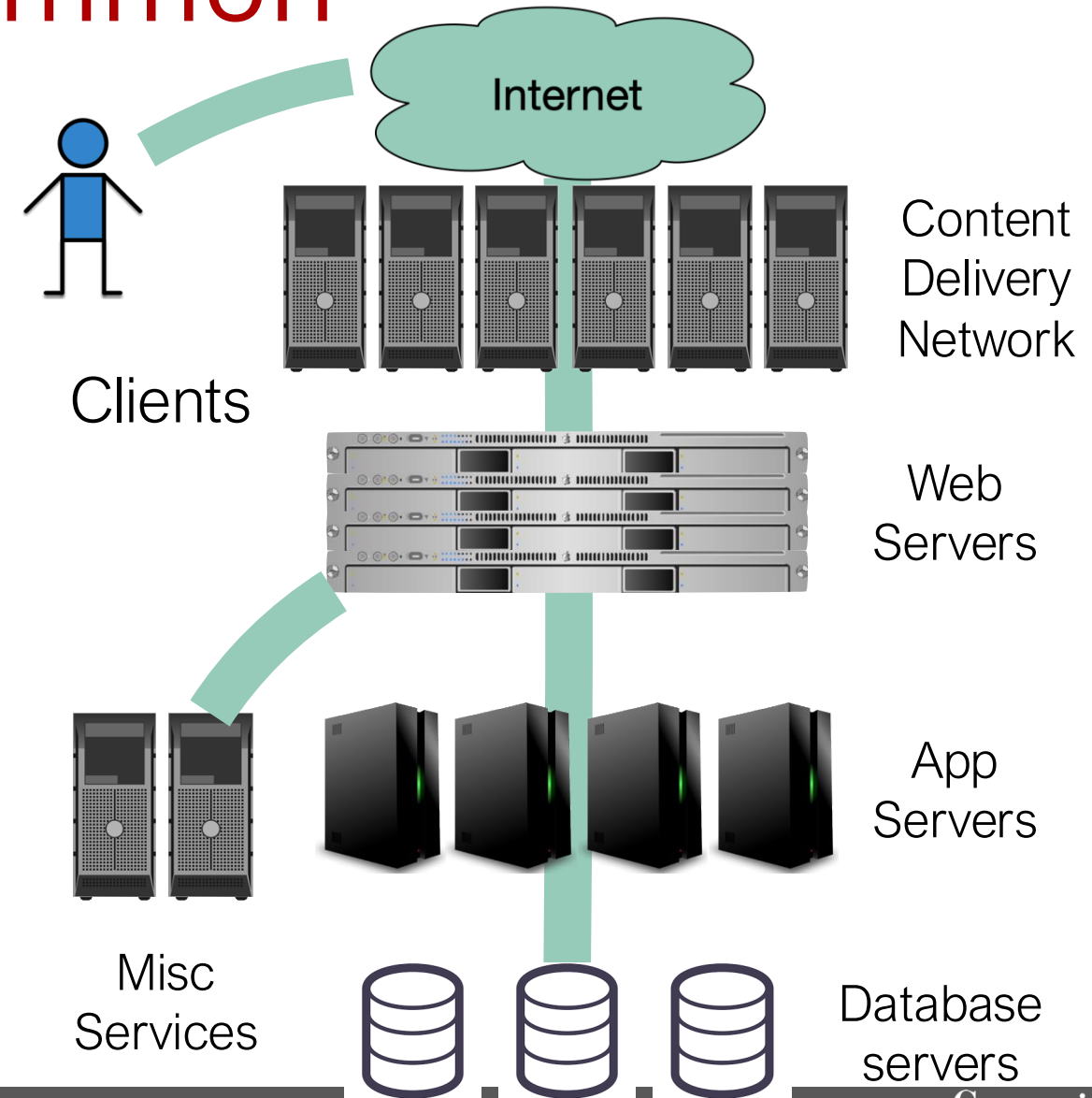
Challenge 2: “It works on my machine”

Cloud infrastructure, containers, and infrastructure as code

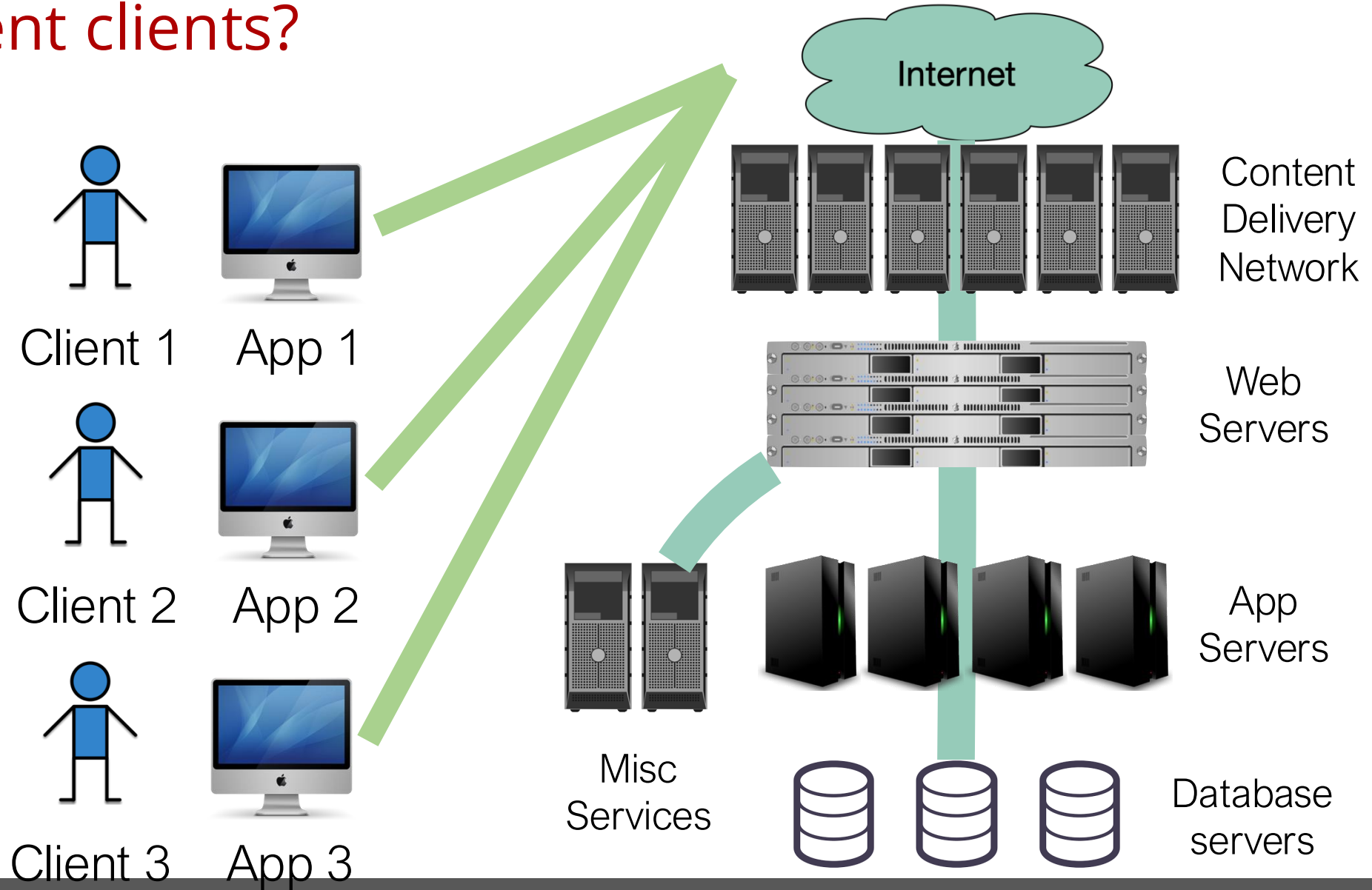
How can we continuously
update our software in
production?

Many apps rely on common infrastructure

- Content delivery network: caches static content “at the edge” (e.g. cloudflare, Akamai)
- Web servers: Speak HTTP, serve static content, load balance between app servers (e.g. haproxy, traefik)
- App servers: Runs our application (e.g. nodejs)
- Misc services: Logging, monitoring, firewall
- Database servers: Persistent data

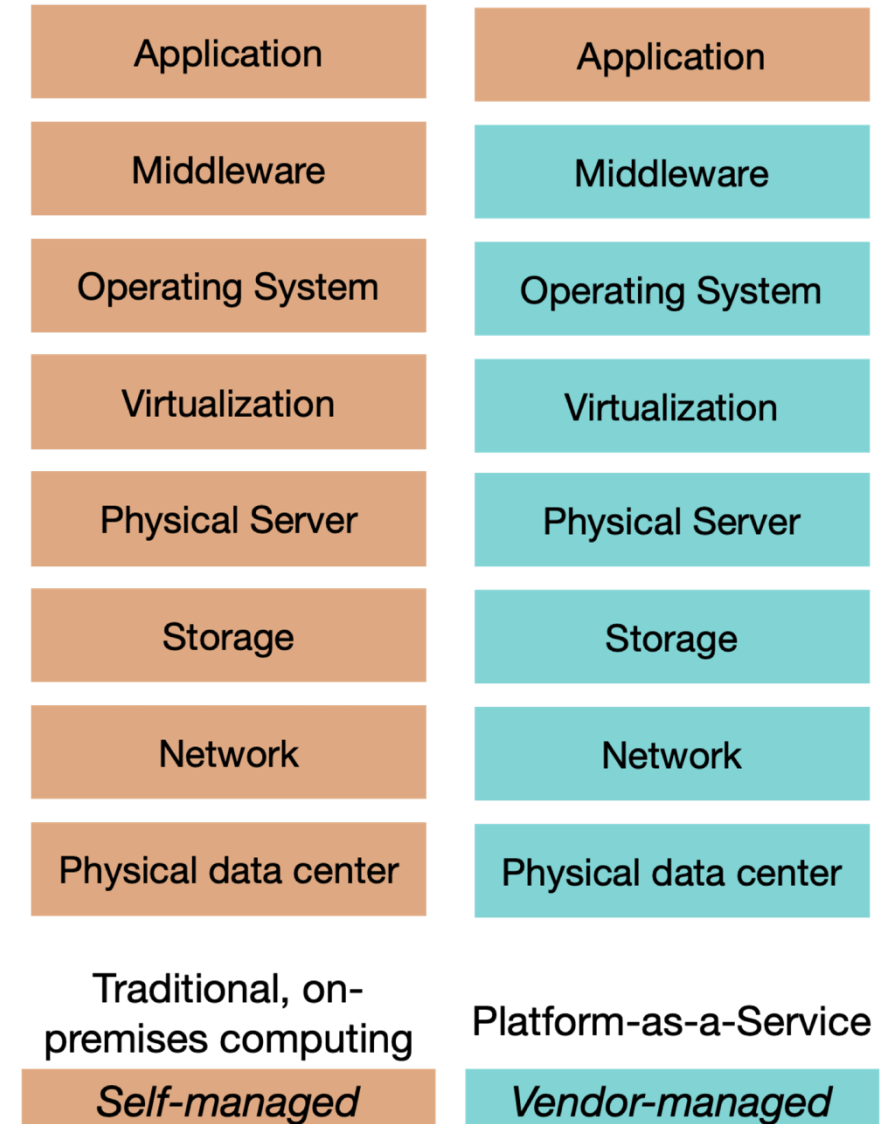


What parts of this infrastructure can be shared across different clients?



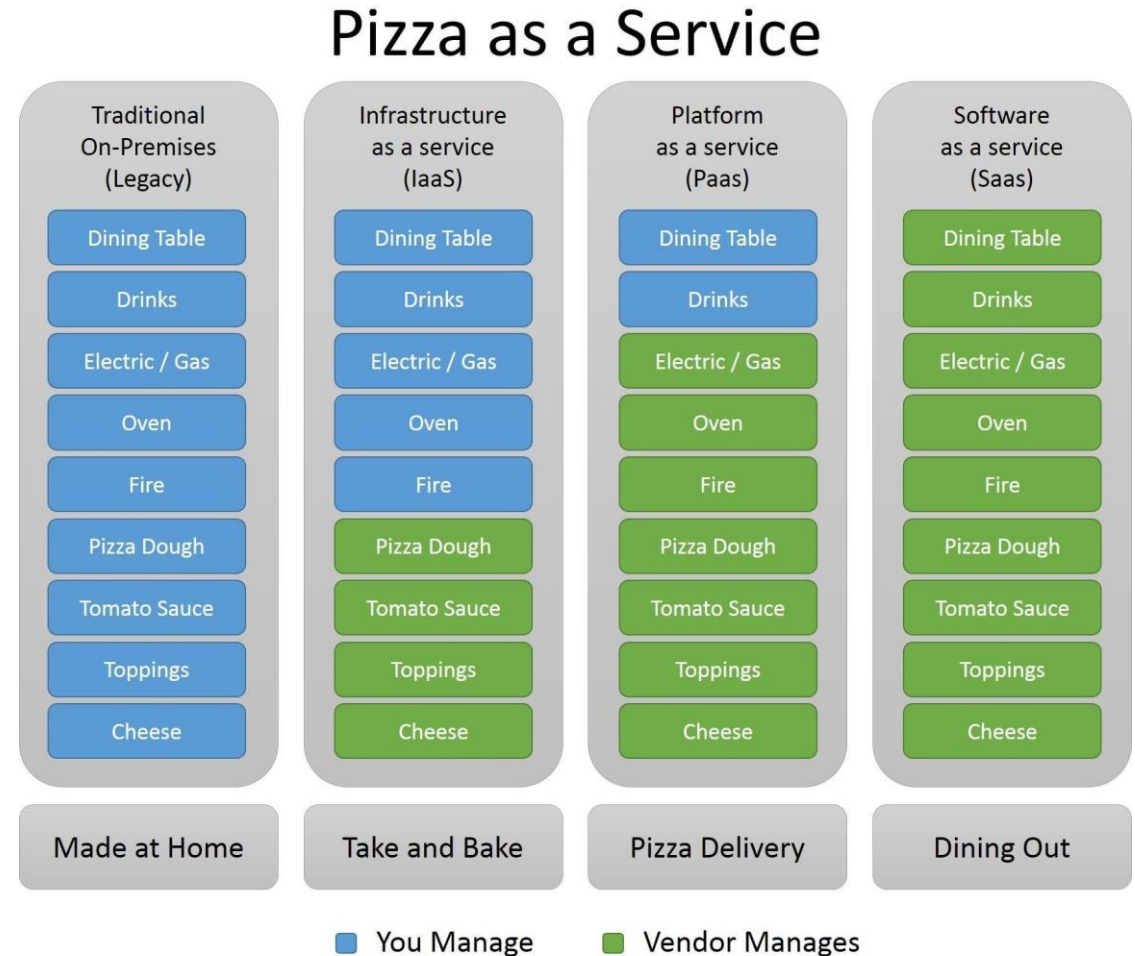
What is the infrastructure that needs to be shared?

- Our apps run on a “tall stack” of dependencies
- Traditionally this full stack is self-managed
- Cloud providers offer products that manage parts of that stack for us:
 - “Infrastructure as a service”
 - “Platform as a service”
 - “Software as a Service”



Shared infrastructure analogy: Pizza

- Four ways to get pizza:
Make yourself, take and bake, delivery, dine out
- Vendor manages different levels of the stack, achieving economies of scale
- When would you choose one over the other?



Pizza as a Service — by Albert Barron (unlicensed?)

Multi-Tenancy creates economies of scale

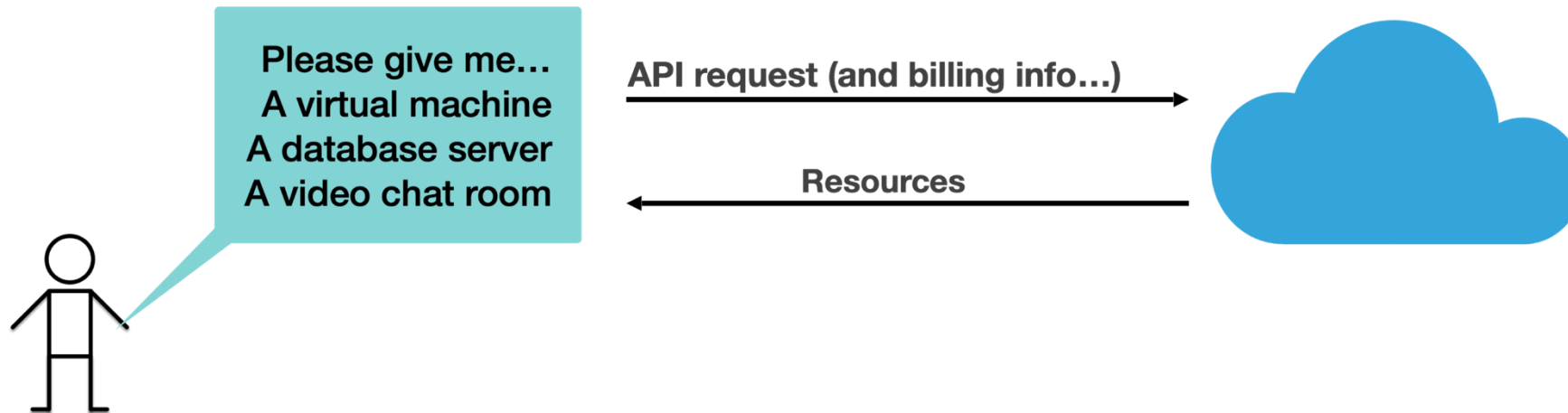
- At the physical level:
 - Multiple customers' physical machines in the same data center
 - Save on physical costs (centralize power, cooling, security, maintenance)
- At the physical server level:
 - Multiple customers' virtual machines in the same physical machine
 - Save on resource costs (utilize marginal computing capacity – CPUs, RAM, disk)
- At the application level:
 - Multiple customer's applications hosted in same virtual machine
 - Save on resource overhead (eliminate redundant infrastructure like OS)
- “Cloud” is the natural expansion of multi-tenancy at all levels

Cloud infrastructure scales elastically

- “Traditional” computing infrastructure requires capital investment
 - “Scaling up” means buying more hardware, or maintaining excess capacity for when scale is needed
 - “Scaling down” means selling hardware, or powering it off
- Cloud computing scales elastically:
 - “Scaling up” means allocating more shared resources
 - “Scaling down” means releasing resources into a pool
 - Billed on consumption (usually per-second, per-minute or per-hour)

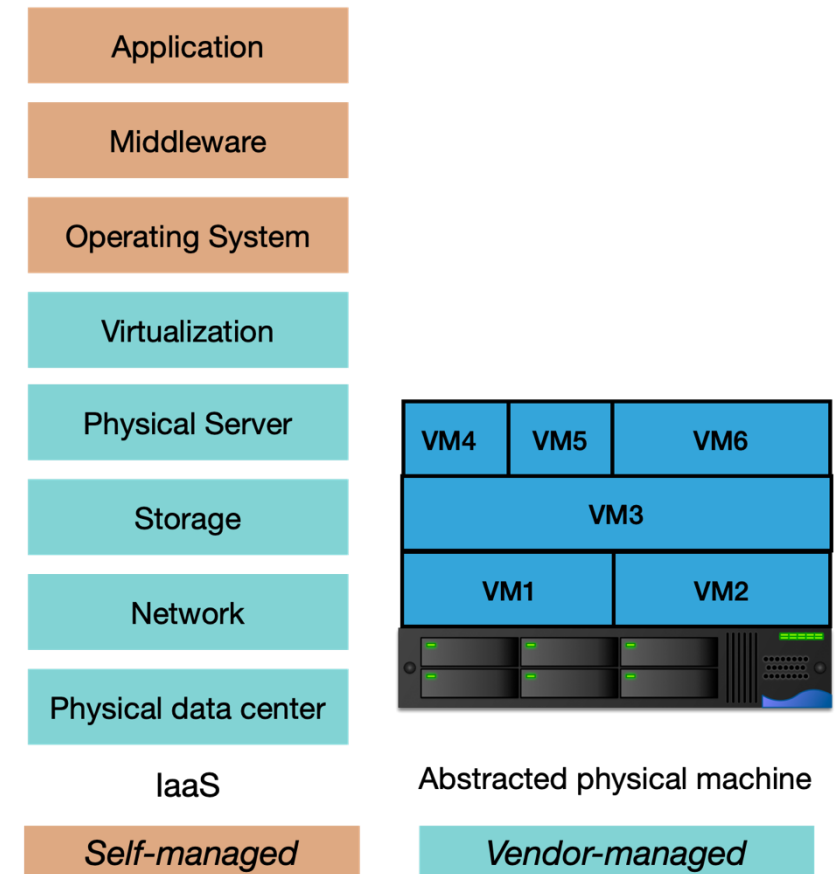
Cloud services gives on-demand access to infrastructure, “as a service”

- Vendor provides a service catalog of “X as a service” abstractions that provide infrastructure as a service
- API allows us to provision resources on-demand
- Transfers responsibility for managing the underlying infrastructure to a vendor



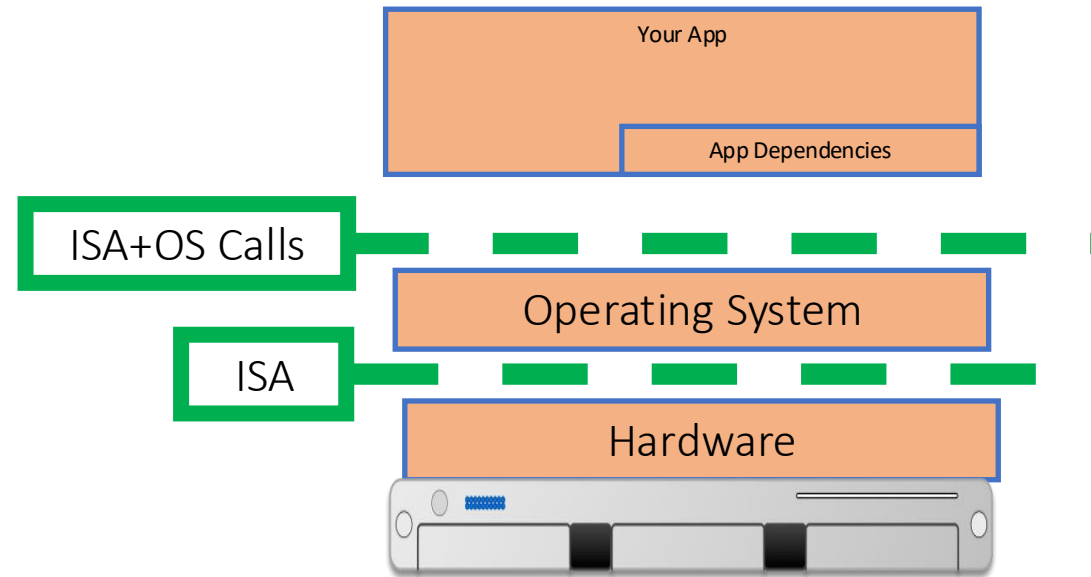
Infrastructure as a Service: Virtual Machines

- Virtual machines:
 - Virtualize a single large server into many smaller machines
 - Separates administration responsibilities for physical machine vs virtual machines
 - OS limits resource usage and guarantees quality per-VM
 - Each VM runs its own OS
 - Examples:
 - Cloud: Amazon EC2, Google Compute Engine, Azure
 - On-Premises: VMWare, Proxmox

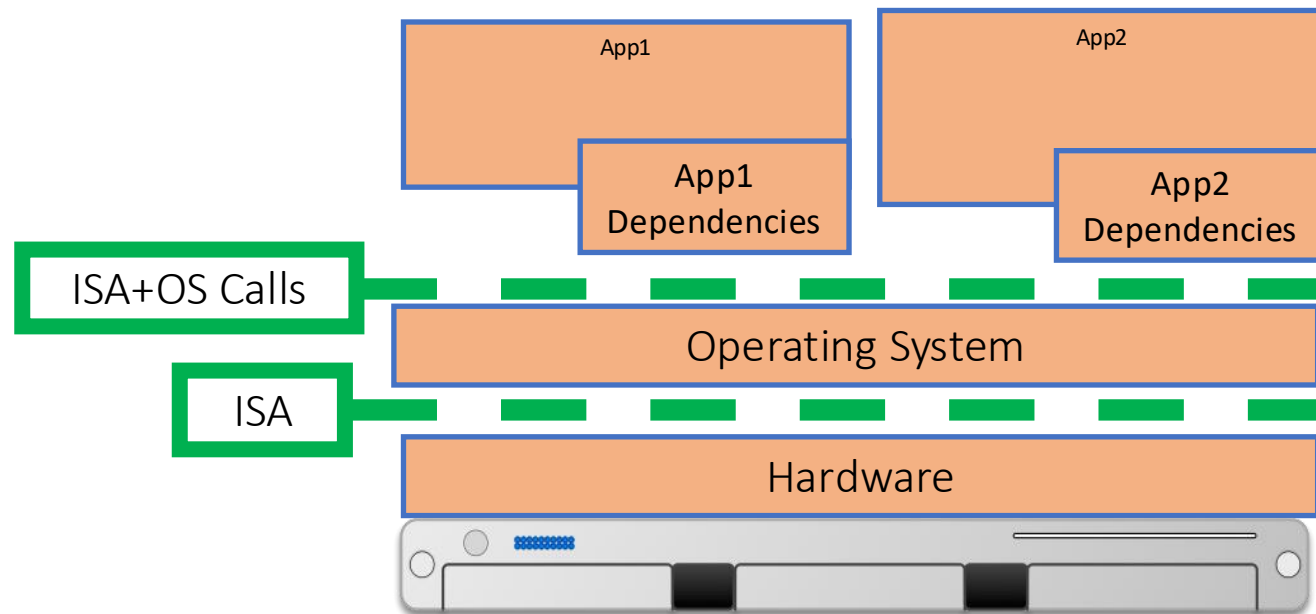


Let's look more closely at this software stack

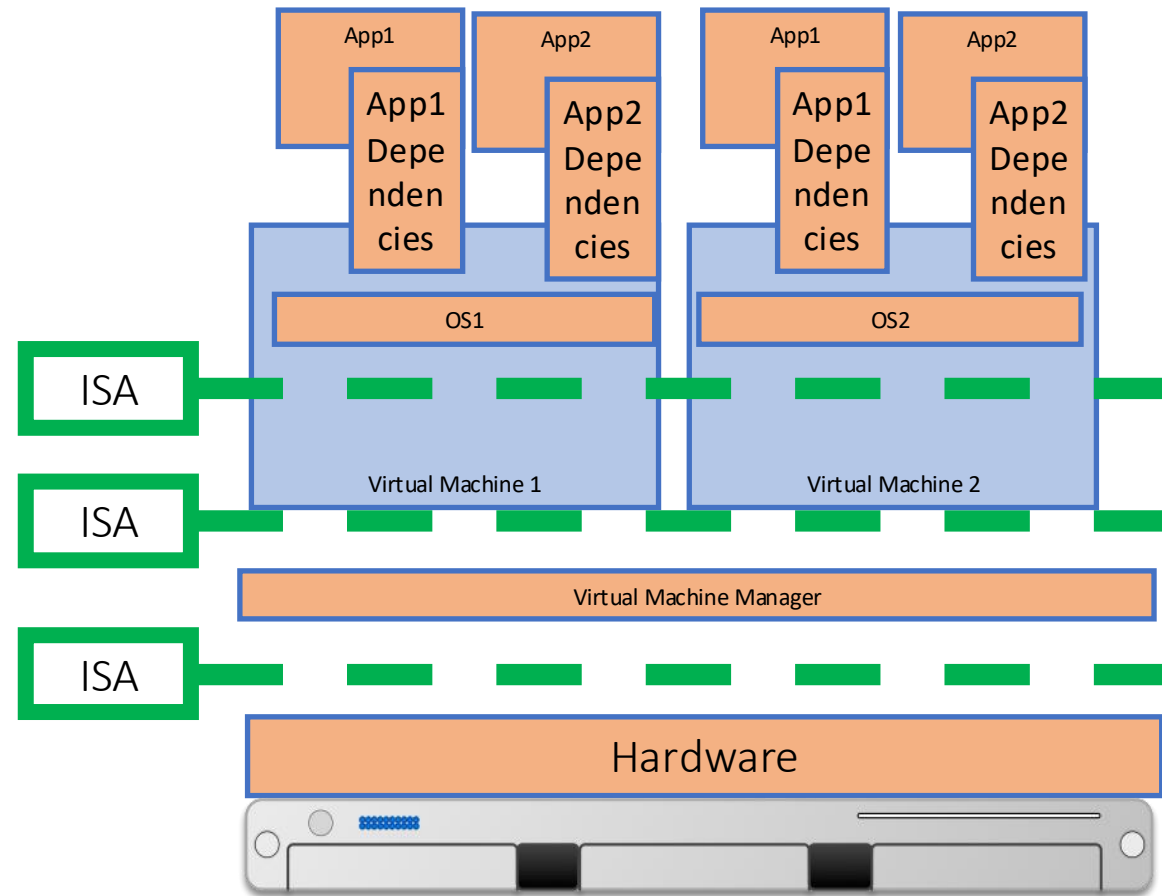
- The “instruction set” is an abstraction of the underlying hardware
- The operating system presents the same abstraction + OS calls.



The operating system allows several apps to share the underlying hardware



A virtual machine allows shared hardware



Virtual Machines facilitate multi-tenancy

- Multi-Tenancy
 - Multiple customers sharing same physical machine, oblivious to each other
- Decouples application from hardware
 - virtualization service can provide “live migration” transparent to the operating system, maximizing utilization
- Faster to provision and release
 - VM v. physical machines == ~mins v. ~hours

Virtual Machines to Containers

- Each VM contains a full operating system
- What if each application could run in the same (overall) operating system? Why have multiple copies?
- Advantages to smaller apps:
 - Faster to copy (and hence provision)
 - Consume less storage (base OS images are usually 3-10GB)

Containers run layered images, reducing storage space

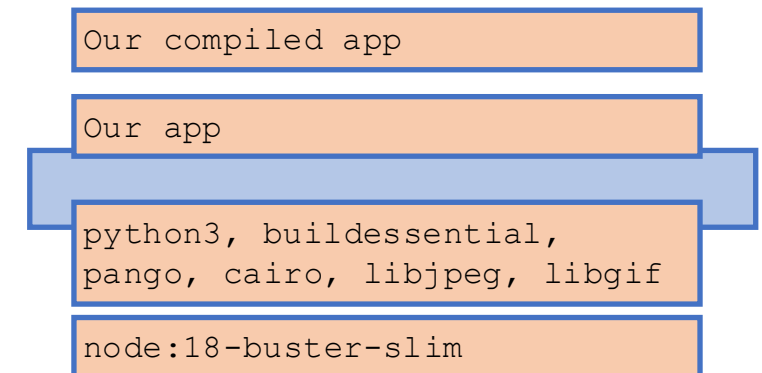
- Images are defined programmatically as a series of “build steps” (e.g. Dockerfile)
- Each step in the build becomes a “layer”
- Built images can be shared and cached
- To run a container, the layers are linked together with an “overlay” filesystem

```
FROM node:18-buster-slim
RUN apt-get update && apt-get install python3
  build-essential libpango1.0-dev libcairo2-dev
  libjpeg-dev libgif-dev -y

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY ./ /usr/src/app

RUN npm ci
RUN npm run build
CMD [ "npm", "start" ]
```

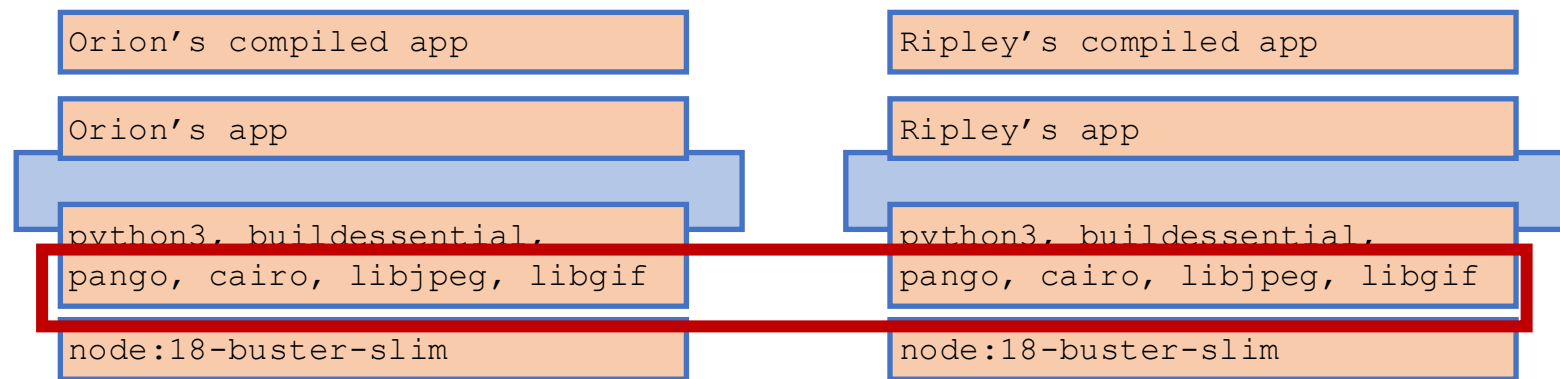
Example image specification (Dockerfile)



Example image, with layers shown

Containers run layered images, reducing storage space

- Many images may share the *same* lower layers (e.g. OS, NodeJS, some system dependencies)
- Layers are shared between images
- Multi-tenancy: N running containers only require *one* copy of each layer (they are read-only)



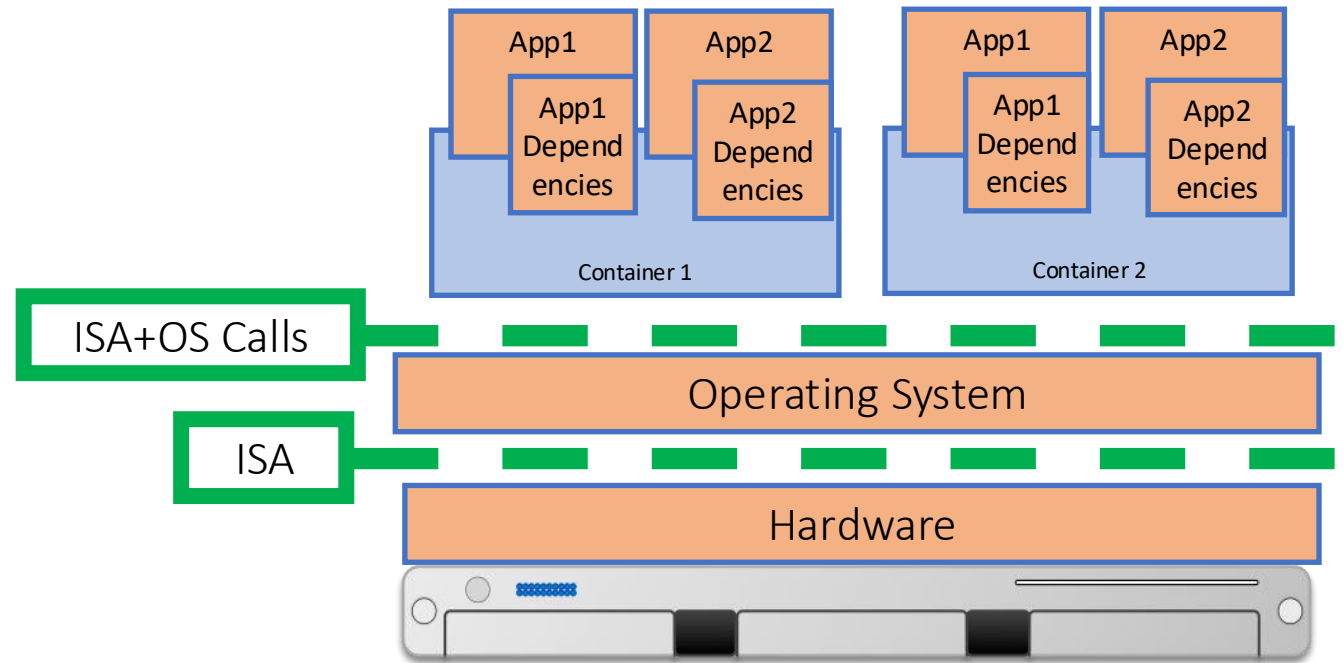
Two images, sharing two layers

A container contains your apps and all their dependencies

- Each application is encapsulated in a “lightweight container,” includes:
 - System libraries (e.g. glibc)
 - External dependencies (e.g. nodejs)
- “Lightweight” in that container images are smaller than VM images - multi tenant containers run in the OS
- Cloud providers offer “containers as a service” (Amazon ECS Fargate, Azure Kubernetes, Google Kubernetes)

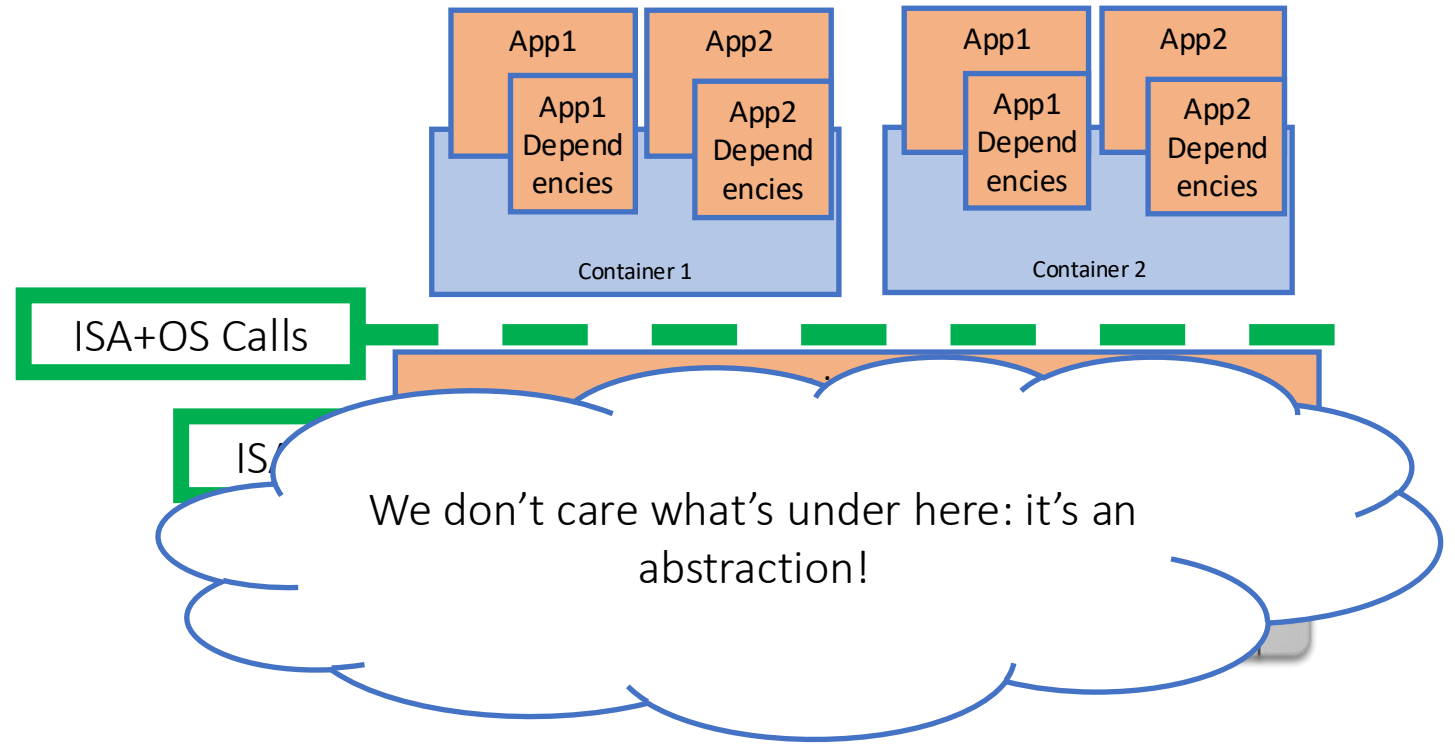
A container contains your apps and all their dependencies

- You might put several apps in a single container, together with their dependencies
- Might have only one copy of shared dependencies



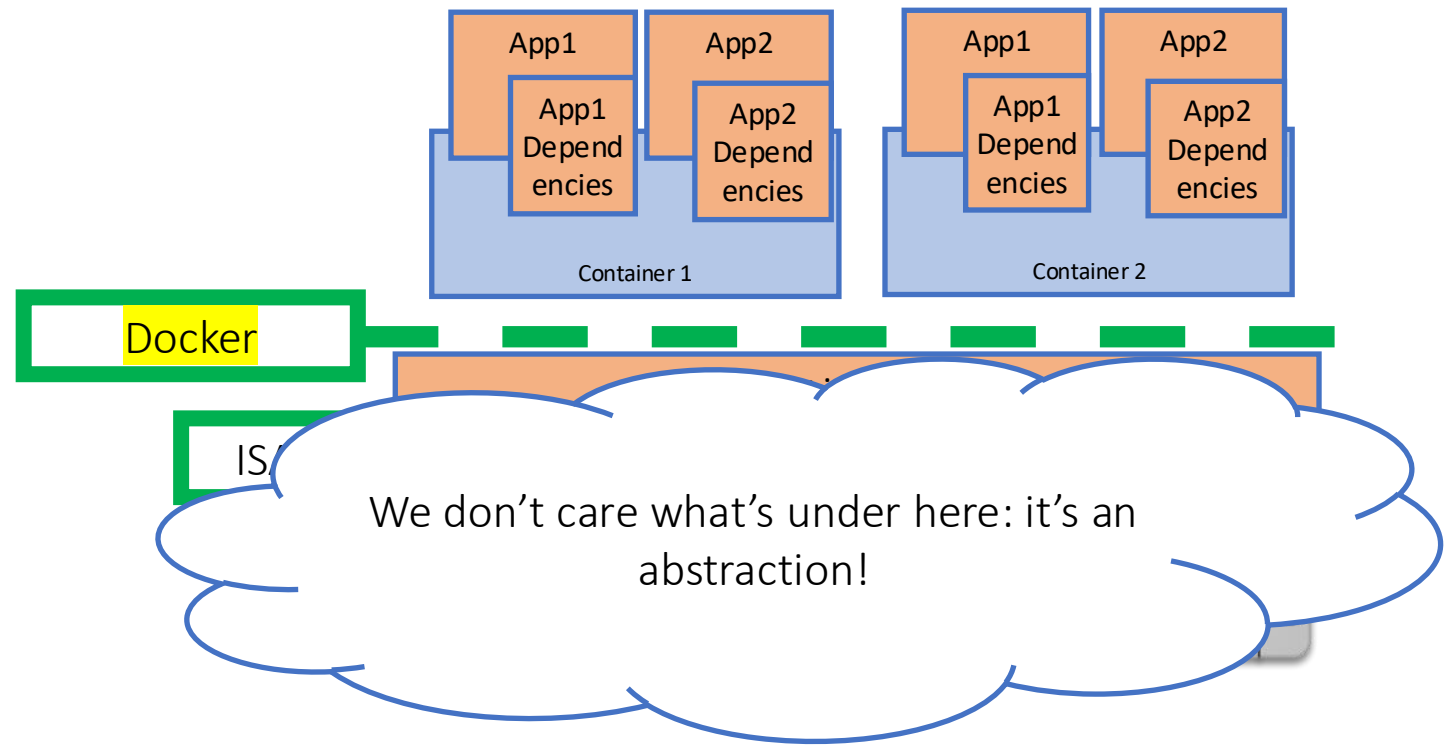
XaaS: Containers as a Service

- Vendor supplies an on-demand instance of an operating system
 - Eg: Linux version NN
- Vendor is free to implement that instance in a way that optimizes costs across many clients.



Docker is the prevailing container platform

- Docker provides a standardized interface for your container to use
- Many vendors will host your Docker container
- An open standard for containers also exists ("OCI")



Code**Blame**

19 lines (14 loc) · 520 Bytes



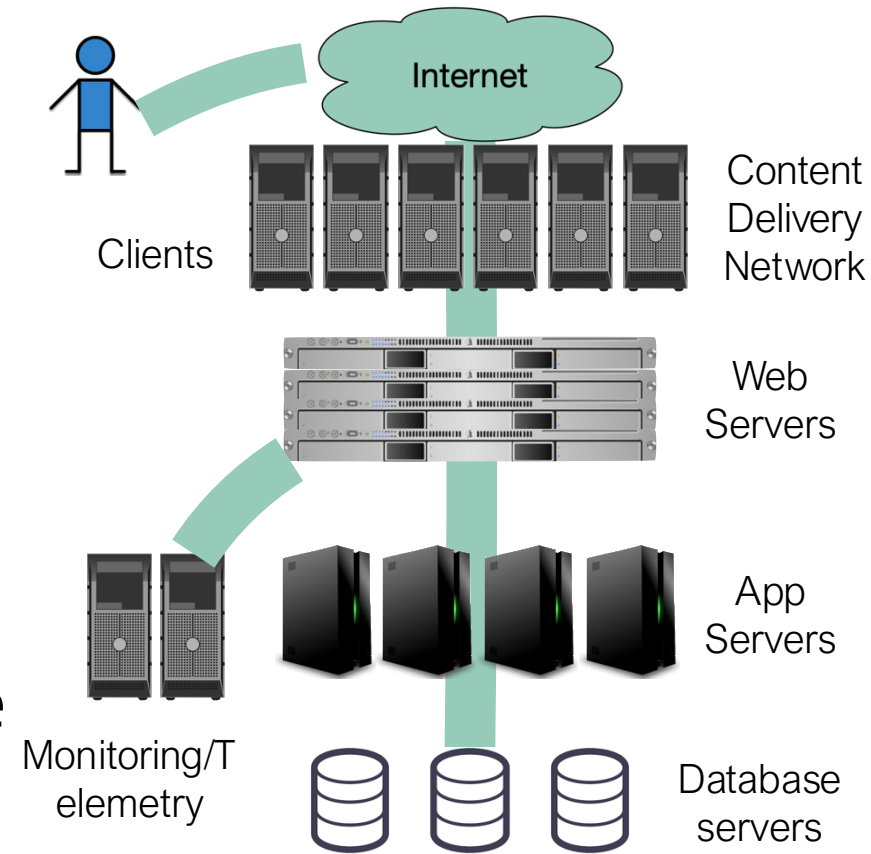
```
1  # specifies the Parent Image from which you are building.
2  FROM python:3.9
3
4  # specify the working directory for the image
5  WORKDIR /code
6
7  # TODO
8  # copy and install requirements (simple and cache-friendly)
9  COPY requirements.txt /code/requirements.txt
10 RUN pip install --upgrade pip \
11     && pip install -r /code/requirements.txt
12
13 # copy application code
14 COPY ./app /code/app
15
16 EXPOSE 8080
17
18 # uvicorn will import the FastAPI "app" object from app/main.py
19 CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Tradeoffs between VMs and Containers

- Performance is comparable
- Each VM has a copy of the OS and libraries
 - Higher resource overhead
 - Slower to provision
 - Support for wider variety of OS'
- Containers are “lightweight”
 - Lower resource overhead
 - Faster to provision
 - Potential for compatibility issues, especially with older software

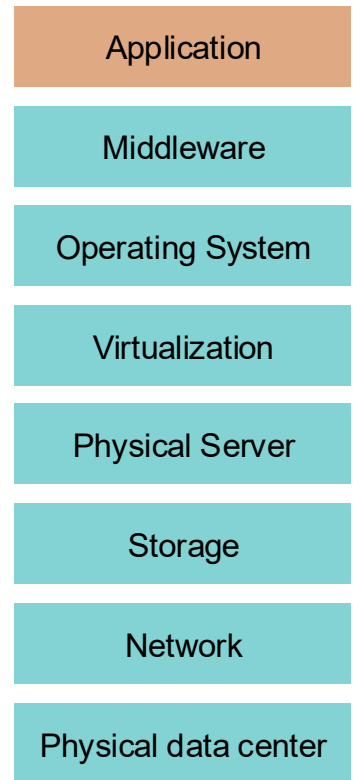
Platform-as-a-Service: vendor supplies OS + middleware

- Middleware is the stuff between our app and a user's requests:
 - Content delivery networks: Cache static content
 - Web Servers: route client requests to one of our app containers
 - Application server: run our handler functions in response to requests from load balancer
 - Monitoring/telemetry: log requests, response times and errors
- Cloud vendors provide managed middleware platforms too: "Platform as a Service"



PaaS is often the simplest choice for app deployment

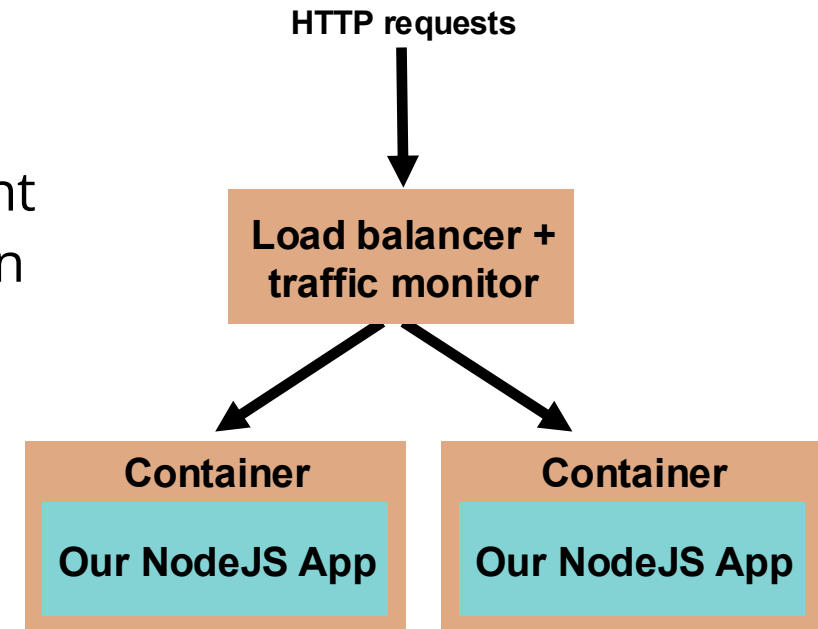
- **Platform-as-a-Service** provides components most apps need, fully managed by the vendor: load balancer, monitoring, application server
- Some PaaS run your app in a container: Heroku, AWS Elastic Beanstalk, Google App Engine, Railway, Vercel...
- Other PaaS run your apps as individual functions/event handlers: AWS Lambda, Google Cloud Functions, Azure Functions
- Other PaaS provide databases and authentication, and run your functions/event handlers: Google Firebase, Back4App



PaaS

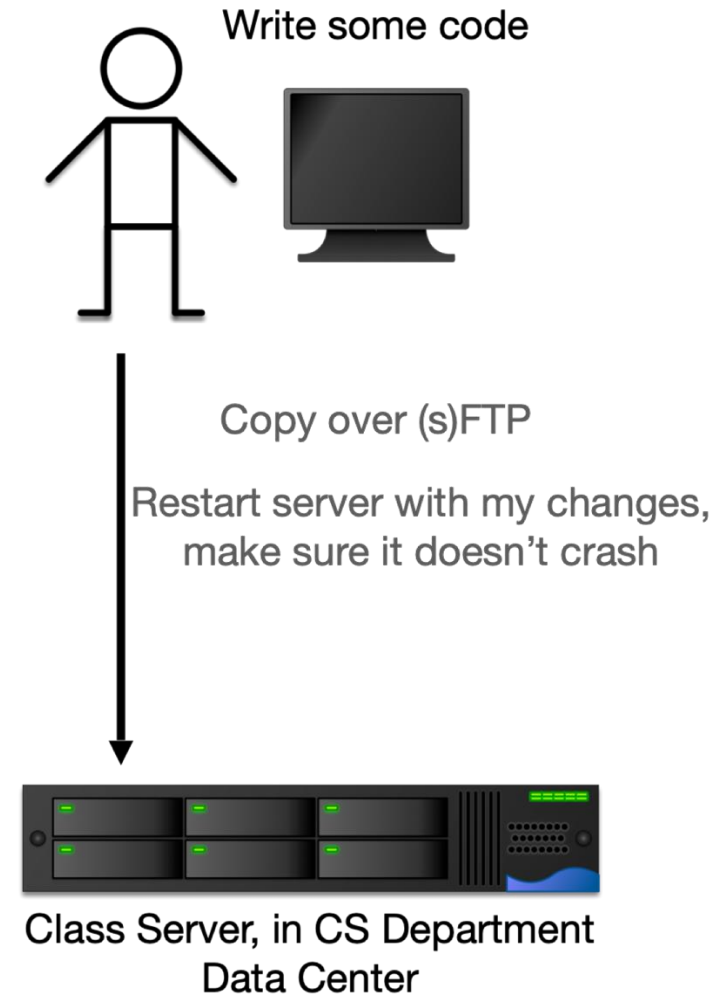
PaaS in the style of Heroku runs containers

- Takes a web app as input
 - Provide an entry point to code, e.g. “npm start”, or optionally, a container specification
- Hosts web app at chosen URL, can scale resources up/down on-demand
 - Load balancer fully managed by Heroku, scaling transparent
 - Auto-scale down to use no resources, spins up container on reception of a request
 - Dashboard for monitoring/reporting
- Newcomers provide similar functionality (Vercel, Railway, etc)
- Host PaaS on-premises, too (Caprover)



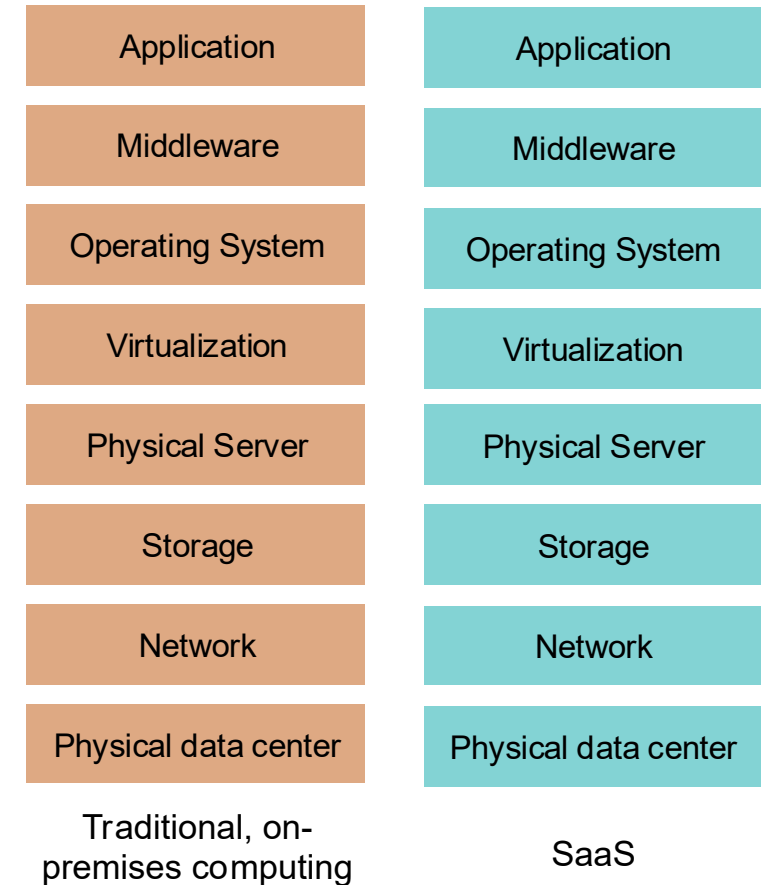
How to deploy web apps?

- What we need:
 - A server that can run our application
 - A network that is configured to route requests from an address to that server
- Questions to think about:
 - What software do we need to run besides our application code? (Databases, caches, etc?)
 - Where does this server come from? (Buy/Borrow?)
 - Who else gets to use this server? (Multi-tenancy or exclusive?)
 - Who maintains the server and software? (Updates OS, libraries, etc?)



Self-managed vs Vendor-managed Infrastructure

- Consider who manages each tier in the stack
- Benefits to vendor-managed options:
 - More ways to reduce resource consumption, improve resource utilization
 - Less management burden
 - Less capital investment, more flexibility in scaling
- Benefits to self-managed options:
 - Greater flexibility to migrate between software platforms
 - Potentially less operating expenses



Motivating scenario: Failed Deployment at Knight Capital

Knightmare: A DevOps Cautionary Tale

👤 D7 📁 DevOps ⌚ April 17, 2014 📖 6 Minutes

I was speaking at a conference last year on the topics of DevOps, Configuration as Code, and Continuous Delivery and used the following story to demonstrate the importance making deployments fully automated and repeatable as part of a DevOps/Continuous Delivery initiative. Since that conference I have been asked by several people to share the story through my blog. This story is true – this really happened. This is my telling of the story based on what I have read (I was not involved in this).



This is the story of how a company with nearly \$400 million in assets went bankrupt in minutes because of a failed deployment.

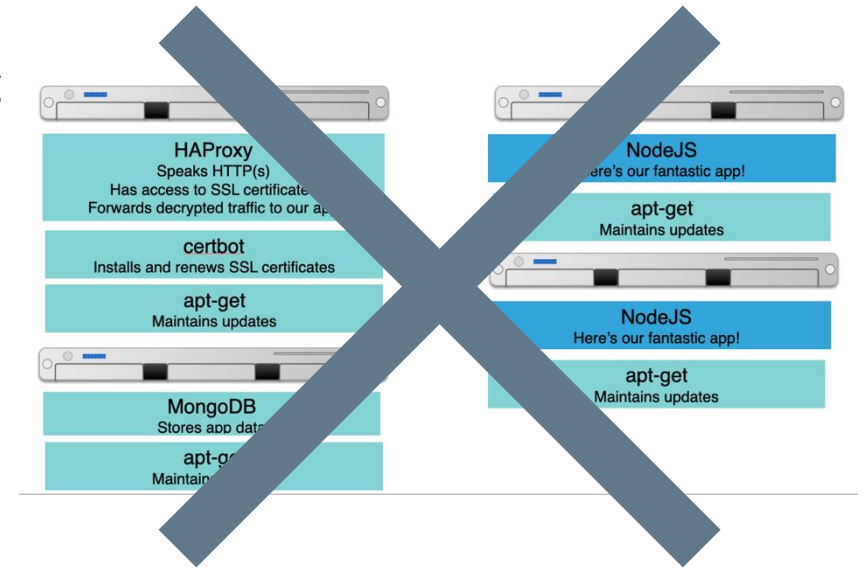
“In the week before go-live, a Knight engineer manually deployed the new RLP code in SMARS to its 8 servers. However, he made a mistake and did not copy the new code to one of the servers. Knight did not have a second engineer review the deployment, and neither was there an automated system to alert anyone to the discrepancy. “

What could Knight capital have done better?

- Use capture/replay testing instead of driving market conditions in a test
- Avoid including “test” code in production deployments
- Automate deployments
- Define and monitor risk-based KPIs
- Create checklists for responding to incidents

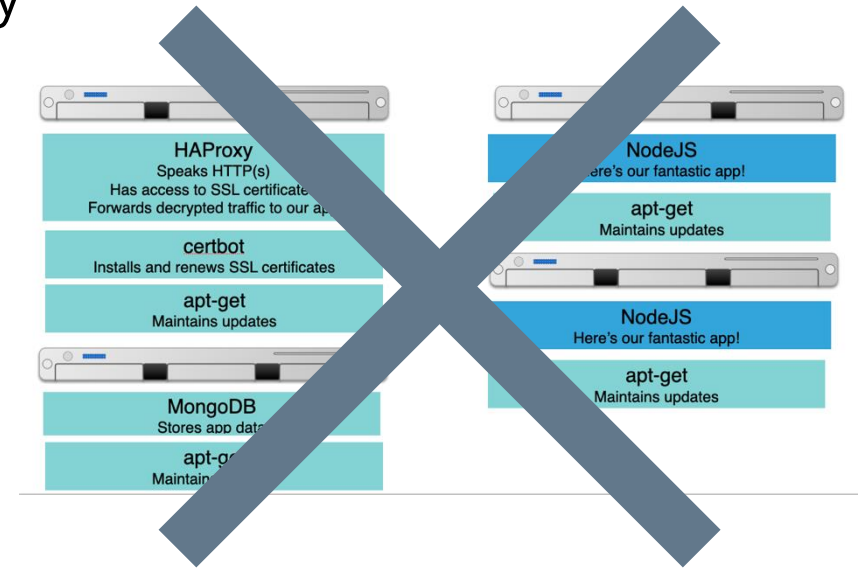
Continuous Delivery Relies on Infrastructure As Code

- Provisioning servers is tedious and error prone
 - Deploy a VM, then ssh to it, install some packages, etc
- Keeping servers up-to-date is also a struggle
- Ideal:
 - “Give me HAProxy with some configuration file, and keep that configuration in a git repo, and when I change it, roll out an update”
 - “Give me some containers running my NodeJS app, and when I update my app, roll it out to those containers”
 - “Give me a bunch of servers with MongoDB set up in a cluster”



Infrastructure as Code

- Goal: Create a system that, when run, can automatically bring physical or virtual machines to some configured state
 - These configurations can then go into version control, code review, etc
- Metaphor: “Recipes” for configuring servers, organized into “cookbooks”
 - Engineers define “healthy” states for infrastructure, then system automatically provisions, validates, and (if needed) repairs deployed resources
 - “Oh, this is how they do things at Amazon” - Inspiration for [Chef, c 2009](#)
 - Other tools with similar aims: Puppet (c 2005), Ansible (c 2012)



Challenge 3: Running hundreds of containers

Container orchestration with Kubernetes

Kubernetes orchestrates containers across a cluster

- Open-source container orchestrator, based on Google's internal Borg system
- You say what should run; Kubernetes decides where and keeps it running:
 - Schedules containers onto machines
 - Restarts or replaces containers that fail
 - Scales the number of copies up and down
 - Rolls out new versions a few at a time
 - Gives services stable names and load balancing
- Every cloud offers it as a service: GKE (Google), EKS (Amazon), AKS (Azure)

Omega: flexible, scalable schedulers for large compute clusters

Mehr Schwartzkopf¹ · Andy Koninski² · Michael Abd-El-Malek³ · John Wilkes³
¹University of Cambridge Computer Laboratory ²University of California Berkeley ³Google Inc.
mschwartz@cam.ac.uk andy@cs.berkeley.edu michael@cs.berkeley.edu john@cs.berkeley.edu

Abstract

Increasing scale and the need for rapid response to changing requirements are hard to meet with current monolithic, one-size-fits-all schedulers. This motivates the need for a scheduler that can be deployed, reconfigured, and extended, and will eventually limit cluster growth. We present a novel approach to address these needs using partitioned, shared state, and lock-free optimistic concurrency control. We compare this approach to existing cluster schedulers, evaluate how much interference between schedulers causes and how much it matters to practice, present some techniques to alleviate it, and finally discuss a new case highlighting the advantages of our approach – all driven by real-life Google production workloads.

Categories and Subject Descriptors: D.4.7 [Operating Systems]: Organization and Design—Distributed systems; K.4.1 [Management of computing and information systems]: System Management—Centralization/centralization

Keywords: Cluster scheduling, optimistic concurrency control

1. Introduction

Large-scale compute clusters are expensive, so it is important to use them well. Utilization and efficiency can be improved by sharing a set of workloads on the same machines. CPUs and memory-intensive jobs, small and large ones, and a mix of batch and low-latency jobs – ones that serve and user requests or provide infrastructure services such as storage, networking or logging. This coexistence reduces the amount of hardware required for a workload, but it makes the scheduling problem (assigning jobs to machines) more complicated: a wider range of requirements

“Work done while writing Omega for Google Inc.”

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted by ACM, provided that the fee code for users to reproduce the work is paid to ACM. Copyright 2015 ACM 978-1-4558-2656-9/15/0001...\$15.00. DOI: <http://dx.doi.org/10.1145/2656909>

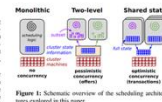


Figure 1: Schematic overview of the scheduling architecture.

and policies have to be taken into account. Monolithic schedulers have to be replaced by a more flexible approach.

We identified the two previous scheduler architectures shown in Figure 1. Monolithic schedulers use a single, centralized scheduling algorithm for all jobs (see existing scheduler as one of these). Two-level schedulers have a single active resource manager that offers compute resources to multiple parallel, independent “scheduler frameworks”, as in Moon [13] and Flakopogon [14].

Neither of these models satisfied our needs. Monolithic schedulers do not make it easy to add new policies and specialized implementations, and they are not well suited to the cluster sizes we are planning for. Two-level scheduling architectures do appear to provide flexibility and modularity, but in practice they concentrate resource availability and locking algorithms, both tasks, and make it hard to place difficult-to-schedule “sticky” jobs in a more distributed fashion across the state of the entire cluster.

Our solution is a new parallel scheduler architecture that instead shares state, using lock-free optimistic concurrency control, to achieve both implementation extensibility and performance scalability. This architecture is being used in

Large-scale cluster management at Google with Borg

Abhishek Verma¹ · Luis Pedrosa² · Madhukar Kuzumapu³ · David Oppenheimer¹ · Eric Tune¹ · John Wilkes³
¹Google Inc. ²Google Inc. ³Google 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, overcommitment, 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, some service integration, and 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 admin, schedules, runs, monitors, and manages the full range of applications that Google runs. This paper explains how.

Borg provides three main benefits: (1) It makes the details of resource management and failure handling go away from users, focusing on application development instead. (2) It operates with high availability and availability, and supports applications that do the same; and (3) It runs on workloads across tens of thousands of machines effectively. Borg is one of the first systems to address these issues, but if one of the few operating at the scale, with this degree of reliability and performance. This paper is organized around these topics, covering:

2. The user perspective

Borg’s users are Google developers and system administrators (aka 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 (library). 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 collects a heterogeneous workload with two main parts. The first is long-running services that should “never” go down, and handle short-lived latency-sensitive requests (e.g., go to a few hundred ms). Both services are used for end-user-facing products, such as Gmail, Google Docs, and with various, and for internal infrastructure services (e.g., BigTable). The second is batch jobs that take from a few minutes to a few days to complete, those that make heavy reliance on short-term performance fluctuations. The workload only varies across cells, which are different sizes of applications depending on their major tenants (e.g., some cells are quite batch-intensive, and jobs varies over time: batch jobs

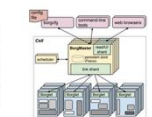
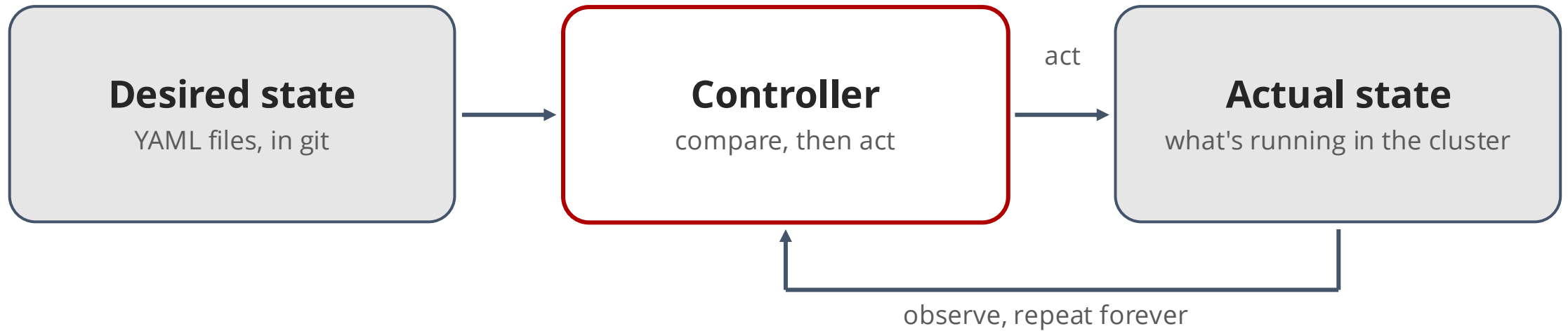


Figure 2: The high-level architecture of Borg. (It is very different from operating Borg in production for more than a decade.)

Omega (EuroSys ’13)

Borg (EuroSys ’15)

You declare the desired state; Kubernetes makes it so



You edit replicas: 3 → 5

Controller sees 3 running, wants 5, starts 2 more

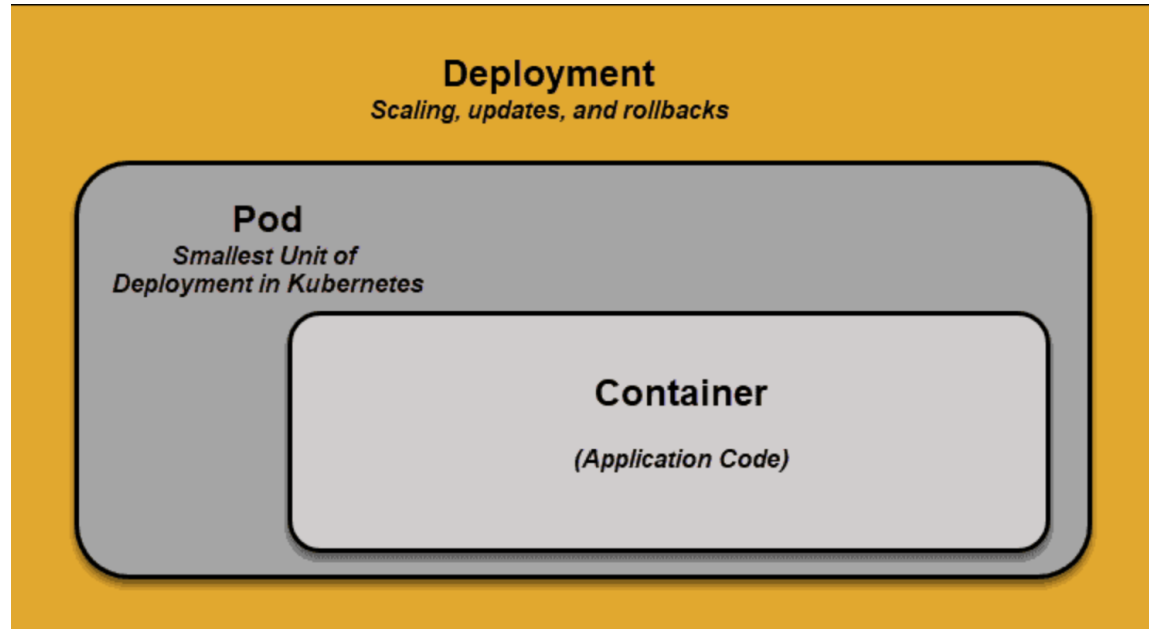
A machine dies, taking 2 pods

Controller sees 1 running, wants 3, schedules 2 elsewhere

You change the image tag

Controller replaces old pods with new ones, a few at a time

Containers, pods, deployments, and services



- **Container:** your app image, built from a Dockerfile
- **Pod:** one or more containers scheduled together; the smallest unit Kubernetes runs
- **Deployment:** how many copies of a pod should exist, and how to roll out new versions
- **Service:** a stable name and load balancer in front of pods, which come and go

Image: phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams

A Deployment and a Service are just YAML in your repo

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodebb
spec:
  replicas: 3
  selector:
    matchLabels: {app: nodebb}
  template:
    metadata:
      labels: {app: nodebb}
    spec:
      containers:
        - name: nodebb
          image: ghcr.io/my-team/nodebb:3f9c2a1
          ports:
            - containerPort: 4567
```

replicas: how many copies should exist. **image:** tagged with the commit SHA that CI built

```
apiVersion: v1
kind: Service
metadata:
  name: nodebb
spec:
  type: LoadBalancer
  selector: {app: nodebb}
  ports:
    - port: 80
      targetPort: 4567
```

selector: send traffic to any pod labeled app: nodebb. Clients use the Service, never individual pods

Kubernetes architecture

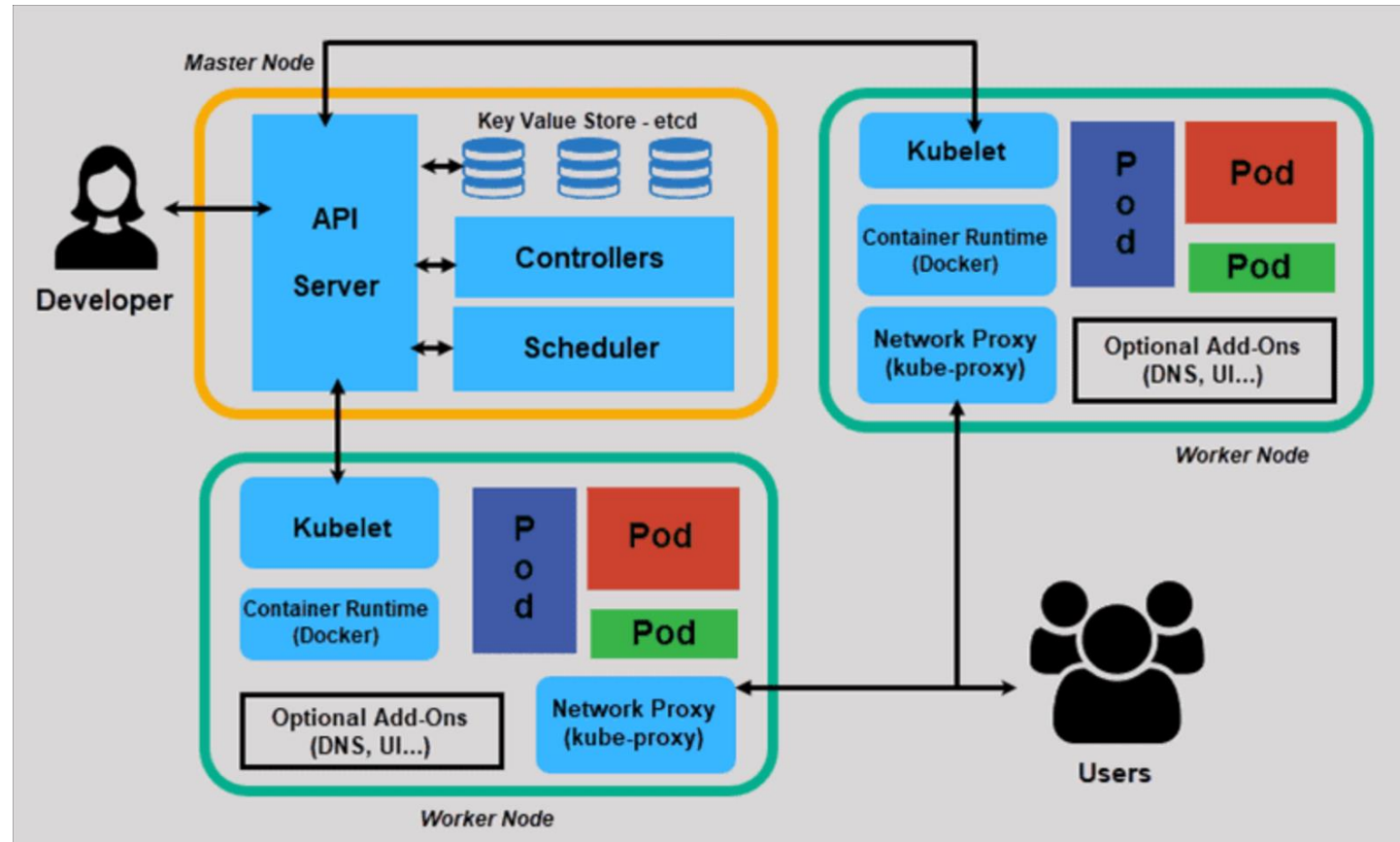


Image: phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams

Kubernetes handles crashes and rollouts for you

```
$ kubectl get pods
nodebb-7c9f-a1    Running
nodebb-7c9f-b2    Running
nodebb-7c9f-c3    Running

$ kubectl delete pod nodebb-7c9f-a1
$ kubectl get pods    # a replacement appears

$ kubectl set image deployment/nodebb \
    nodebb=ghcr.io/my-team/nodebb:8d41e07
$ kubectl rollout status deployment/nodebb
$ kubectl rollout undo deployment/nodebb
```

- **Self-healing:** delete or crash a pod and a new one takes its place
- **Rolling updates by default:** replace pods a few at a time
- **Readiness probes:** new pods get traffic only after they report ready
- **Rollback:** one command back to the previous version

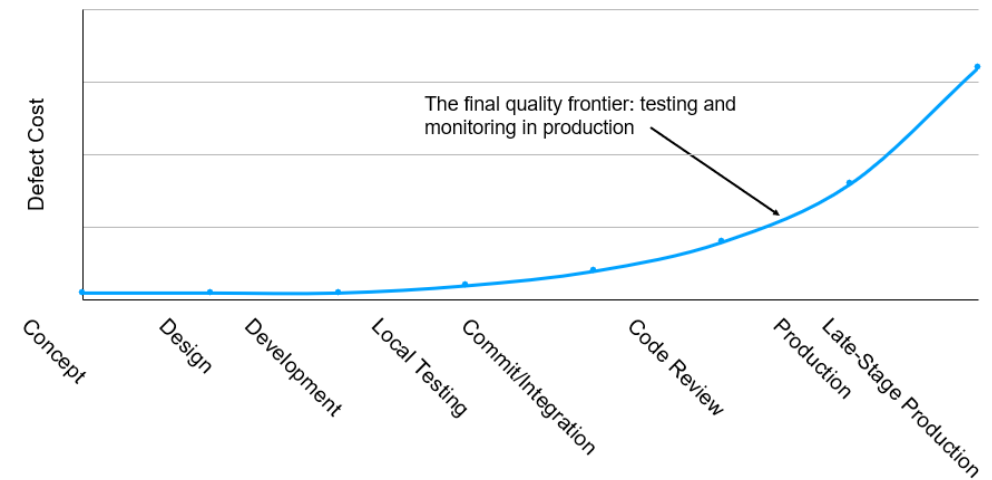
But Kubernetes only knows whether your pods are running, not whether the new version is correct.

Challenge 4: Every release is a bet

Continuous delivery and deployment strategies

Continuous Delivery

- “Faster is safer”: Key values of continuous delivery
 - Release frequently, in small batches
 - Maintain key performance indicators to evaluate the impact of updates
 - Phase roll-outs
 - Evaluate business impact of new features



Continuous Delivery != Immediate Delivery

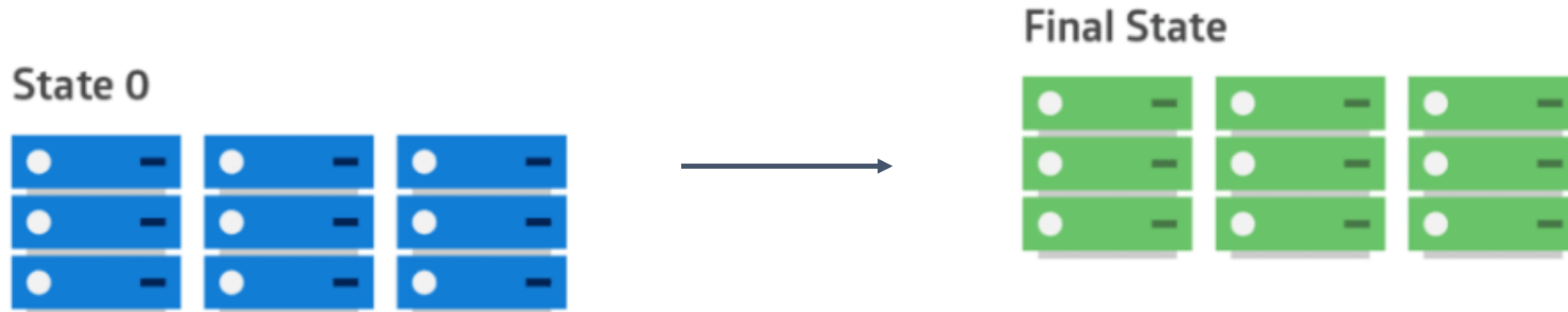
- Even if you are deploying every day (“continuously”), you still have some latency
- A new feature I develop today won't be released today
- But, a new feature I develop today can begin the **release pipeline** today (minimizes risk)
- **Release Engineer**: gatekeeper who decides when something is ready to go out, oversees the actual deployment process

Fast to deploy, slow to release

- **Deploy:** put the new code on production servers
- **Release:** turn it on for users
- Separating the two lets teams ship small changes constantly while controlling who sees them
- Chuck Rossi at Facebook: *“Get your s*** in, fix it in production”*



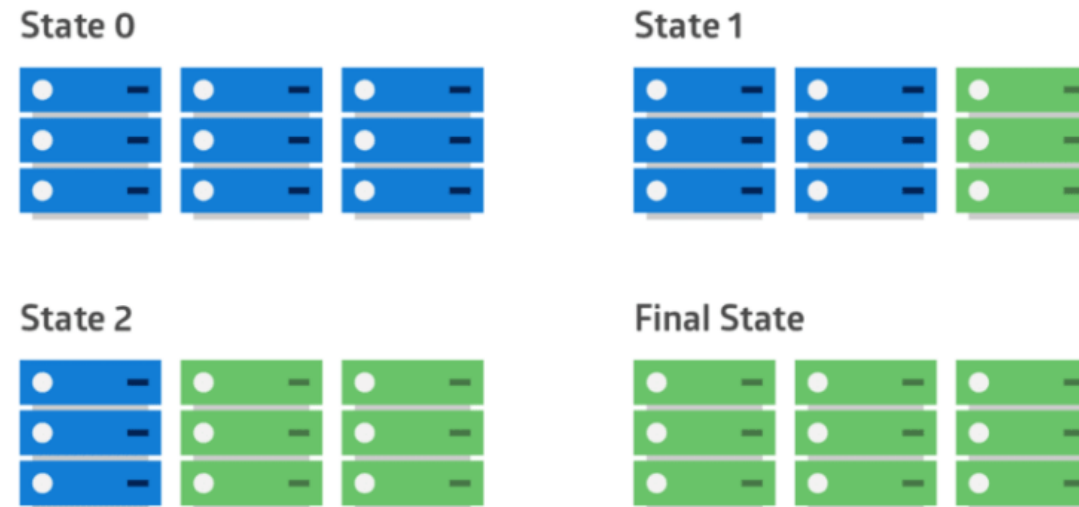
“Big bang” deployments



Every server switches to the new version at once. Simple and fast, and every bug reaches every user.

Image: dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3

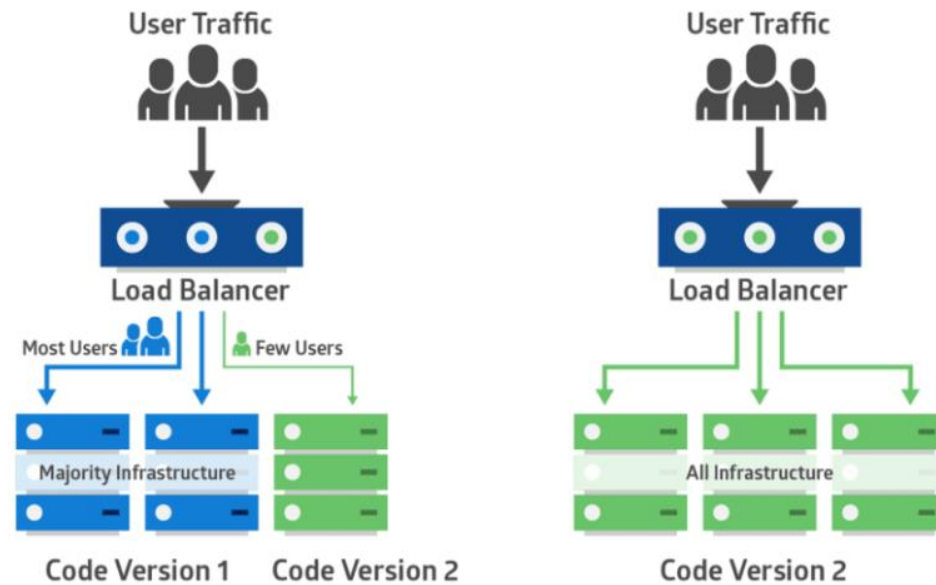
Rolling deployments



Replace servers a few at a time. No downtime, but old and new versions serve traffic side by side for a while.

Image: dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3

Canary deployments



Send a small slice of traffic to the new version, compare its health to the old version, then widen or roll back.

Image: dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3

Back to CrowdStrike: what would a canary have changed?

What happened

Every online Windows sensor, 04:09–05:27 UTC

With a canary

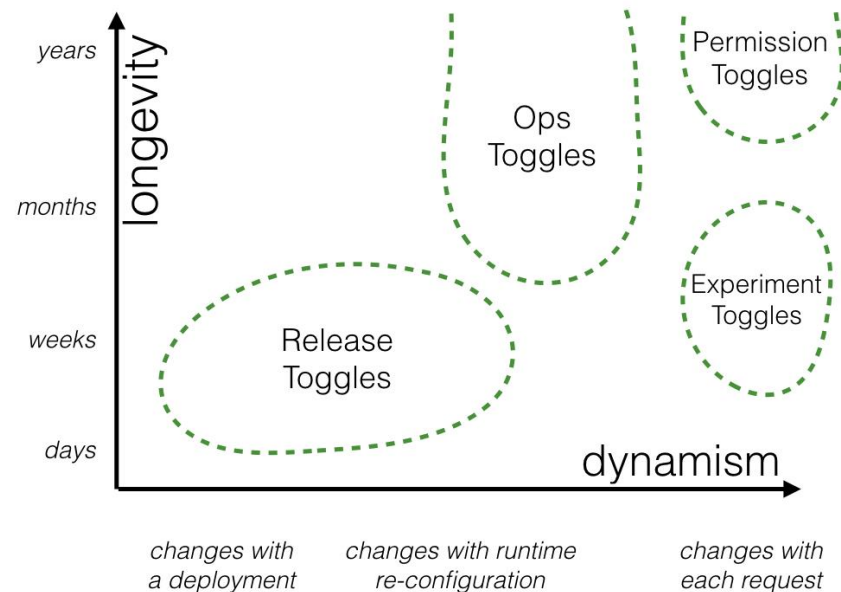
Ring 0

Held back until ring 0 looks healthy

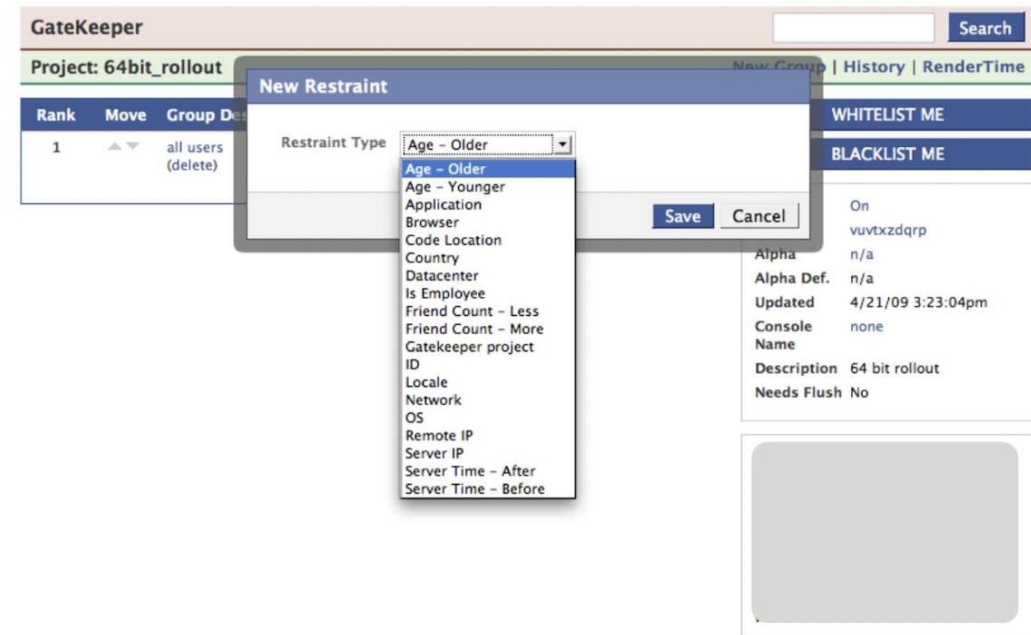
- A crash on boot is about as loud as failures get; a small first ring would have shown it within minutes
- The fix is cheap to describe and hard to build: rings, a health signal, and an automatic stop
- **Question:** what metric would tell you the canary is sick?

Feature flags

Ship code turned off, then turn it on per user, group, or percentage of traffic without another deployment

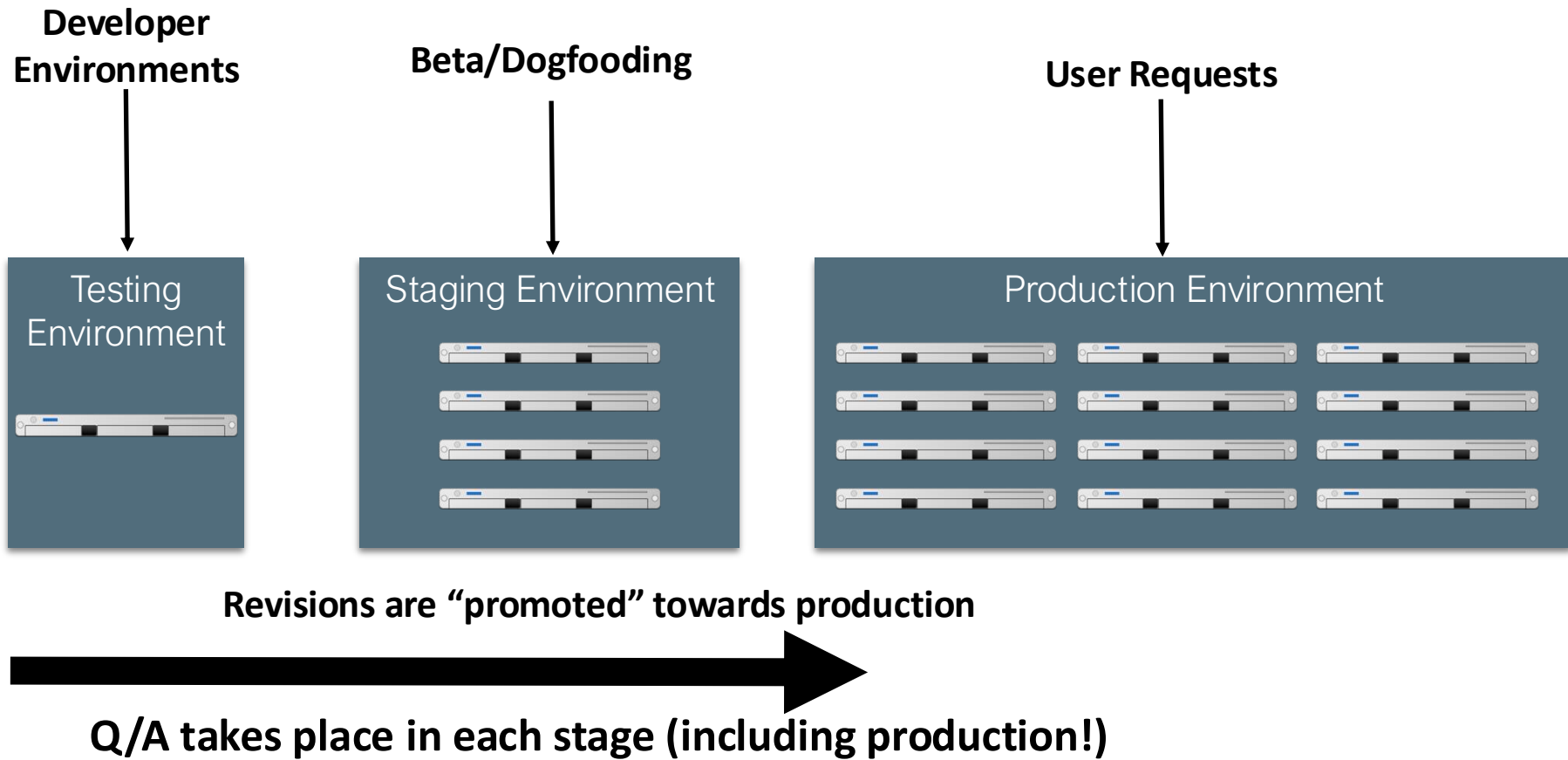


Pete Hodgson, "Feature Toggles", martinowler.com



Facebook's Gatekeeper

Continuous Delivery Leverages Relies on Staging Environments



Ring deployment: Microsoft



Commits flow outward ring by ring; if a ring shows a problem, the build is pulled back (“de-flighted”)

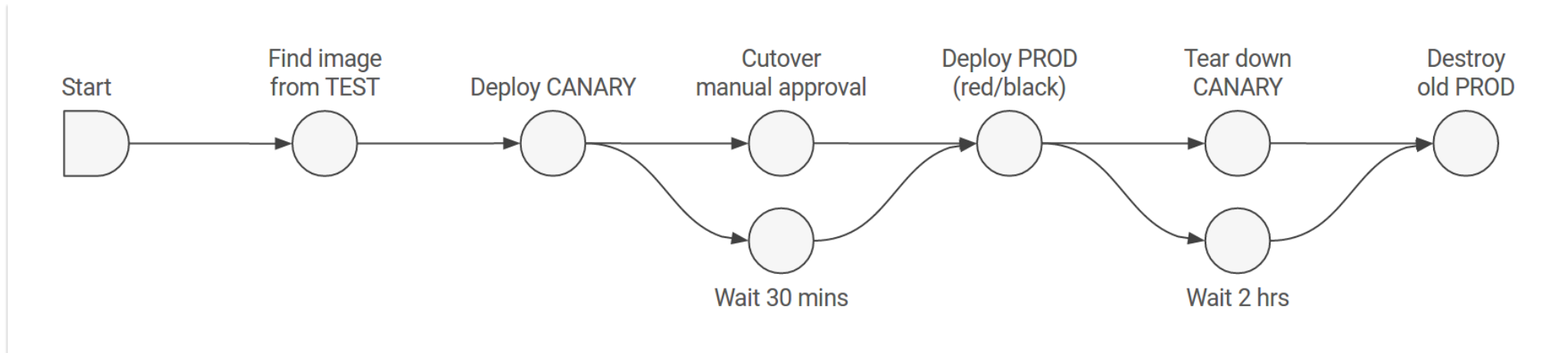
- Windows Insider program:
 - Dev Channel: newest builds, weekly
 - Beta Channel: builds Microsoft has validated more
 - Release Preview Channel: highest quality, just before general release

Release channels for software users install

- When users install the software (a browser, an OS), you can't flip a load balancer, so releases move through channels
- Firefox: Nightly → Beta → Release, with a new release about every 4 weeks
 - Extended Support Release (ESR) for organizations that need slower change
- Edge (and Chrome): Canary (daily) → Dev (weekly) → Beta → Stable
- A release engineer can fast-track an urgent fix through the channels

Continuous Delivery Tools

- Simplest tools deploy from a branch to a service (e.g. Render.com, Heroku)
- More complex tools:
 - Auto-deploys from version control to a staging environment + promotes through release pipeline
 - Monitors key performance indicators to automatically take corrective actions
 - Example: “[Spinnaker](https://spinnaker.io/)” (Open-Sourced by Netflix, c 2015)



What can it look like when it's done?

- Netflix Spinnaker (open-source, fully automated CD pipeline):
 - Takes code from the repository to production
 - Developers specify the required tests
 - Decides where and how code runs (replication, placement)
 - Supports canary deployments and traffic management

5×

lower change failure rate

440×

faster from commit to
deploy

46×

more frequent
deployments

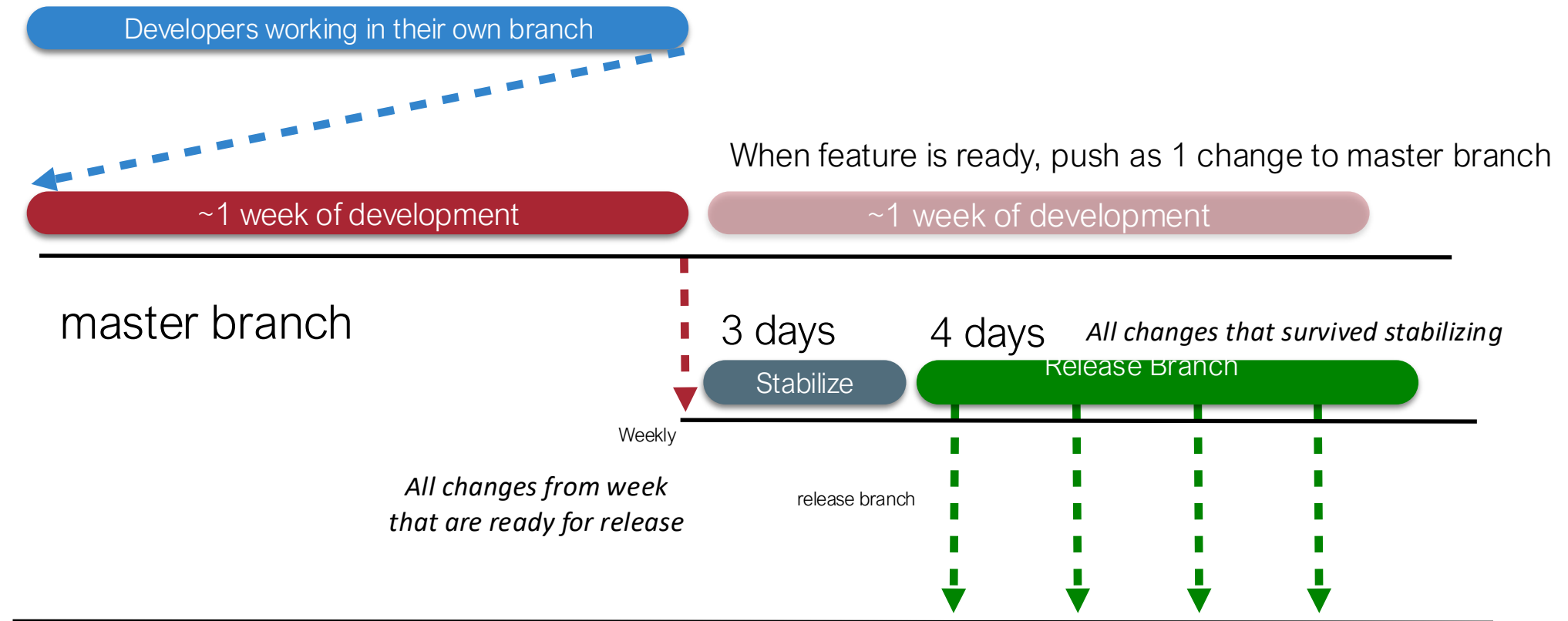
44%

more time on new
features and code

High performers vs. low performers, Puppet State of DevOps Report 2017. Spinnaker: [youtube.com/watch?v=UTKIT6STSVM](https://www.youtube.com/watch?v=UTKIT6STSVM)

Deployment Example: Facebook.com

- Pre-2016



Deployment Example



“Our main goal was to make sure that the new system made people’s experience better — or at least, didn’t make it worse. After a year of planning and development, over the course of three days **we enabled 100% of our production web servers to run code deployed directly from master**”

- Chuck Rossi, Director Software Infrastructure & Release Engineering @ Facebook

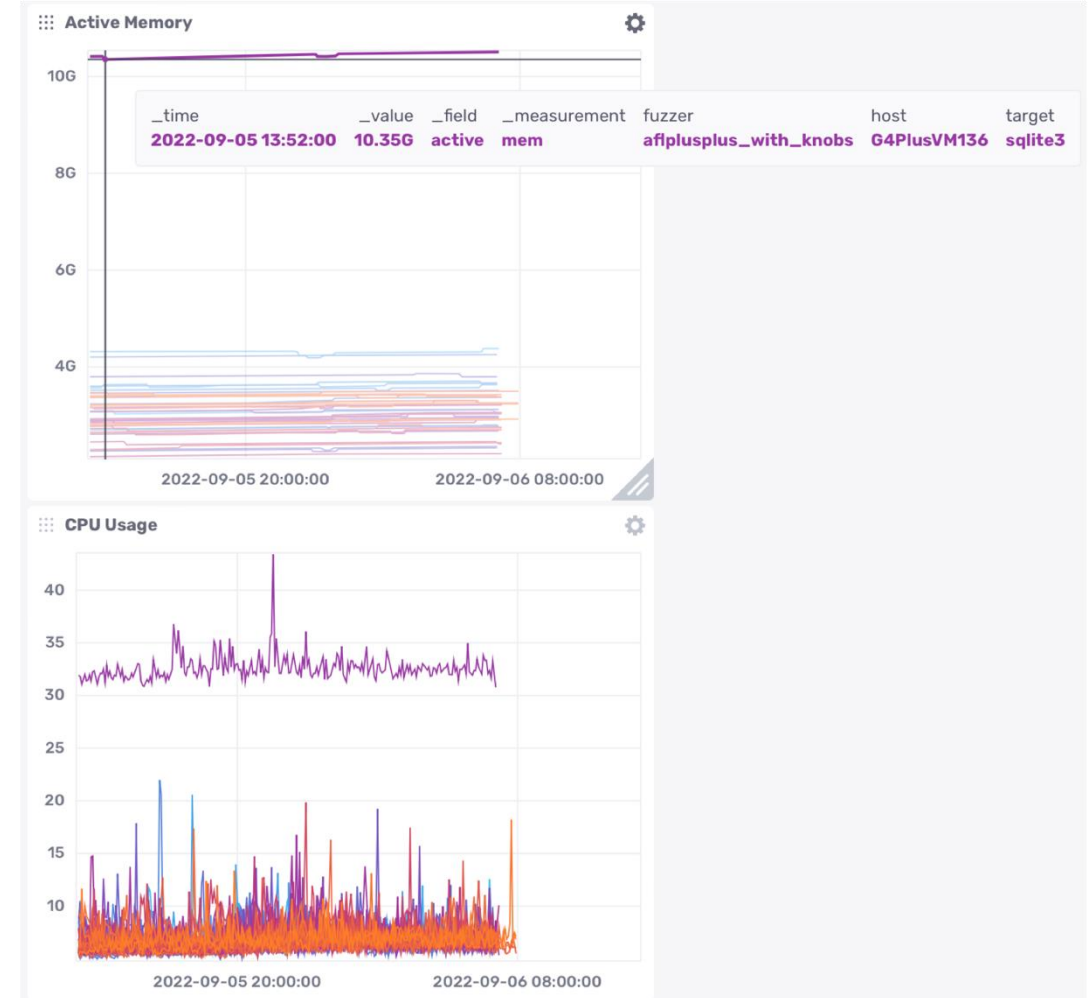
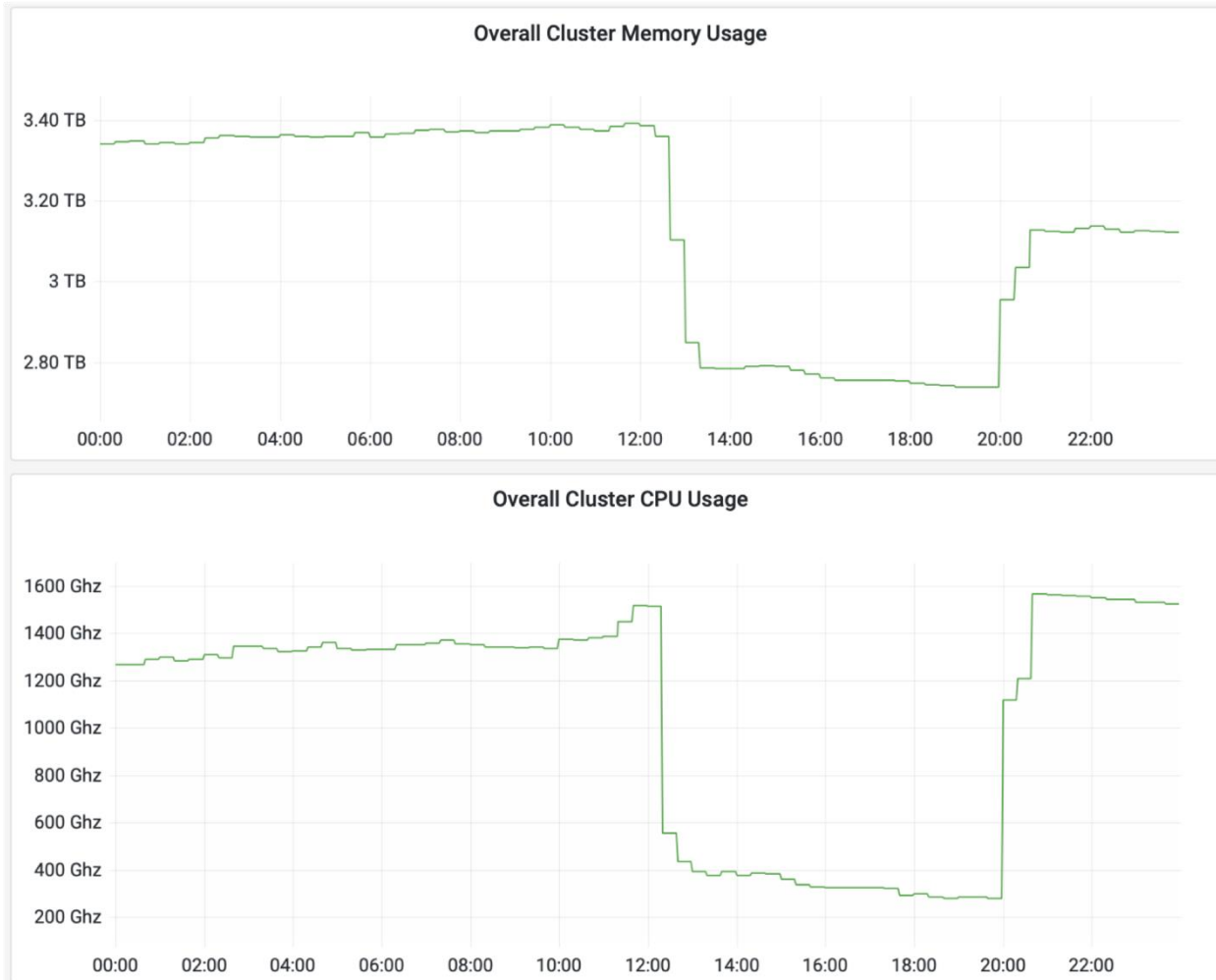
Challenge 5: Is the release actually healthy?

Monitoring and observability

Continuous Delivery Relies on Monitoring

- Consider both direct (e.g. business) metrics, and indirect (e.g. system) metrics
- Hardware
 - Voltages, temperatures, fan speeds, component health
- OS
 - Memory usage, swap usage, disk space, CPU load
- Middleware
 - Memory, thread/db connection pools, connections, response time
- Applications
 - Business transactions, conversion rate, status of 3rd party components

Monitoring can help identify operational issues



Continuous Delivery Tools Take Automated Actions

- Example: Automated roll-back of updates at Netflix based on SPS

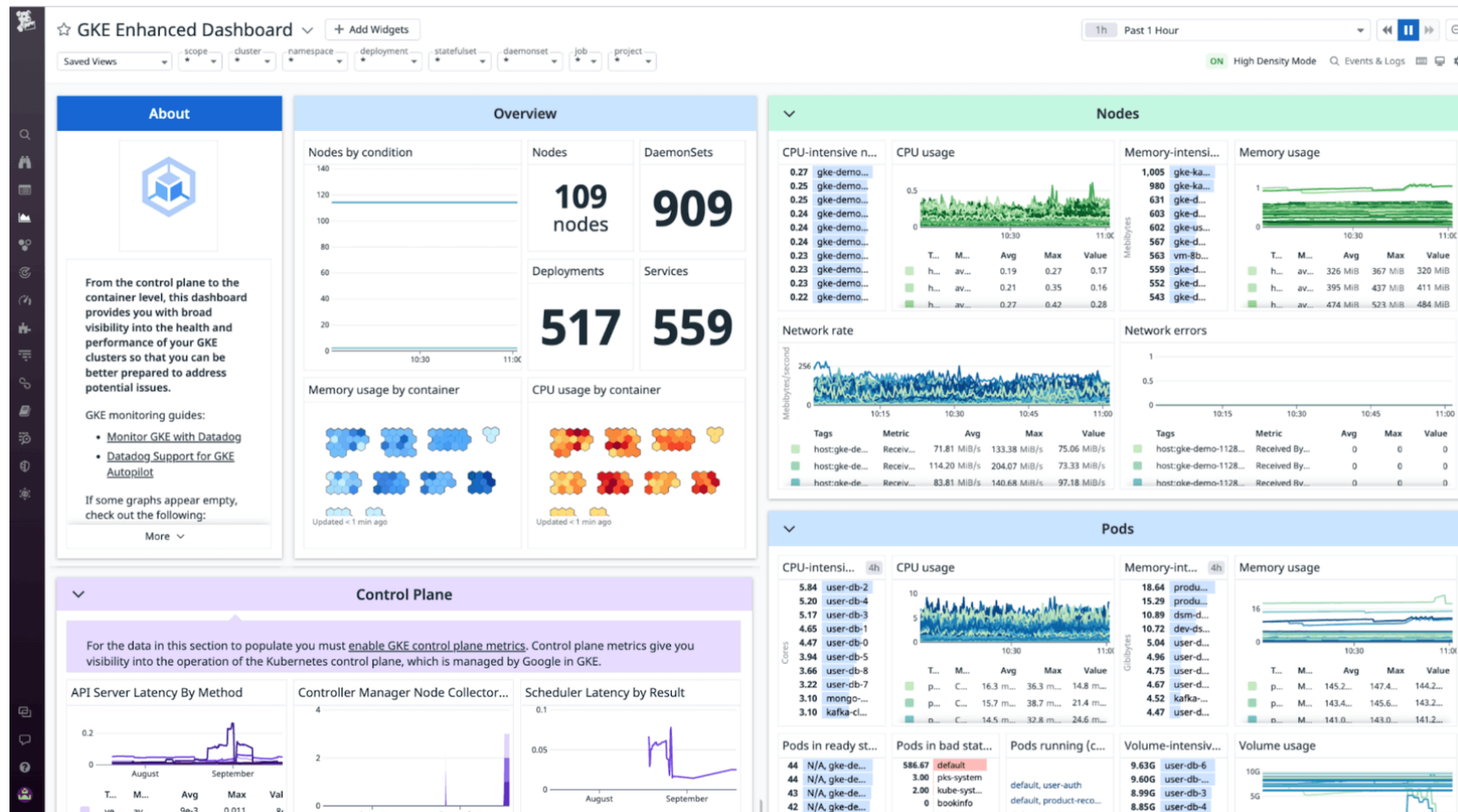


From Monitoring to Observability

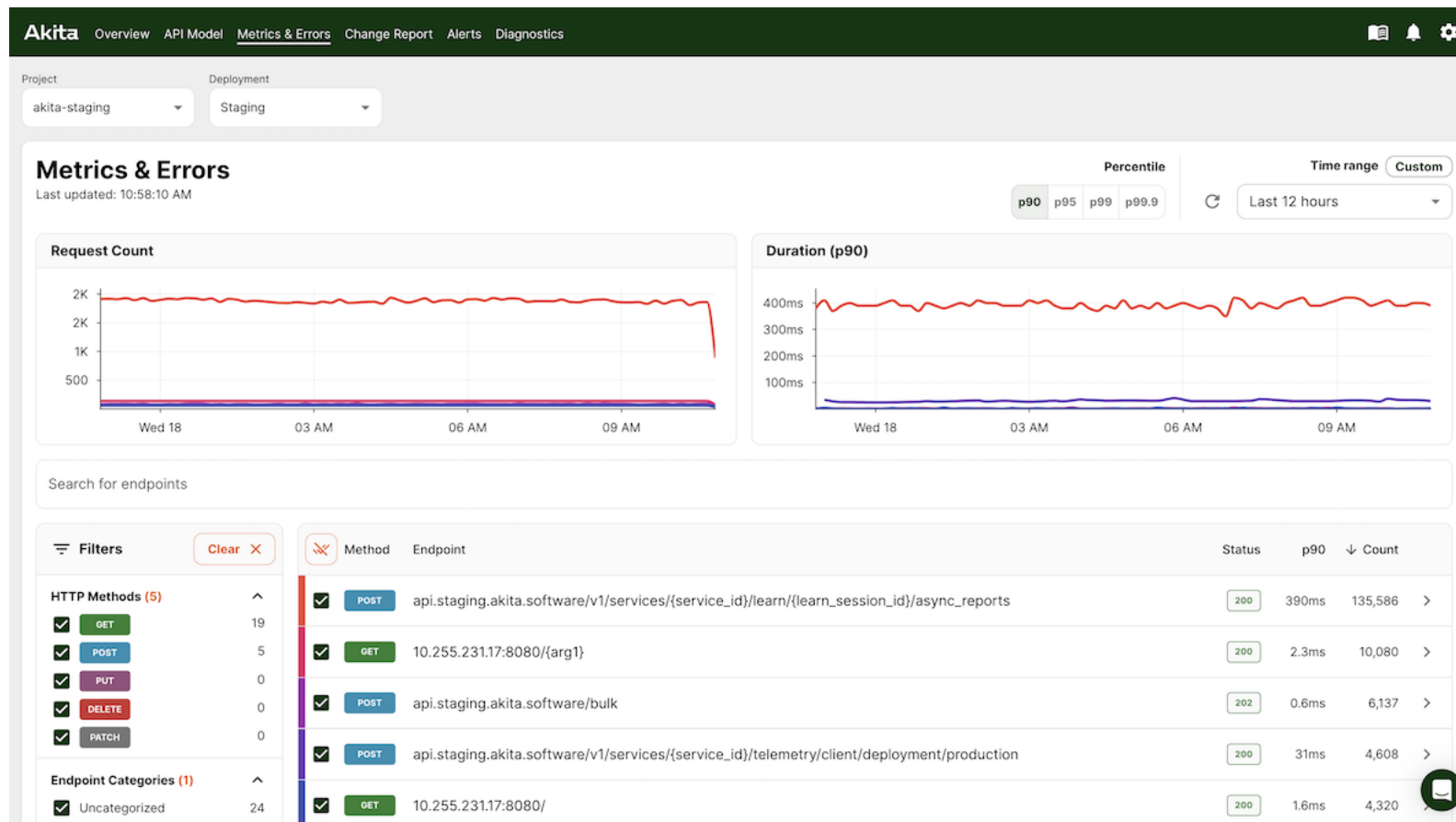
- Understanding what is going on inside of our deployed systems

Example dashboard by DataDog:

<https://www.datadoghq.com/blog/gke-dashboards-integration-improvements/>



New Tools allow Observability inside of Apps, Too



Remember:

- McNamara Fallacy
 - Measure whatever can be easily measured
 - Disregard that which cannot be measured easily
 - Presume that which cannot be measured easily is not important
 - Presume that which cannot be measured easily does not exist

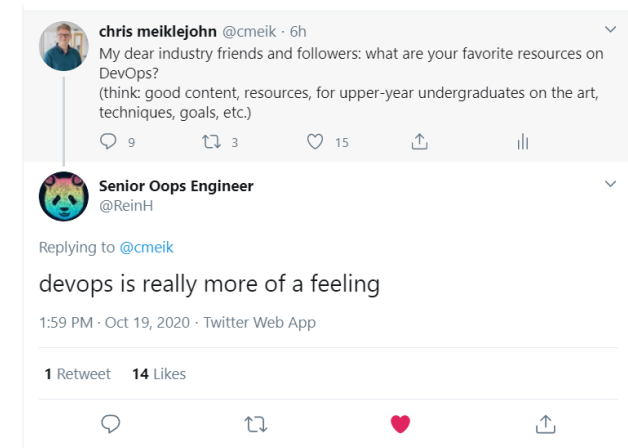
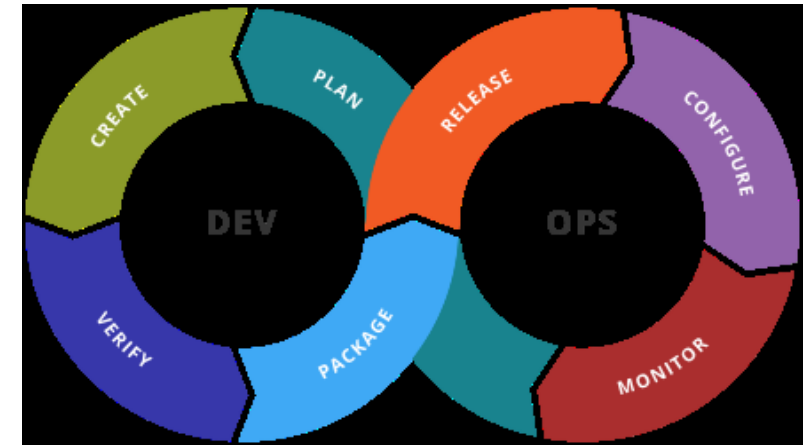


Putting it together: DevOps

Who is responsible for all of this?

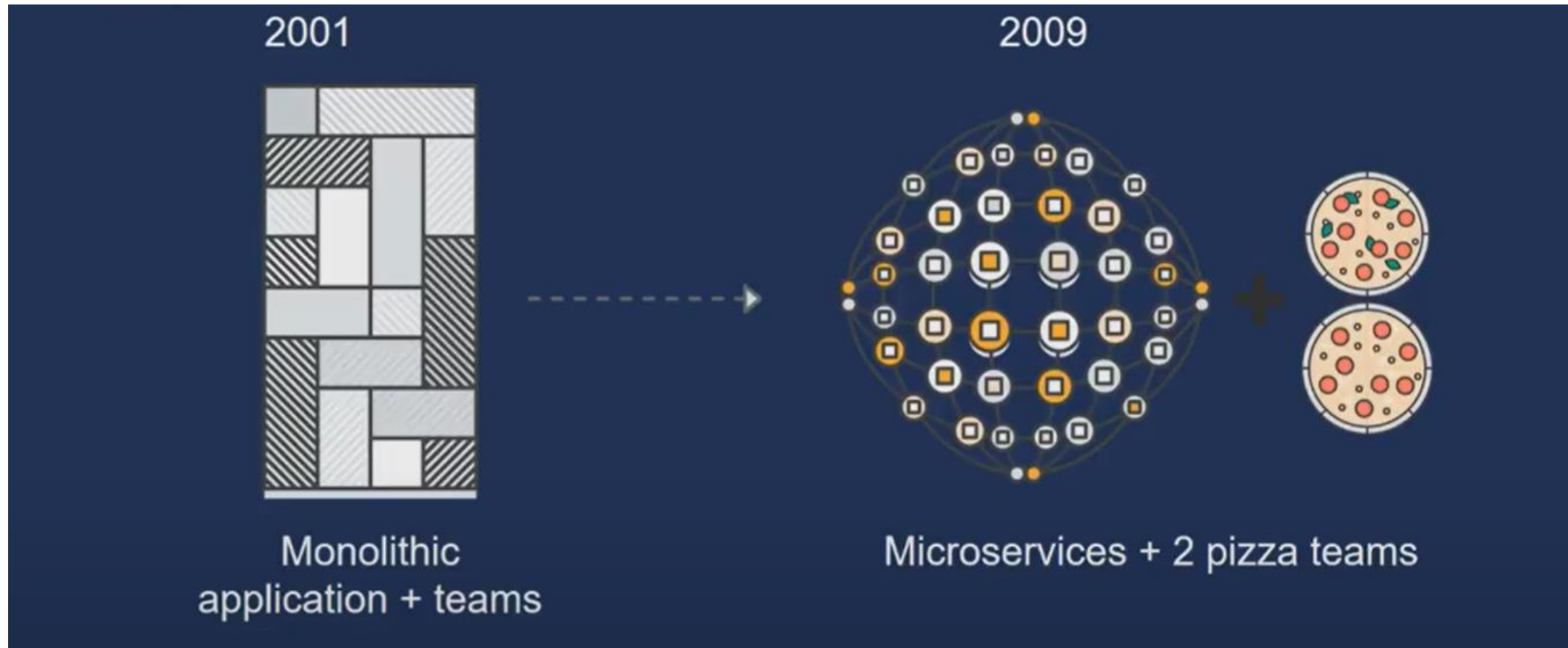
What is DevOps?

- Bringing together two groups that used to be separate:
 - **Development**, measured on features completed and code shipped
 - **Operations**, measured on stability, reliability, availability
- Benefits:
 - **Velocity**: how quickly changes reach users
 - **Quality**: successful delivery of features



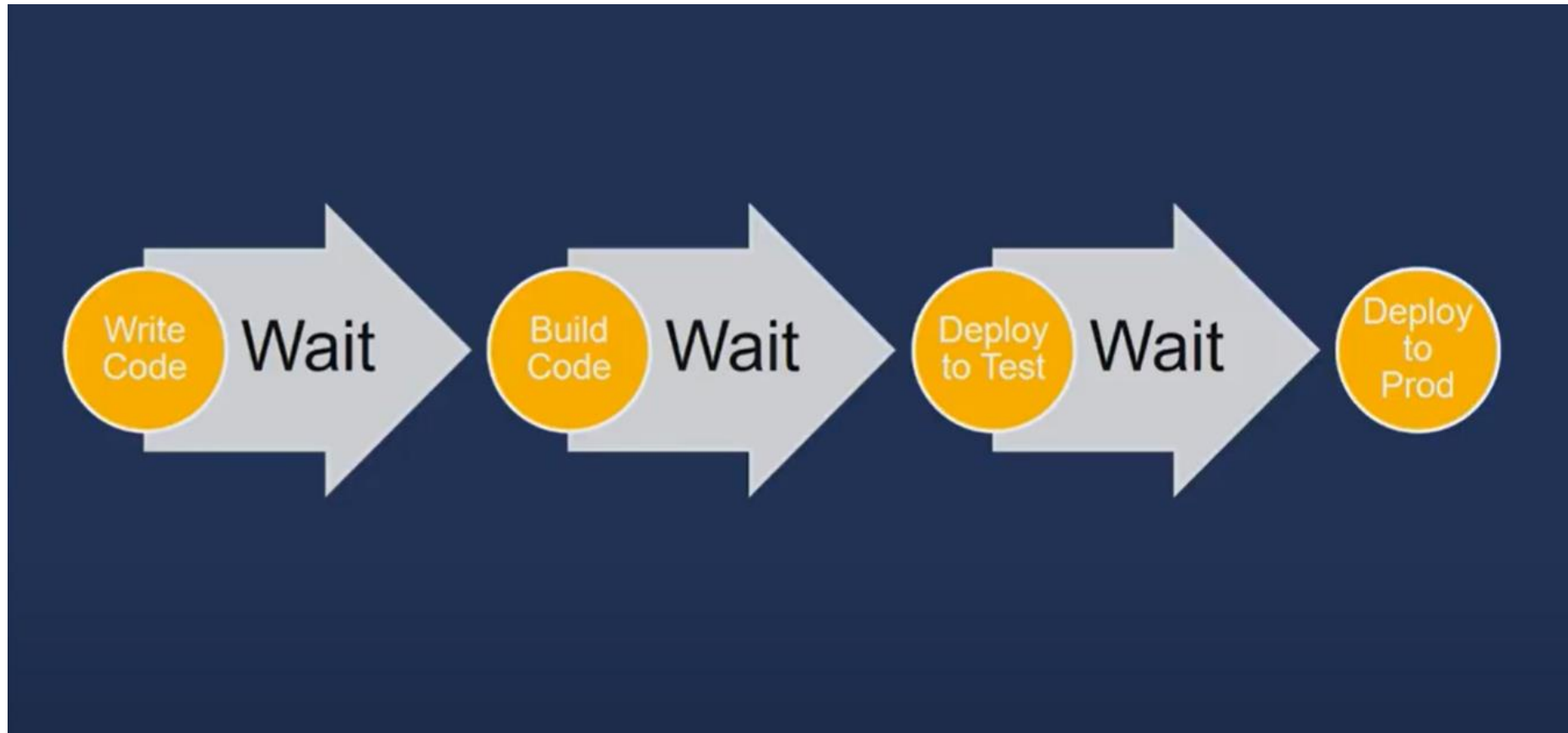
youtube.com/watch?v=UbtB4sMaaNM

Amazon 2001–2009: monolith to microservices



youtube.com/watch?v=mBU3Aj3j1rg

Amazon 2001: “We were just waiting”



youtube.com/watch?v=mBU3AJ3j1rg

How do we get to DevOps?

Technological

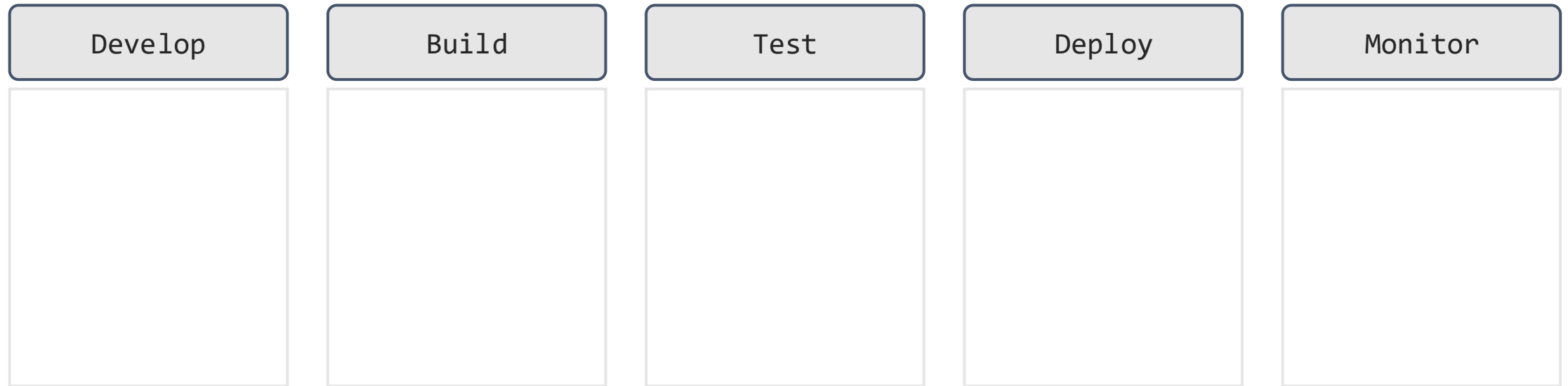
An automated process for moving code from development to release: check-in, build, unit test, build artifact, integration test, load test, staging, production, and finally monitoring and telemetry.

Cultural

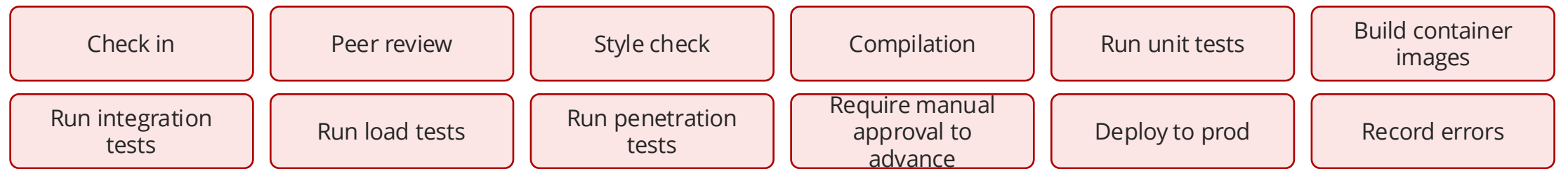
Cohesive, multidisciplinary teams. Developers are the “first responders” when things go wrong in production, with a sense of ownership from inception to release.

youtube.com/watch?v=UbtB4sMaaNM

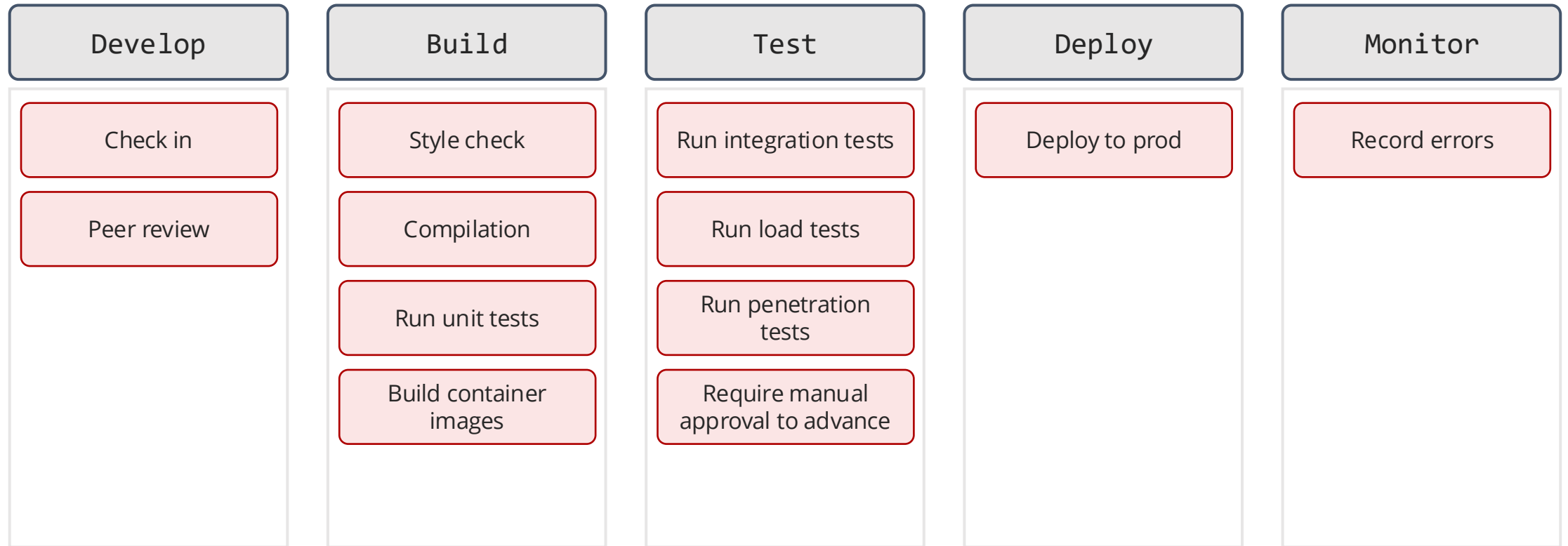
Exercise: assemble a DevOps pipeline



Put each step into the stage where it belongs. Some could go in more than one place; be ready to defend your choice.



A typical DevOps pipeline



One reasonable answer, not the only one.

Compare Continuous Delivery and TDD

- Test driven development
 - Write and maintain tests per-feature
 - Unit tests help locate bugs (at unit level)
 - Integration/system tests also needed to locate interaction-related faults
- Continuous delivery
 - Write and maintain high-level observability metrics
 - Deploy features one-at-a-time, look for canaries in metrics
 - Write fewer integration/system tests