CI and Deployment

17-313 Spring 2024

Foundations of Software Engineering

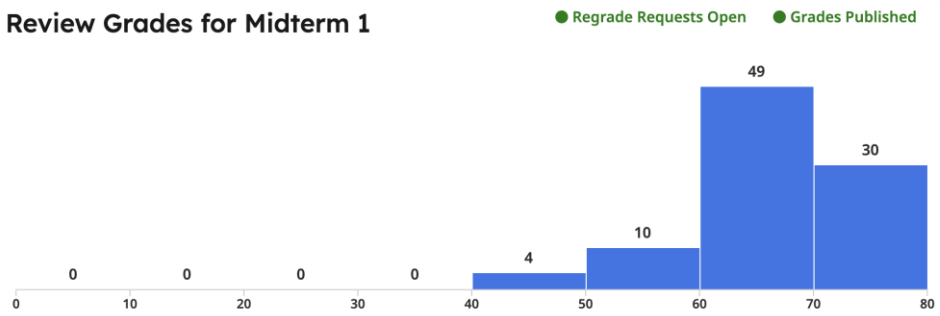
https://cmu-313.github.io

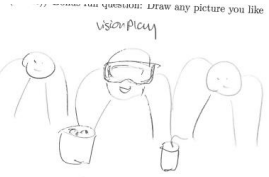
Michael Hilton and Eduardo Feo Flushing

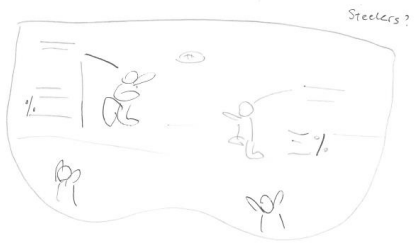
Thanks to Jon Bell for slide inspiration:
https://neu-se.github.io/CS4530-Spring-2024/

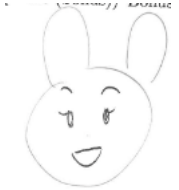
Administrivia

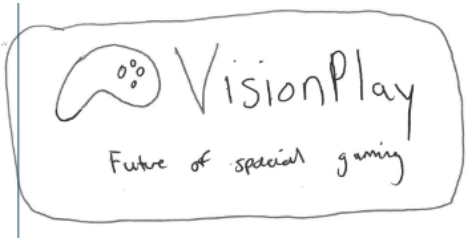
- Midterm re-grade requests open
- Thursday will be an activity, bring your laptop. If you have not done recitation, you should do that before Thursday





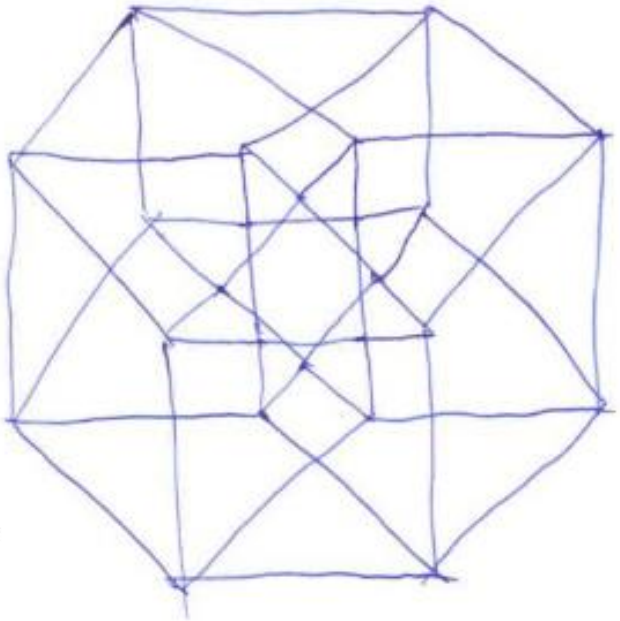


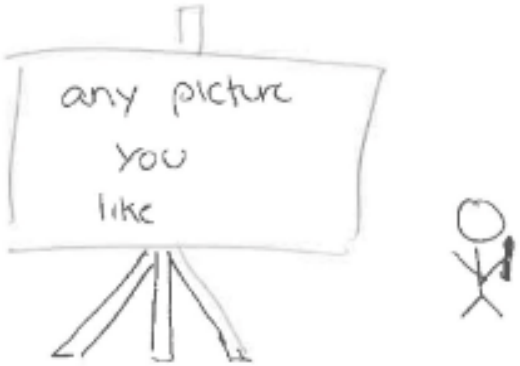




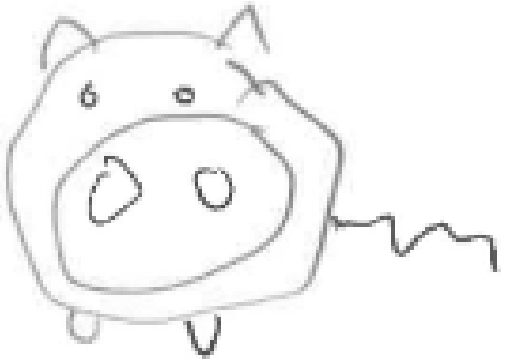
Gojo
Satoru
AKA "The GOAT"
AKA "The Honored One"
AKA "The Strongest"


beep
boop

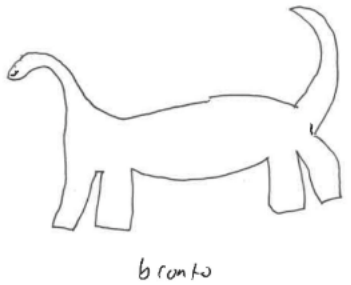



any picture
you
like


Draw any picture you like
mooo!

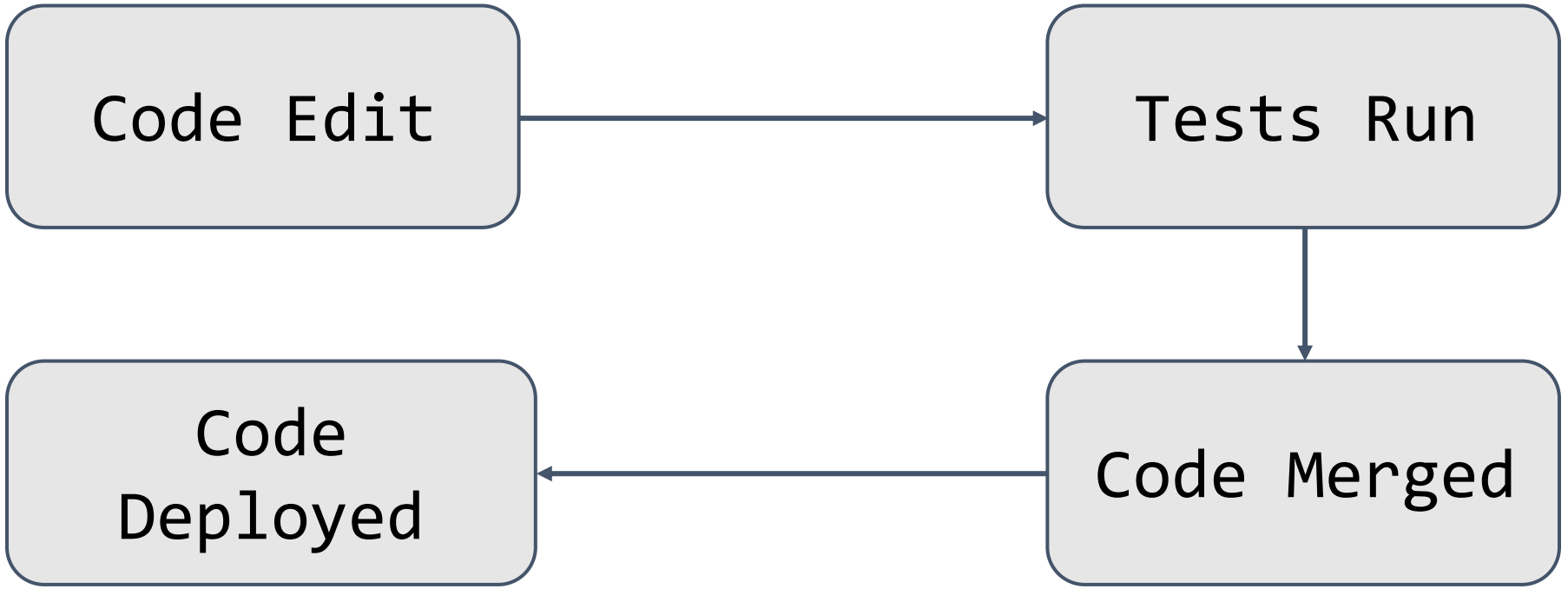



professor


bronto

Review: 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

(2005) Hudson/Jenkins

(2011) Travis CI

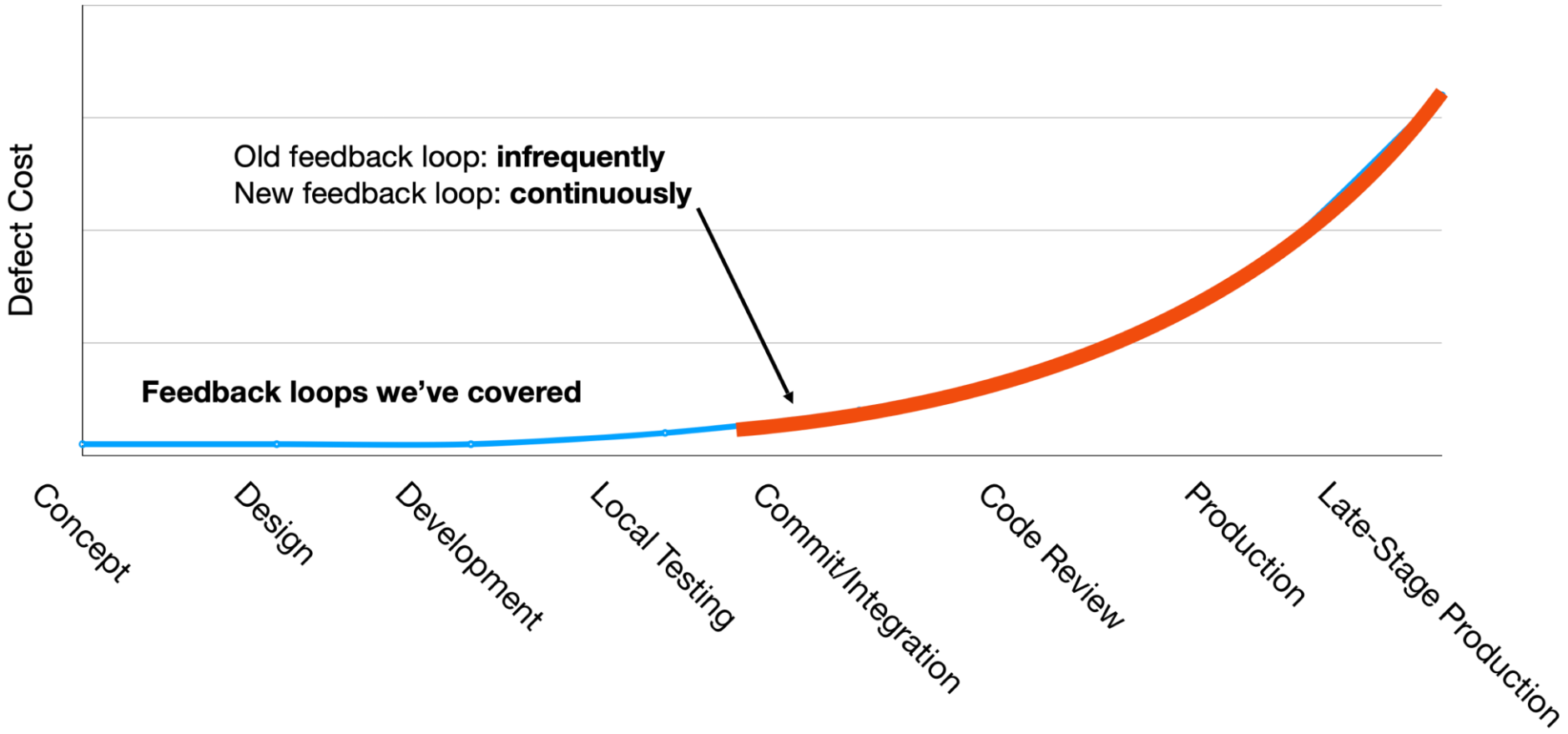
(2019) GitHub Actions

Observation

CI helps us catch errors before others see them

Agile values fast quality feedback loops

- Faster feedback = lower cost to fix bugs



Example: Some bugs slip through testing, even in highly-regulated industries

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." …

Photo: saiters_photography (IG, different plane/airpot)

… "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.

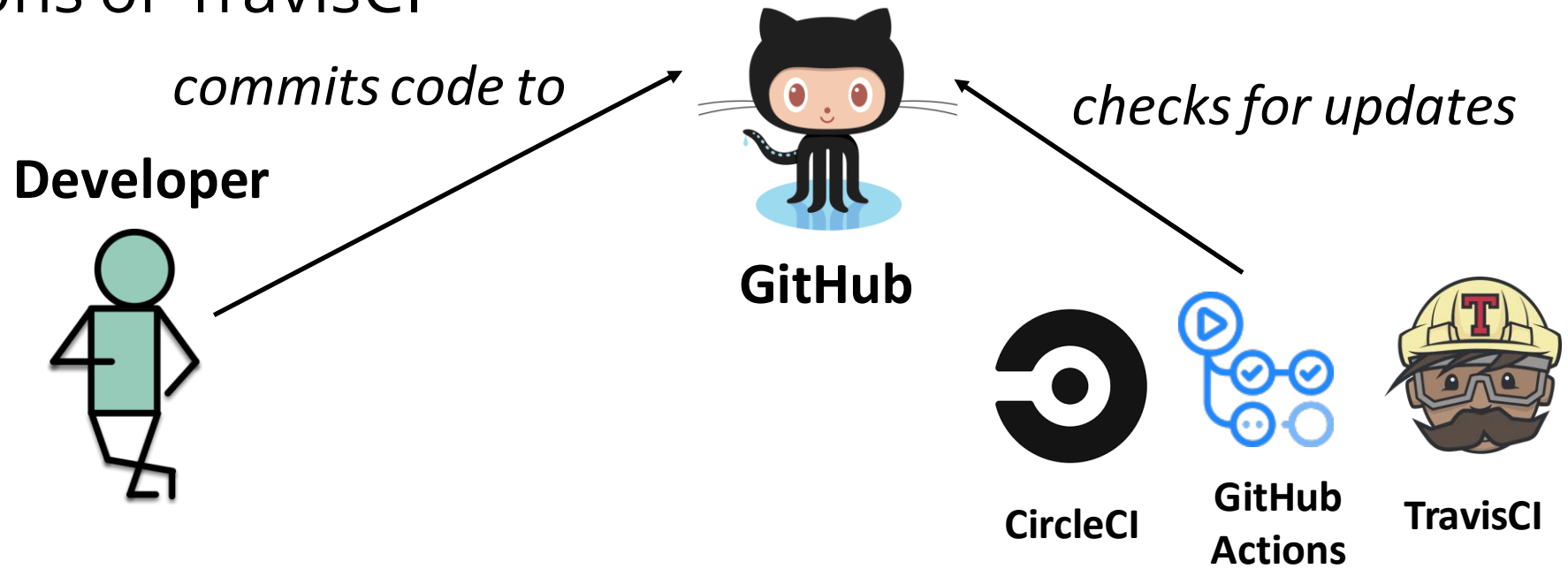
Peyton added that even though the update to the DynamicSource software had been tested over an extended period, the bug was missed because it only presented when many aircraft at the same time were using the system.

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/

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

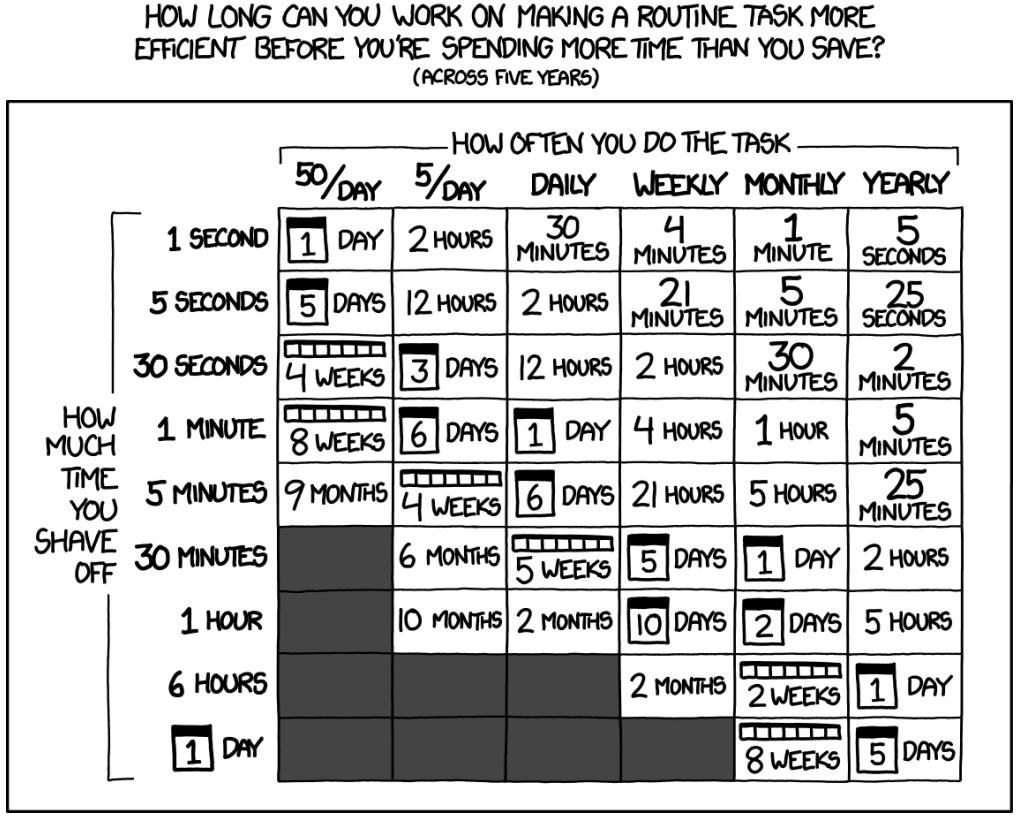
Example: Small scale CI, with a service like CircleCI, GitHub Actions or TravisCI



Runs build for each commit

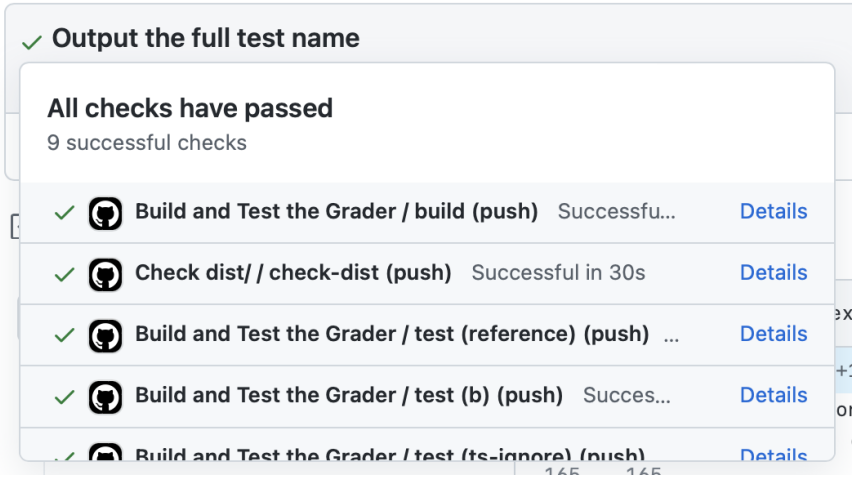
Automating Feedback Loops is Powerful

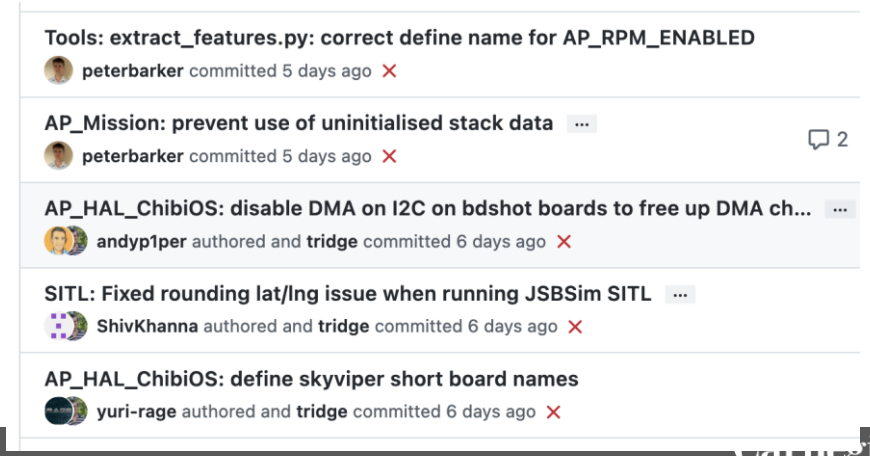
Consider tasks that are done by *dozens* of developers (e.g. testing/deployment)



Attributes of effective CI processes

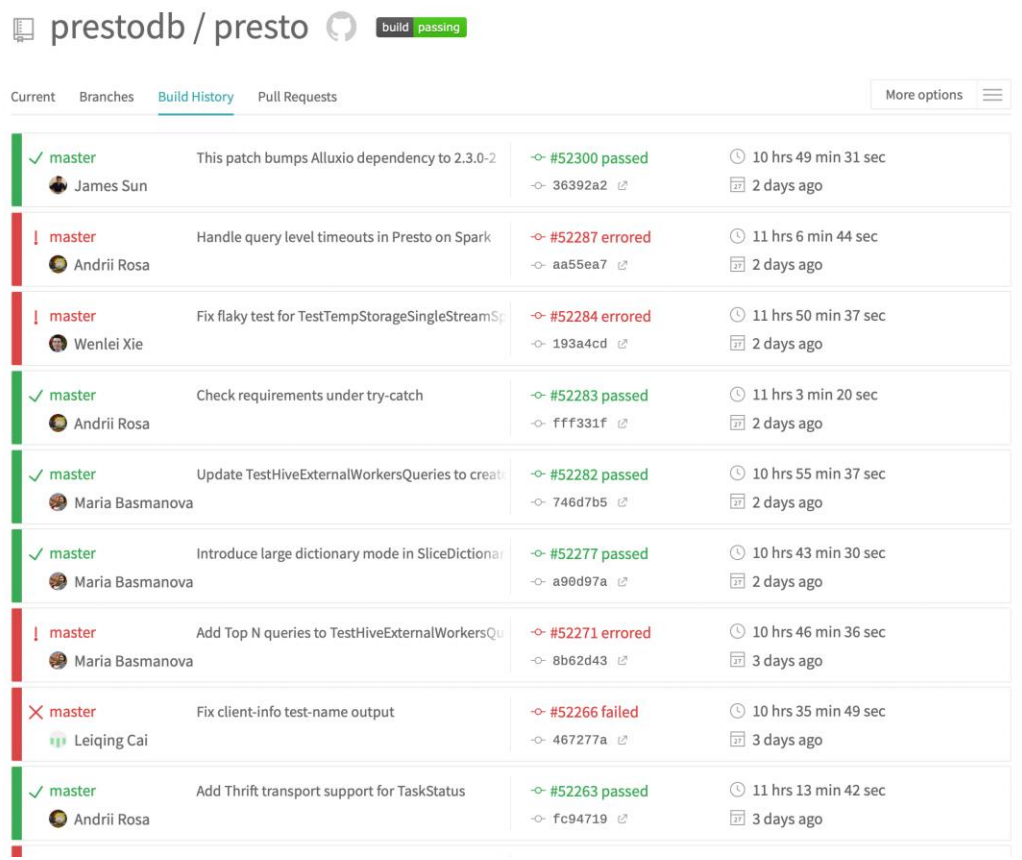
- Policies:
  - Do not allow builds to remain broken for a long time
  - CI should run for every change
  - CI should not completely replace pre-commit testing
- Infrastructure:
  - CI should be fast, providing feedback within minutes or hours
  - CI should be repeatable (deterministic)





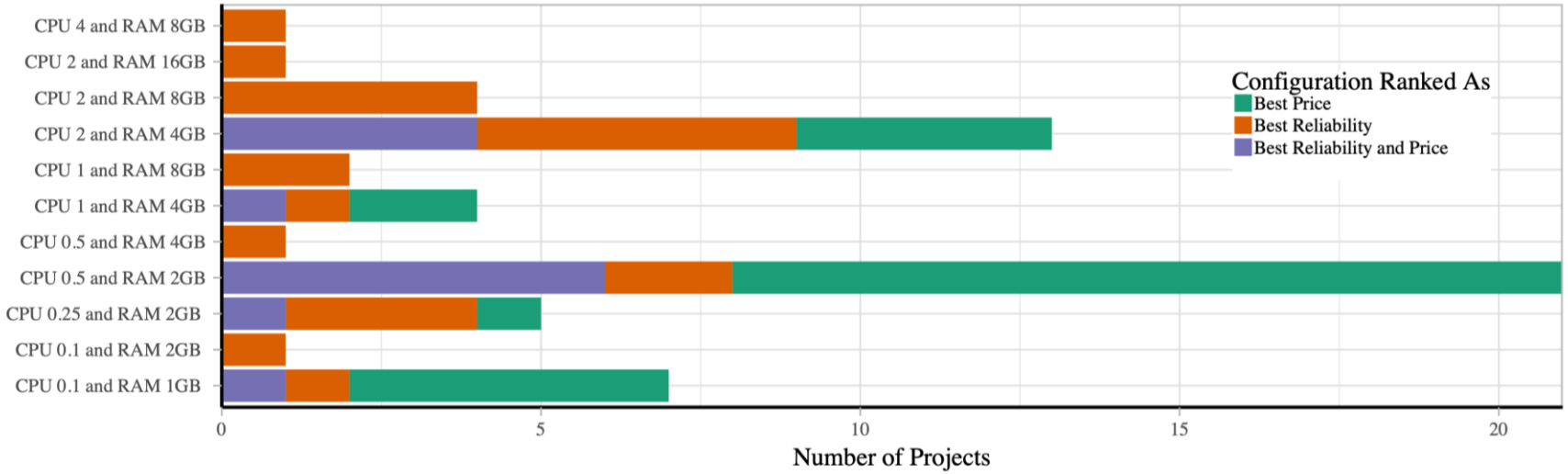
Effective CI processes are run often enough to reduce debugging effort

- 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



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

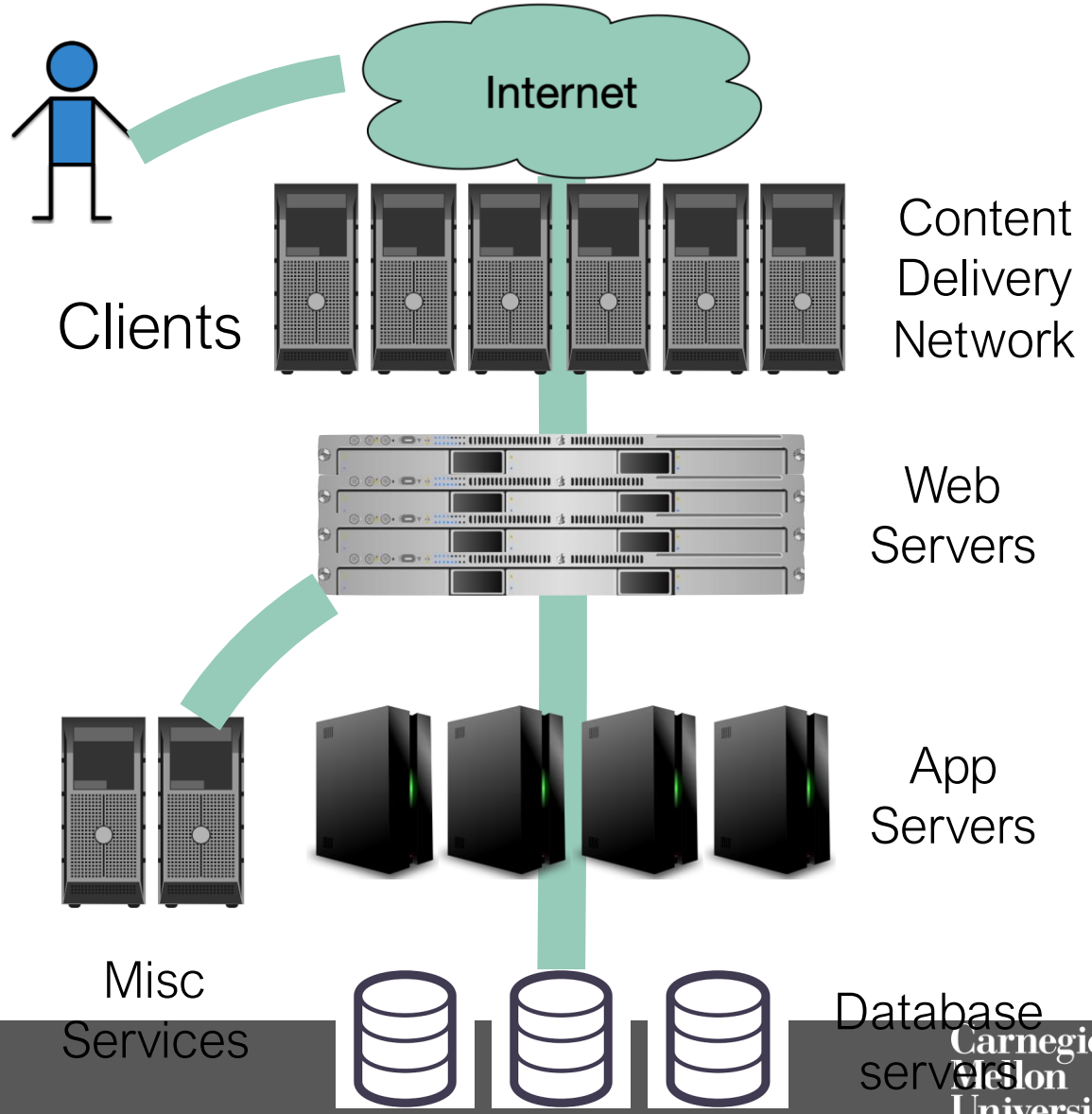
How can we continuously update our software in production?

Cloud Computing enables CD

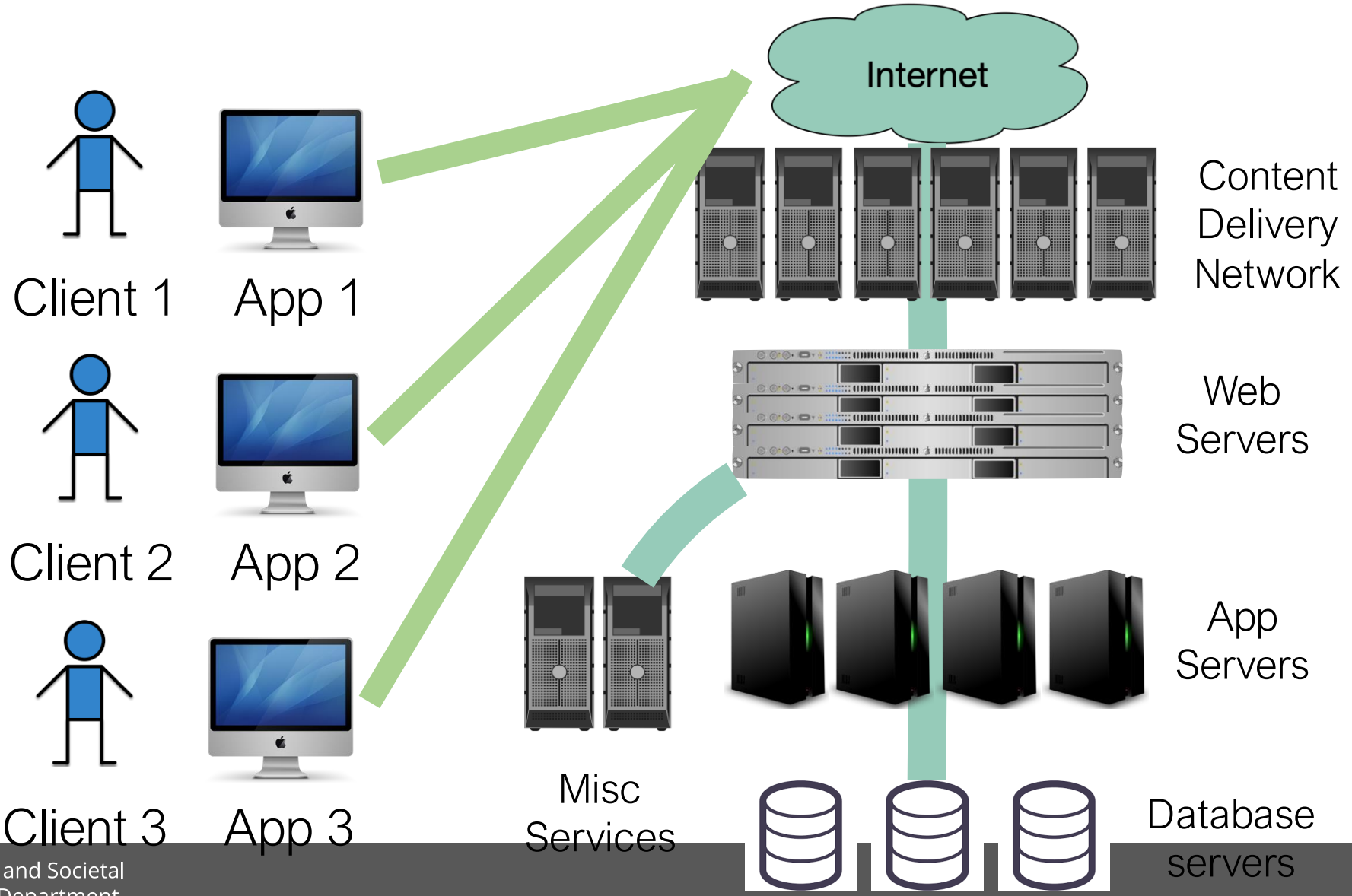
Cloud Computing/Deployment refresher

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

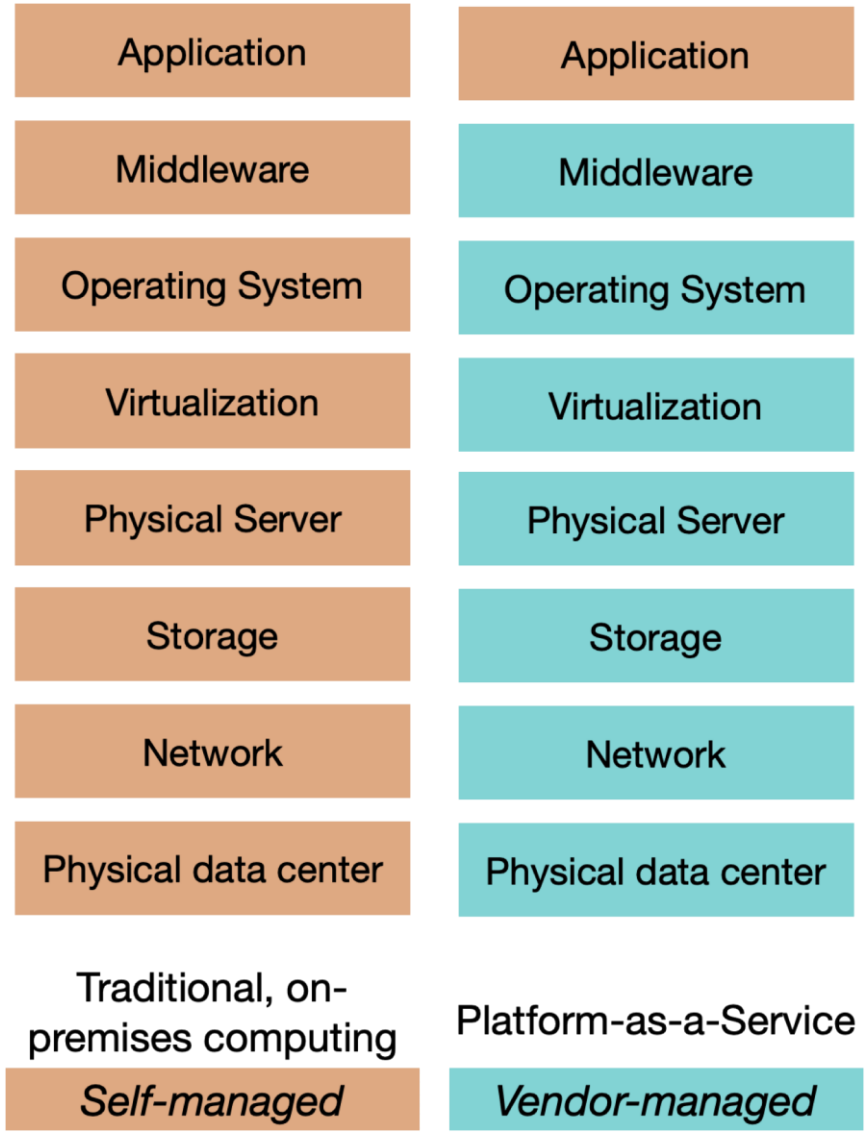


What parts of this infrastructure can be shared across different clients?



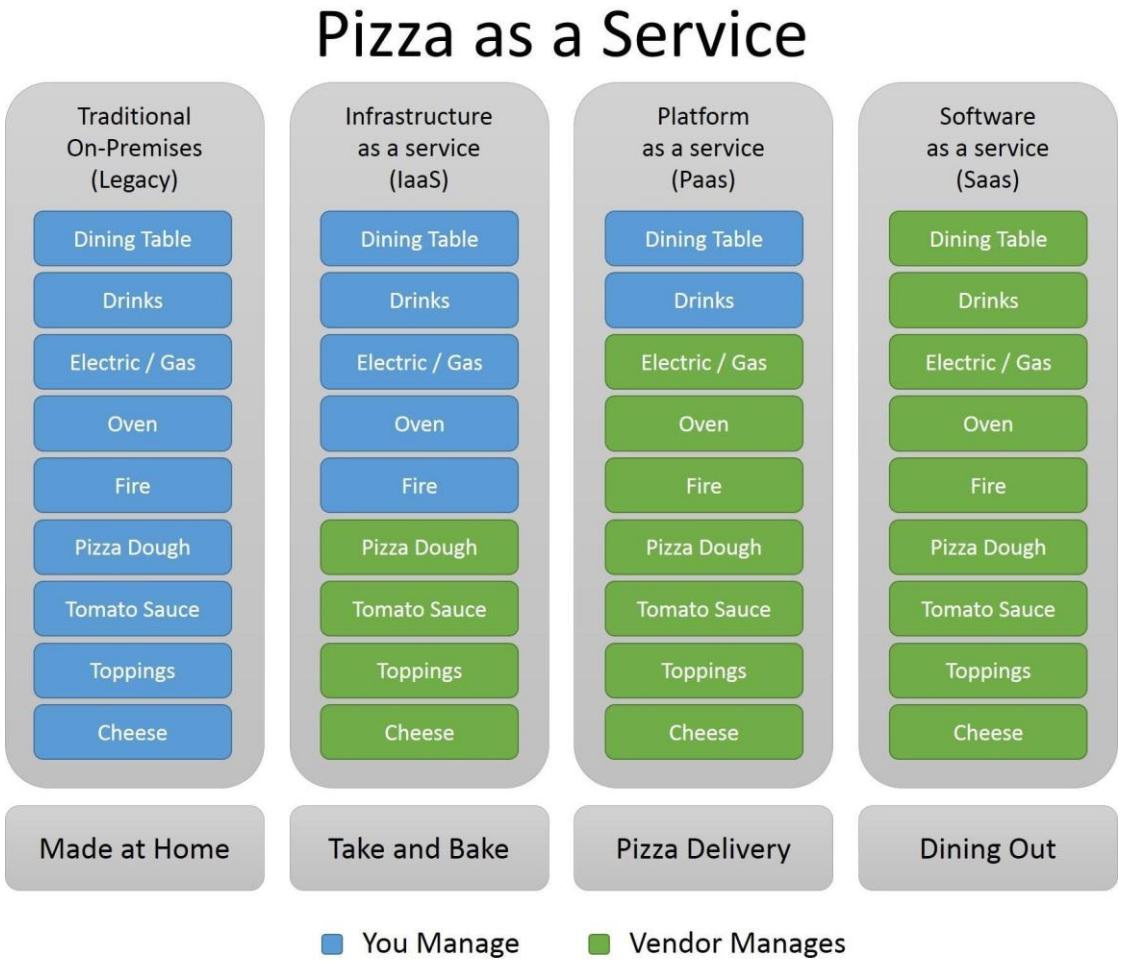
What is the infrastructure that needs to be shared?

- 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

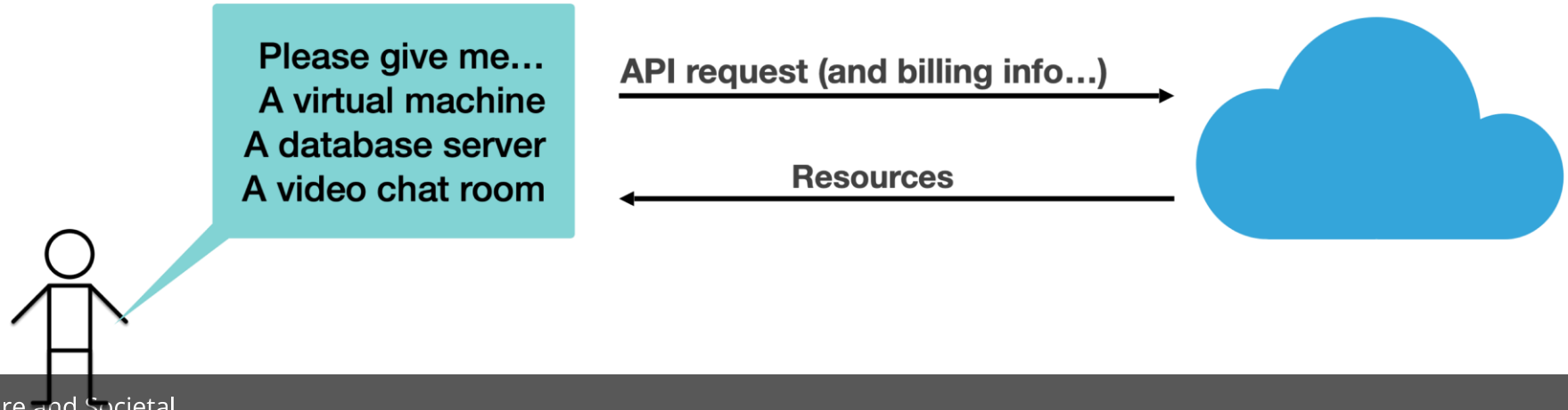
- At the physical level:
  - Multiple customers' physical machines in the same data center
  - Save on physical costs (centralize power, cooling, security, maintenance)
- At the physical server level:
  - Multiple customers' virtual machines in the same physical machine
  - Save on resource costs (utilize 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)

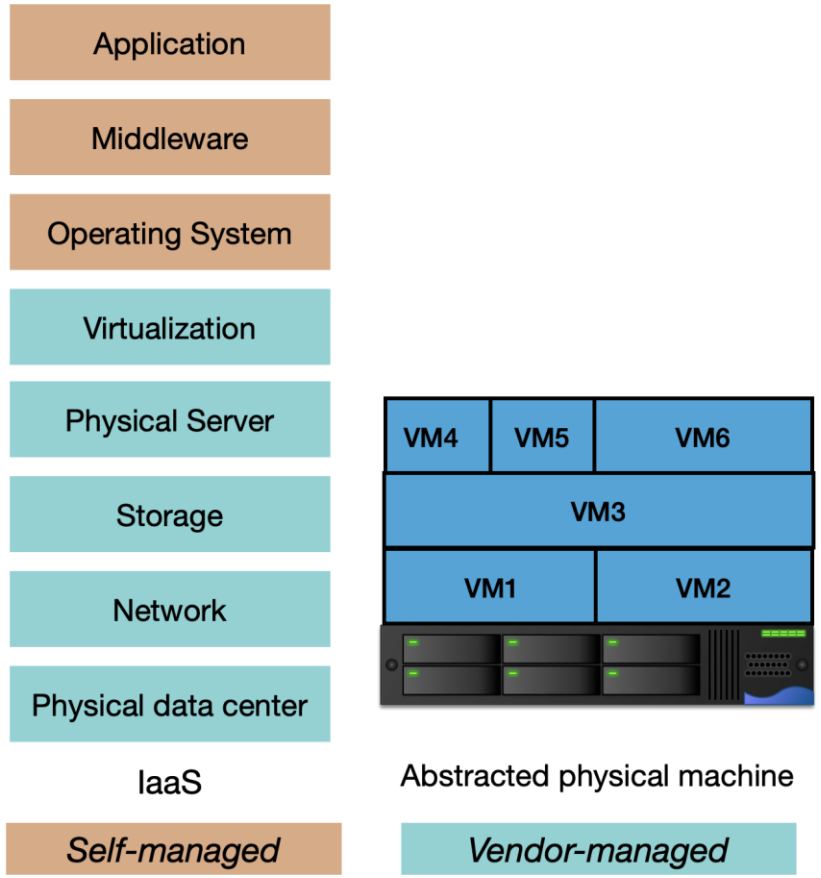
Cloud services gives on-demand access to infrastructure, “as a service”

- 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



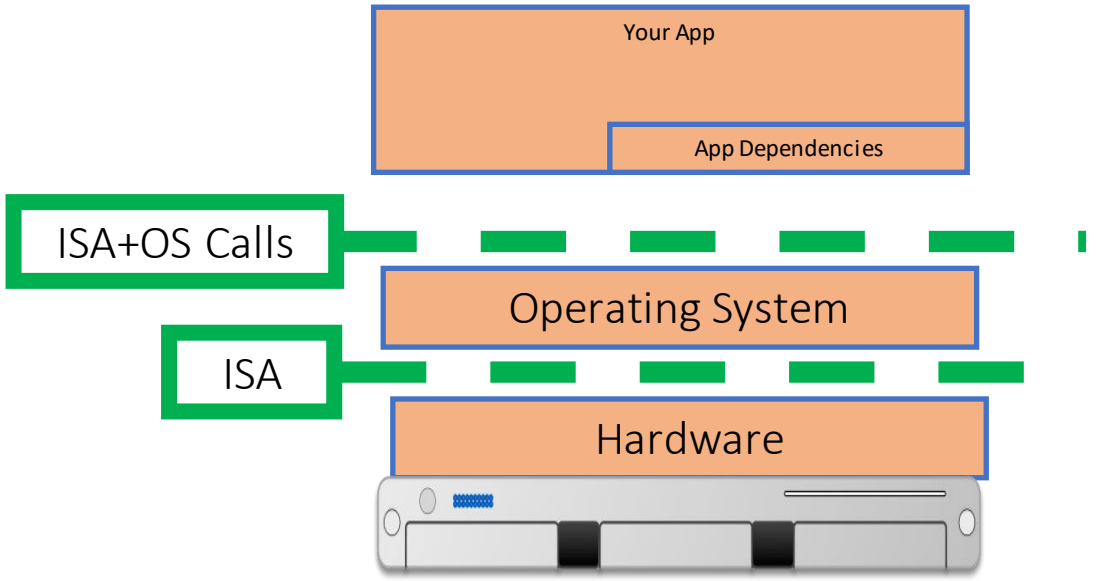
Infrastructure as a Service: Virtual Machines

- 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

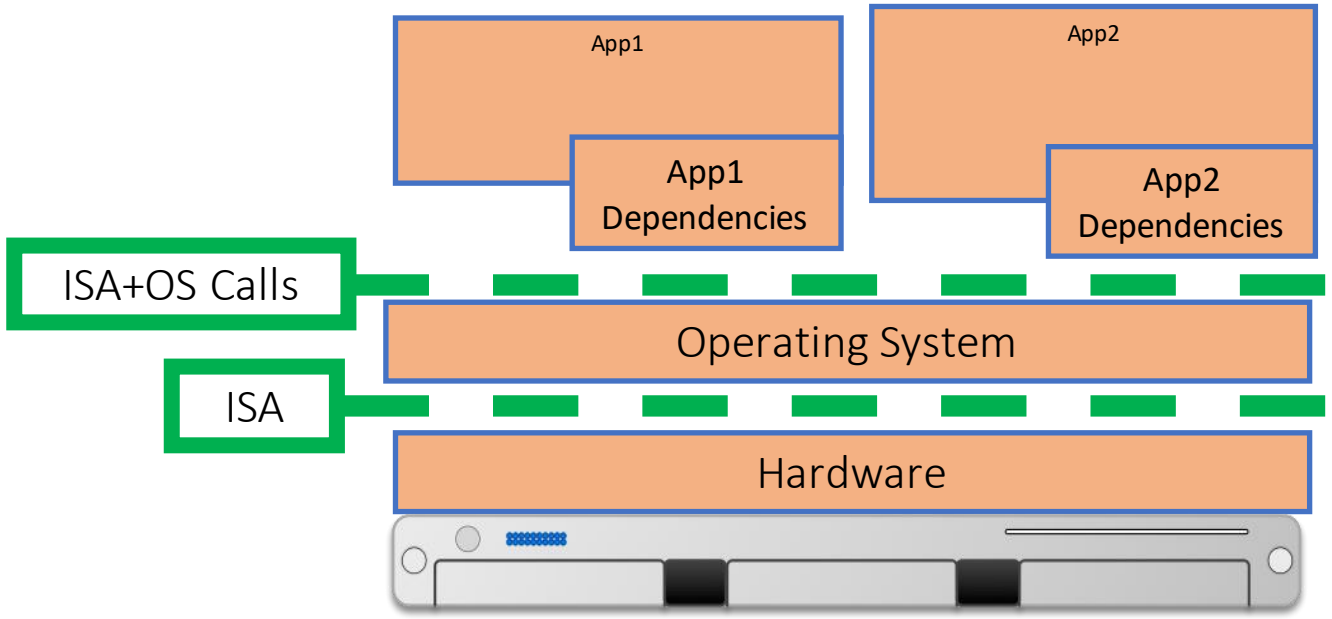


Let's look more closely at this software stack

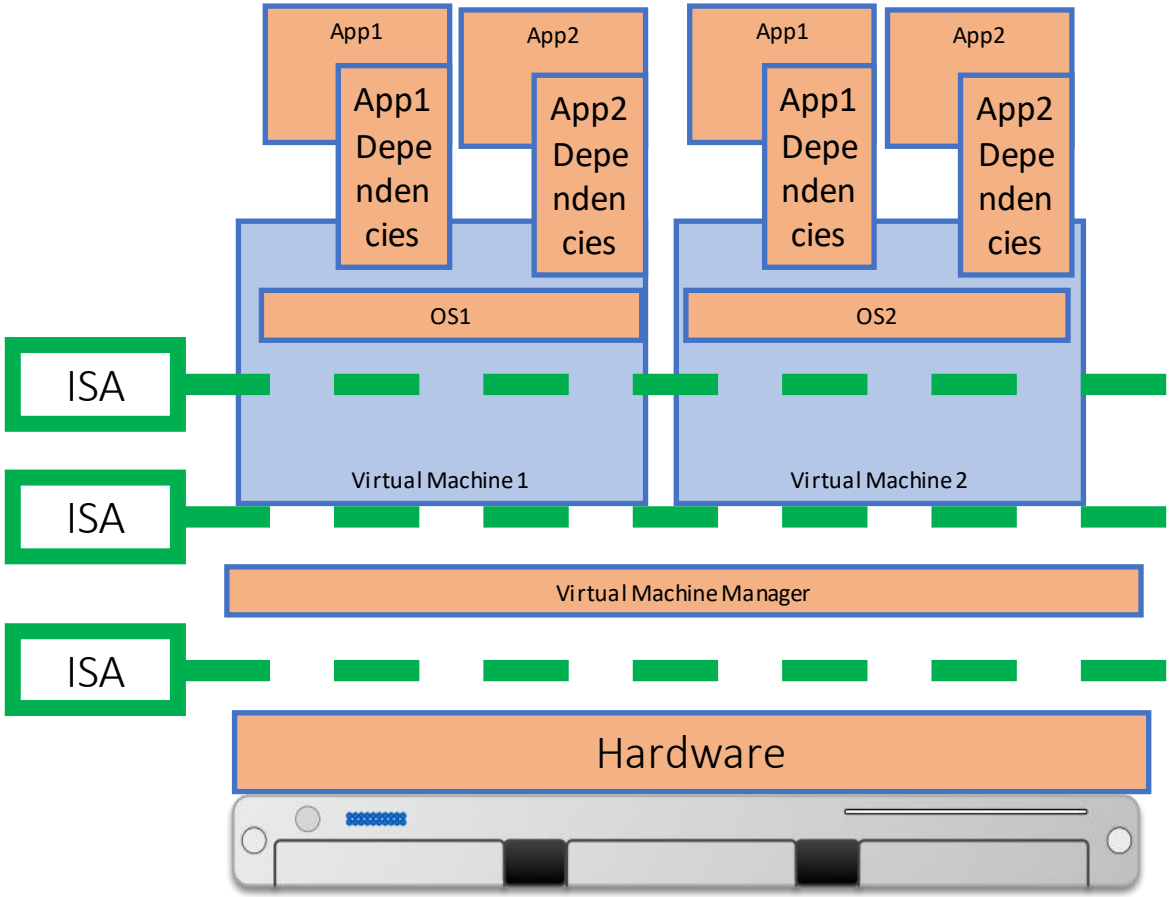
- The "instruction set" is an abstraction of the underlying hardware
- The operating system presents the same abstraction + OS calls.



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

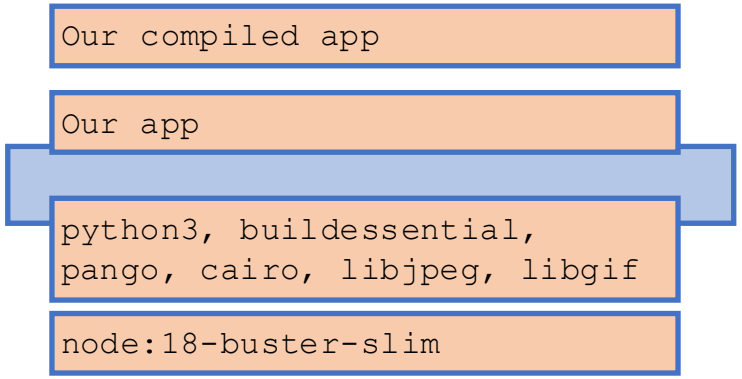
- Images are defined programmatically as a series of “build steps” (e.g. Dockerfile)
- Each step in the build becomes a “layer”
- Built images can be shared and cached
- To run a container, the layers are linked together with an “overlay” filesystem

```
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" ]
```

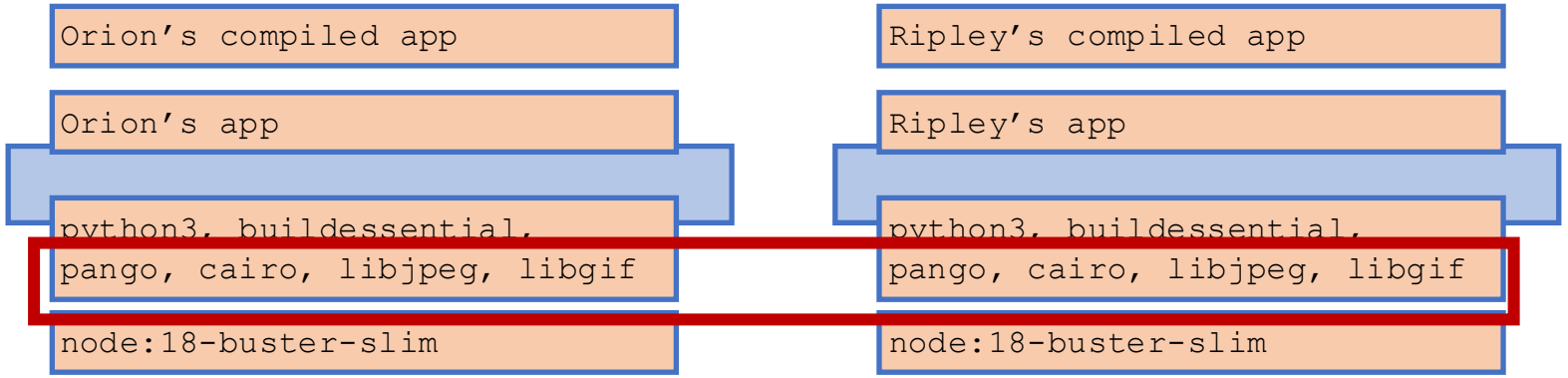
Example image specification (Dockerfile)



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)



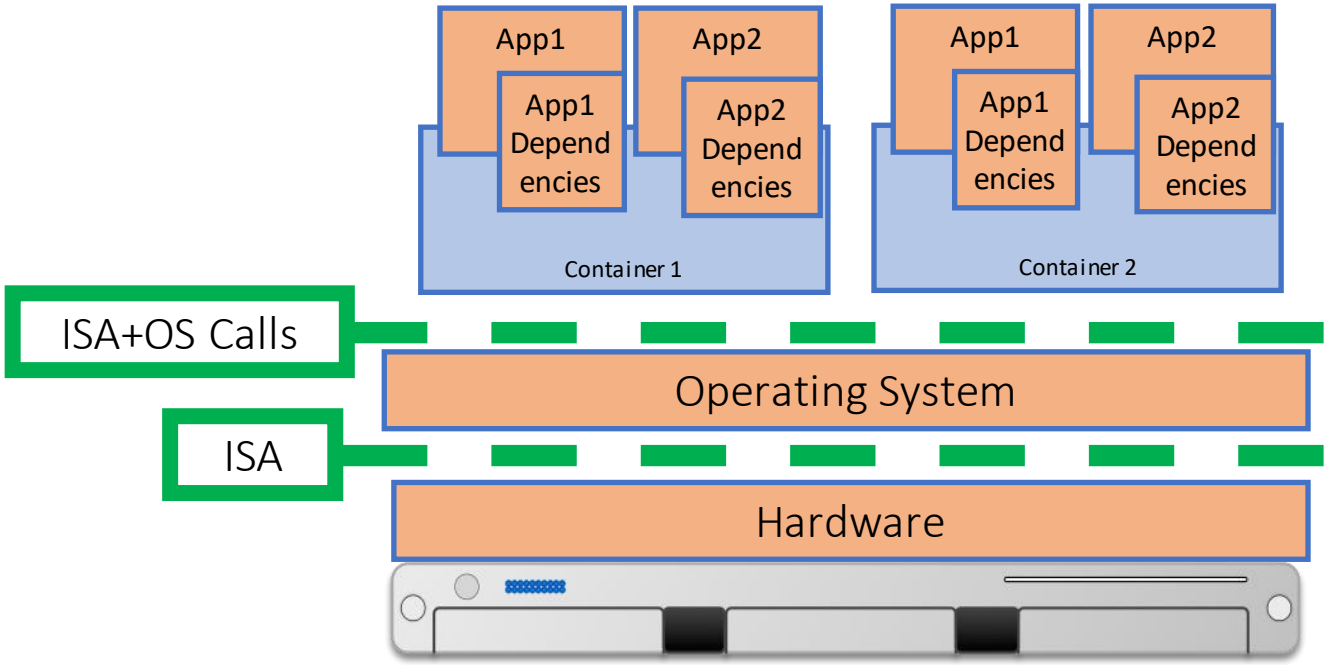
Two images, sharing two layers

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)

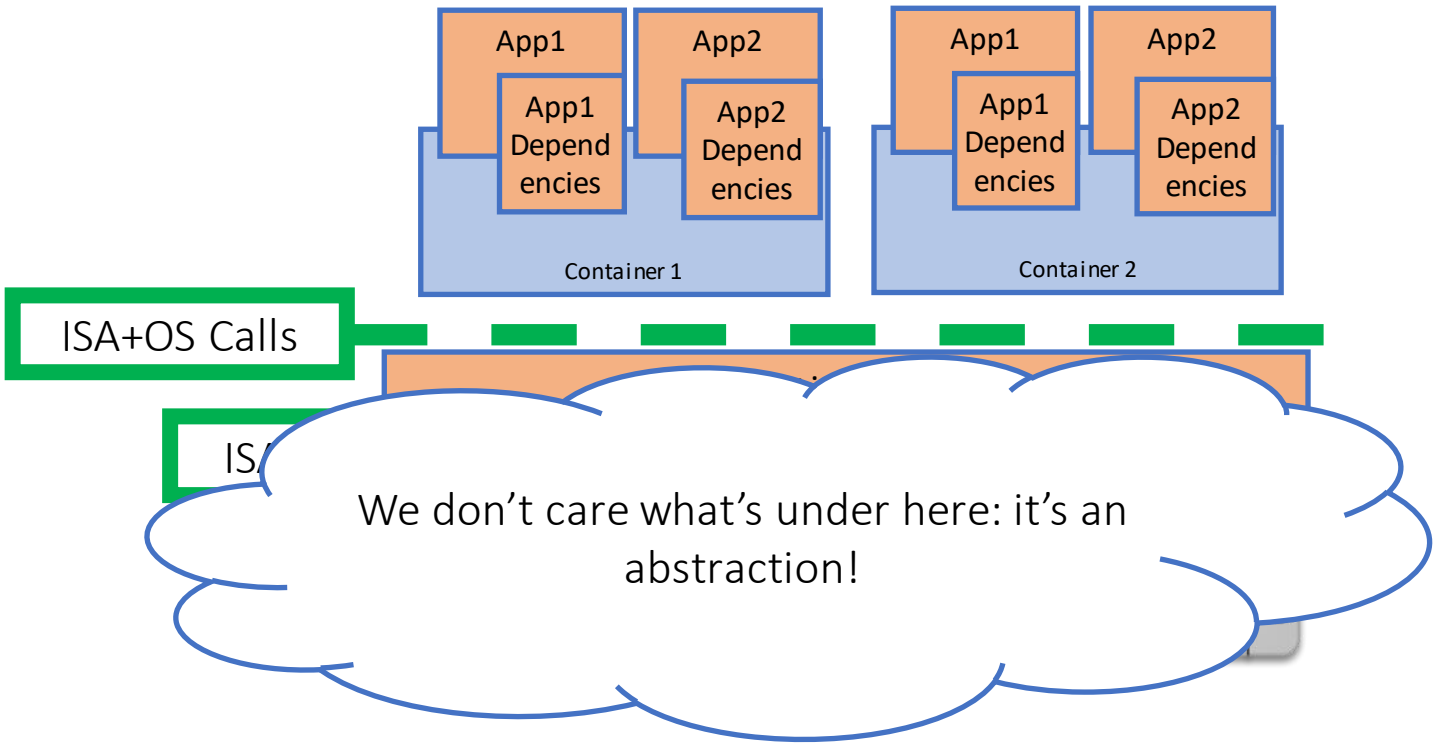
A container contains your apps and all their dependencies

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



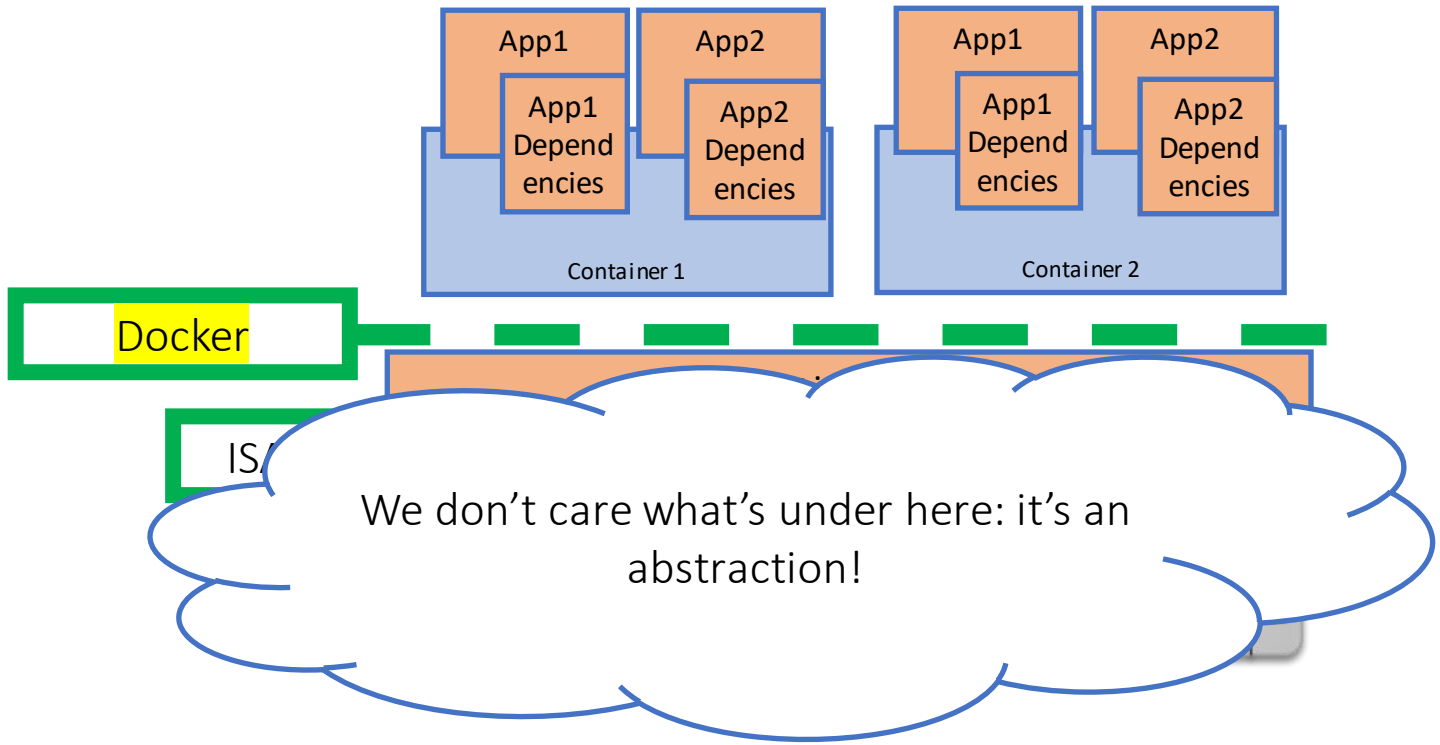
XaaS: Containers as a Service

- 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”)



NodeBB / Dockerfile

angelaz1 Initial NodeBB Commit — b6951a8 · last year — History

Code | Blame | 25 lines (16 loc) · 485 Bytes | Raw

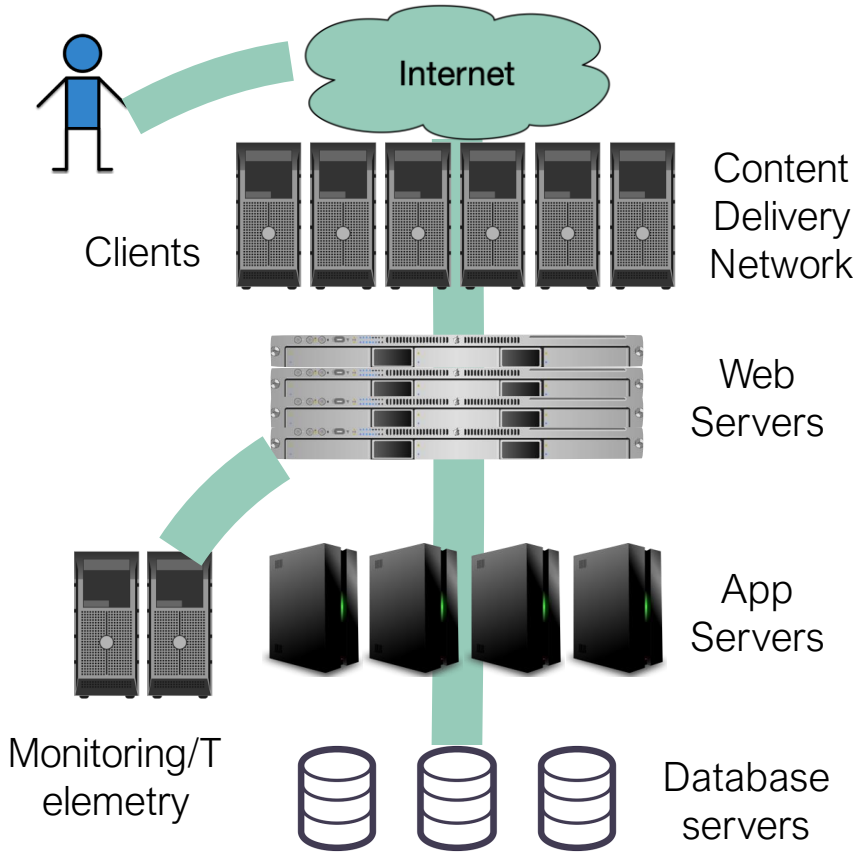
```
1  FROM node:lts
2
3  RUN mkdir -p /usr/src/app && \
4      chown -R node:node /usr/src/app
5  WORKDIR /usr/src/app
6
7  ARG NODE_ENV
8  ENV NODE_ENV $NODE_ENV
9
10 COPY --chown=node:node install/package.json /usr/src/app/package.json
11
12 USER node
13
14 RUN npm install --only=prod && \
15     npm cache clean --force
16
17 COPY --chown=node:node . /usr/src/app
18
19 ENV NODE_ENV=production \
20     daemon=false \
21     silent=false
22
23 EXPOSE 4567
24
25 CMD test -n "${SETUP}" && ./nodebb setup || node ./nodebb build; node ./nodebb start
```

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

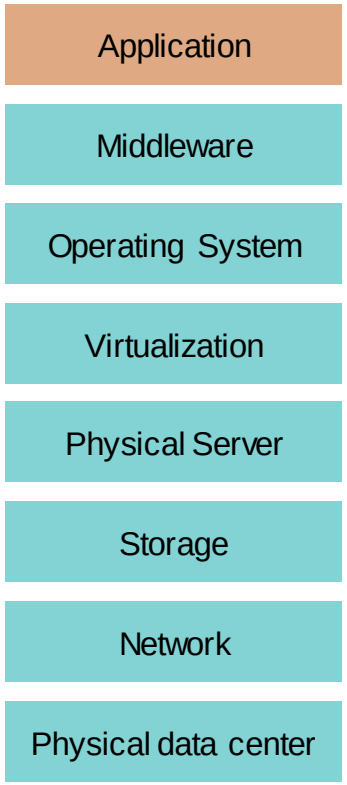
Platform-as-a-Service: vendor supplies OS + middleware

- 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

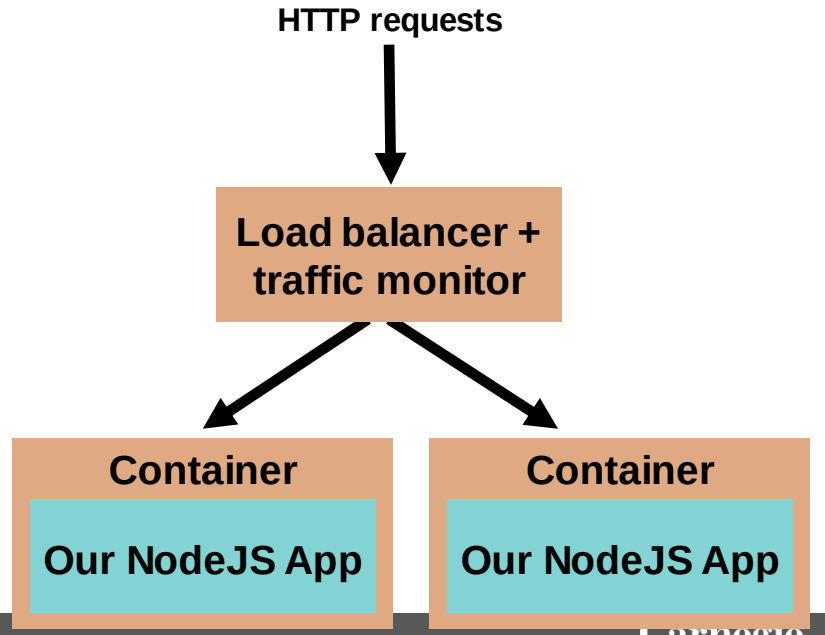
- **Platform-as-a-Service** provides components most apps need, fully managed by the vendor: load balancer, monitoring, application server
- Some PaaS run your app in a container: Heroku, AWS Elastic Beanstalk, Google App Engine, Railway, Vercel…
- Other PaaS run your apps as individual functions/event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



PaaS

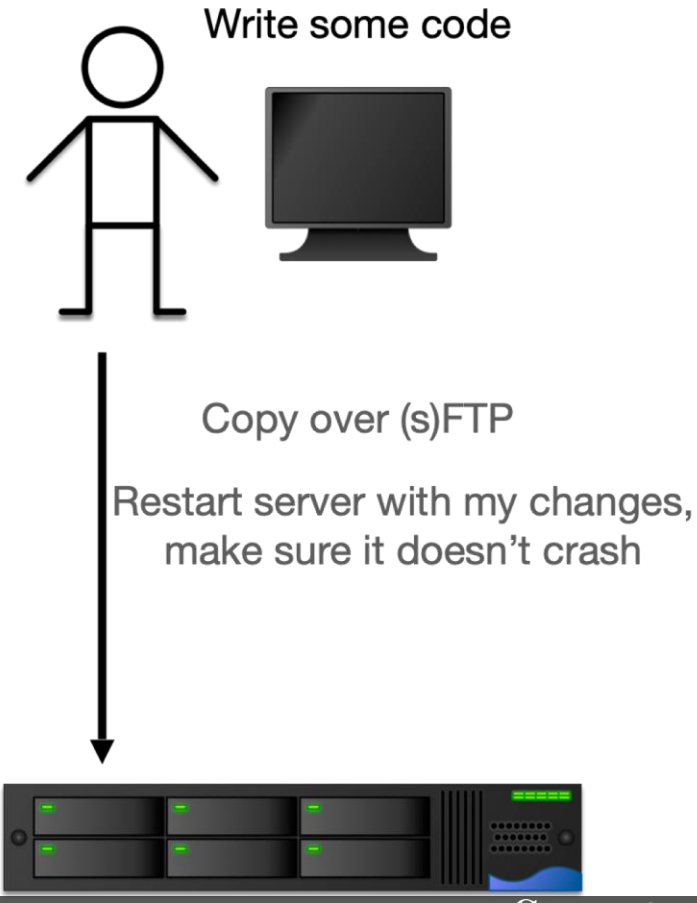
PaaS in the style of Heroku runs containers

- 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)



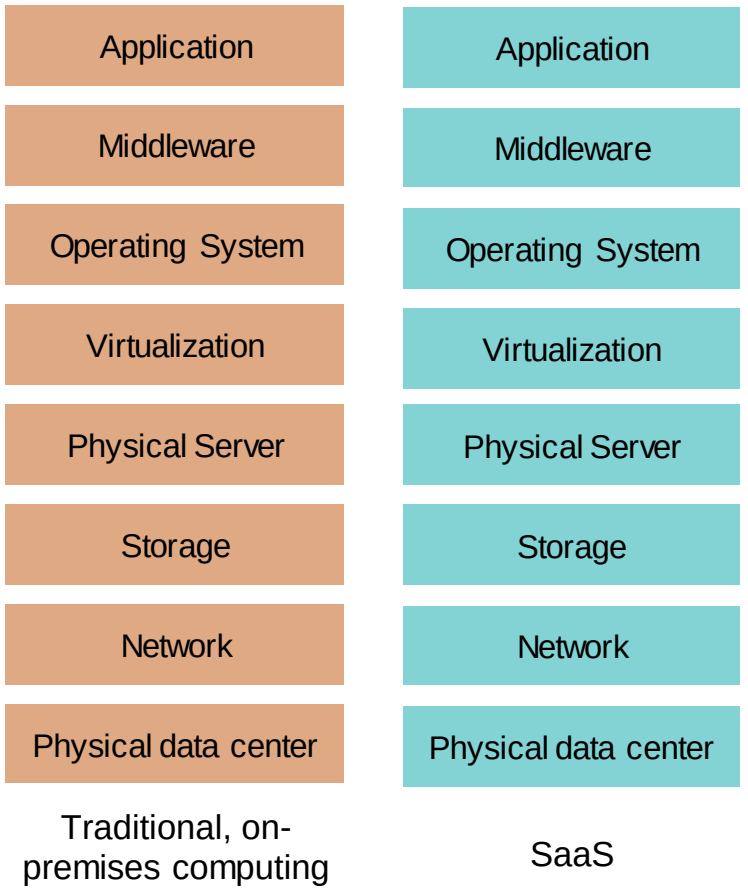
How to deploy web apps?

- 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



Cloud Infrastructure is best for variable workloads

- Consider:
  - Does your workload benefit from ability to scale up or down?
  - Variable workloads have different demands over time (most common)
  - Constant workloads require sustained resources (less common)
- Example:
  - Need to run 300 VMs, each 4 vCPUs, 16GB RAM
- Private cloud:
  - Dell PowerEdge Pricing (AMD EPYC 64 core CPUs)
  - 7 servers, each 128 cores, 512GB RAM, 3 TB storage = $162,104
- Public cloud:
  - Amazon EC2 Pricing (M7a.xlarge instances, $0.153/VM-hour)
  - 10 VMs for 1 year + 290 VMs for 1 month: $45,792.90
  - 300 VMs for 1 year: $402,084.00

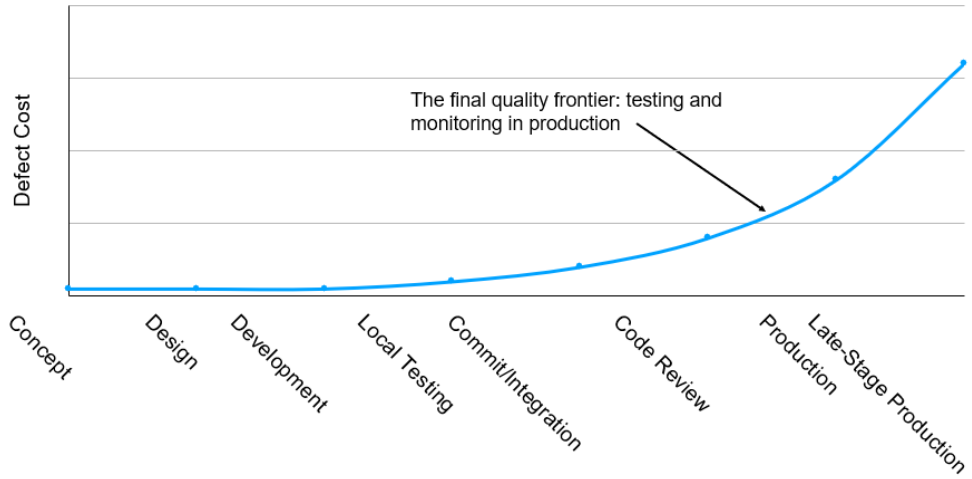
Public clouds are not the only option

- “Public” clouds are connected to the internet and available for anyone to use
  - Examples: Amazon, Azure, Google Cloud, DigitalOcean
- “Private” clouds use cloud technologies with on-premises, self-managed hardware
  - Cost-effective when a large scale of baseline resources are needed
  - Example management software: OpenStack, VMWare, Proxmox, Kubernetes
- “Hybrid” clouds integrate private and public (or multiple public) clouds
  - Effective approach to “burst” capacity from private cloud to public cloud

Cloud enables Continuous Delivery

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



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 ba
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. "

https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html

What could Knight capital have done better?

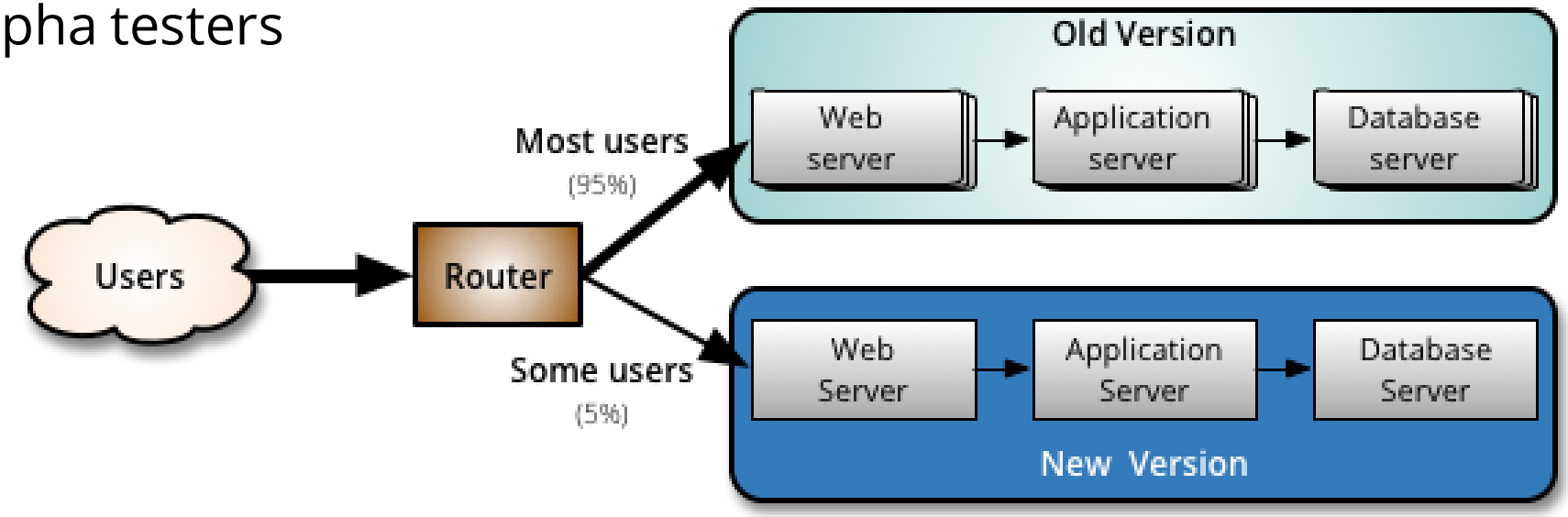
- Use capture/replay testing instead of driving market conditions in a test
- Avoid including “test” code in production deployments
- Automate deployments
- Define and monitor risk-based KPIs
- Create checklists for responding to incidents

Continuous Delivery != Immediate Delivery

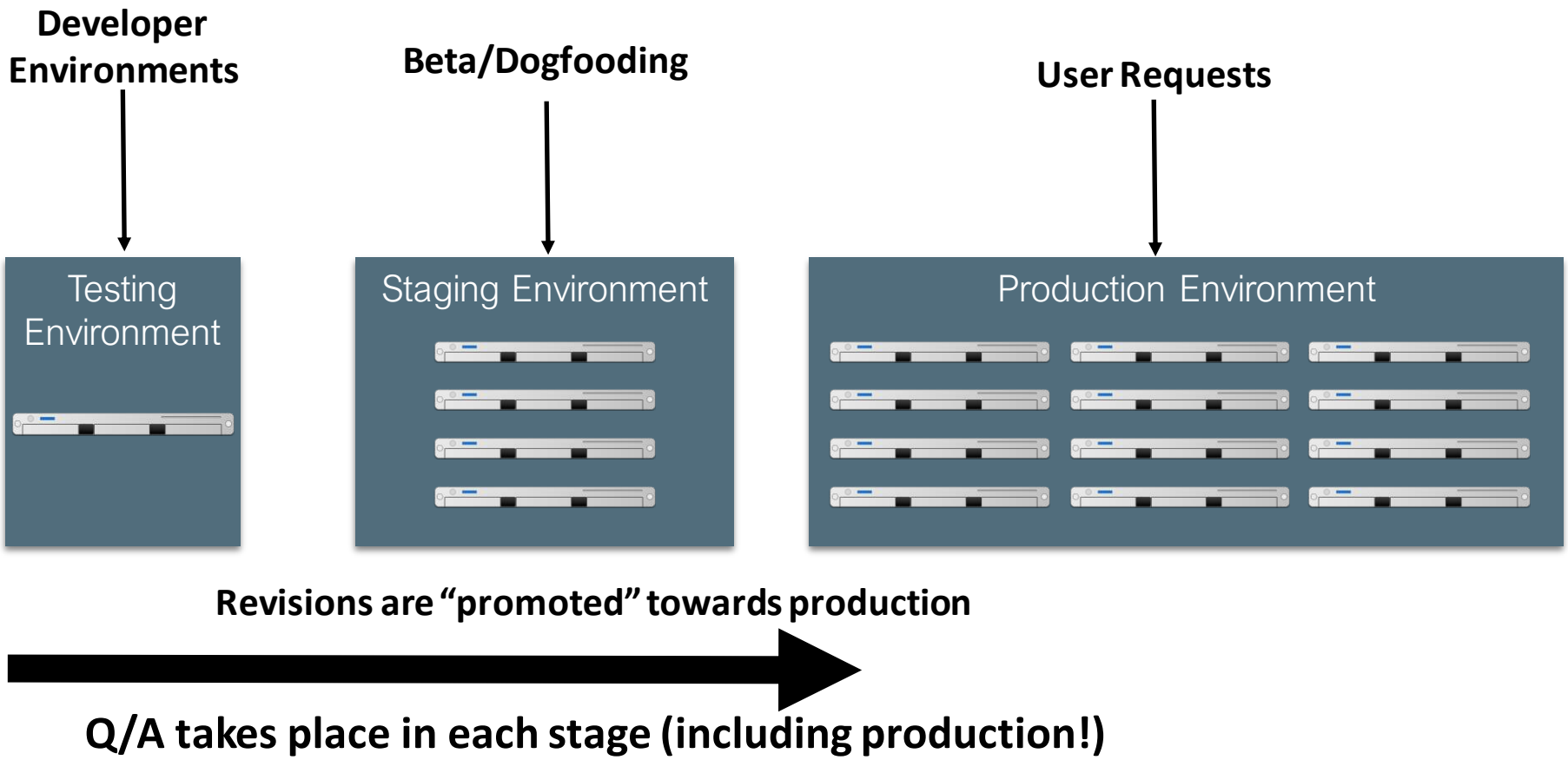
- Even if you are deploying every day (“continuously”), you still have some latency
- A new feature I develop today won't be released today
- But, a new feature I develop today can begin the **release pipeline** today (minimizes risk)
- **Release Engineer**: gatekeeper who decides when something is ready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

Split Deployments Mitigate Risk

- Idea: Deploy to a complete production-like environment, but don't have users use it, collect preliminary feedback
- Lower risk if a problem occurs in staging than in production
- Examples:
  - "Eat your own dogfood"
  - Beta/Alpha testers

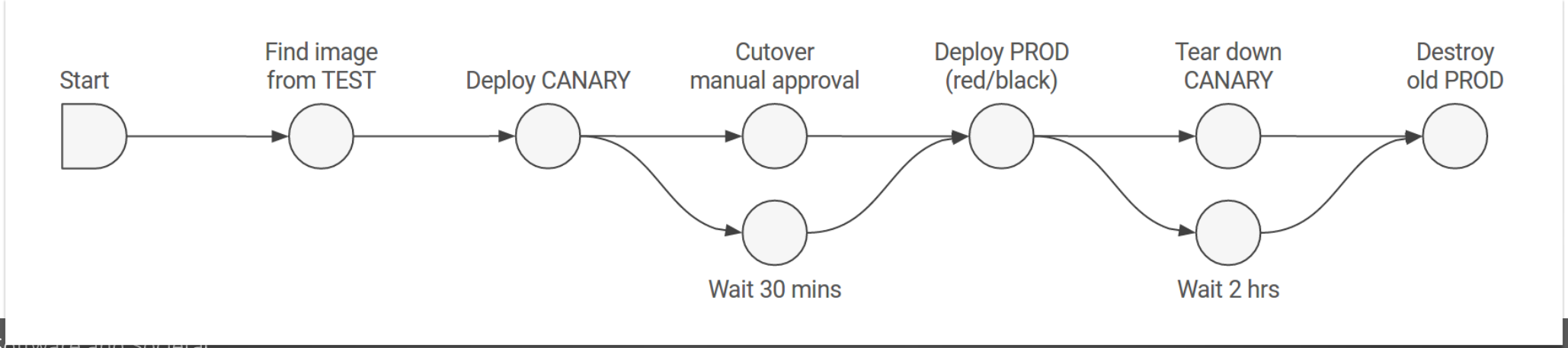


Continuous Delivery Leverages Relies on Staging Environments



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" (Open-Sourced by Netflix, c 2015)



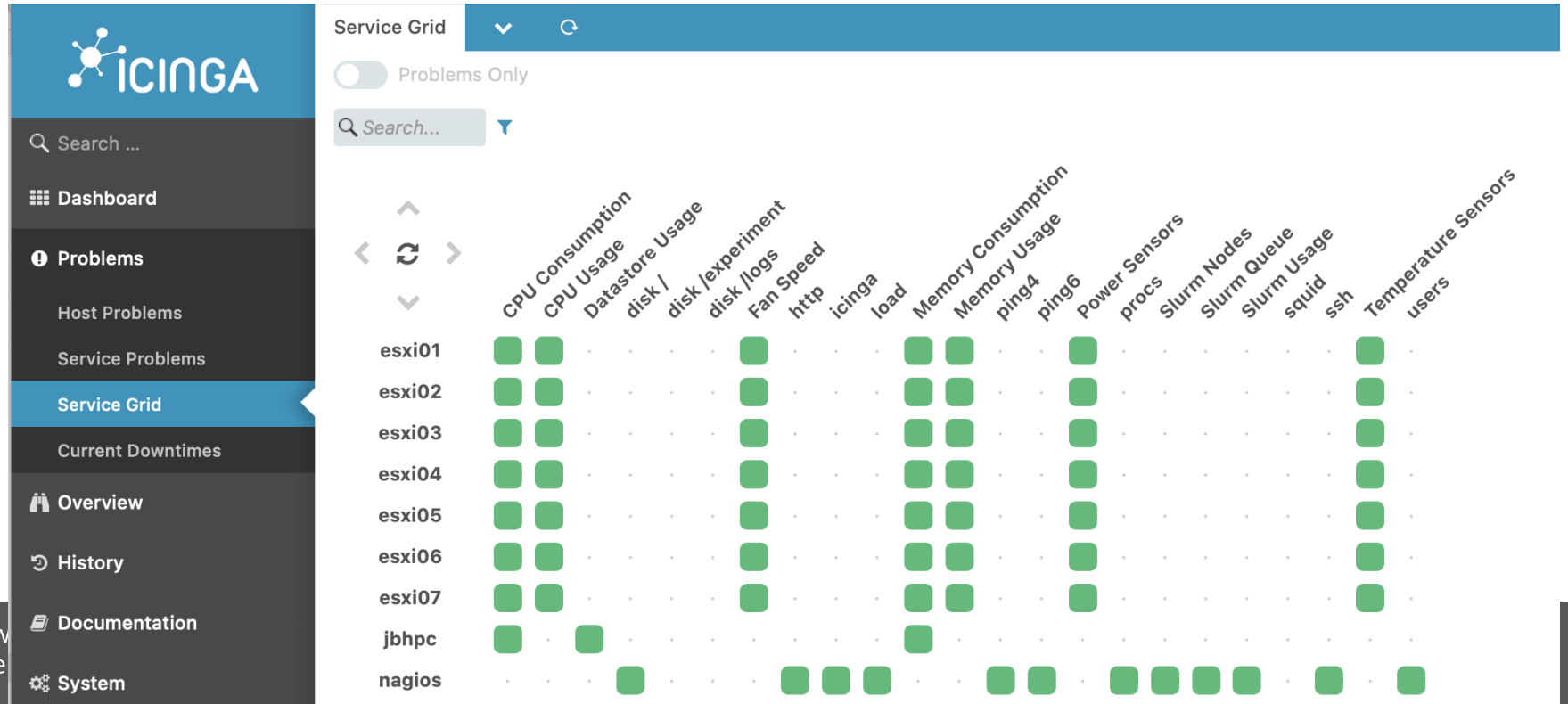
Example CD pipeline from Spinnaker's documentation: https://spinnaker.io/docs/concepts/#application-deployment

Continuous Delivery Relies on Monitoring

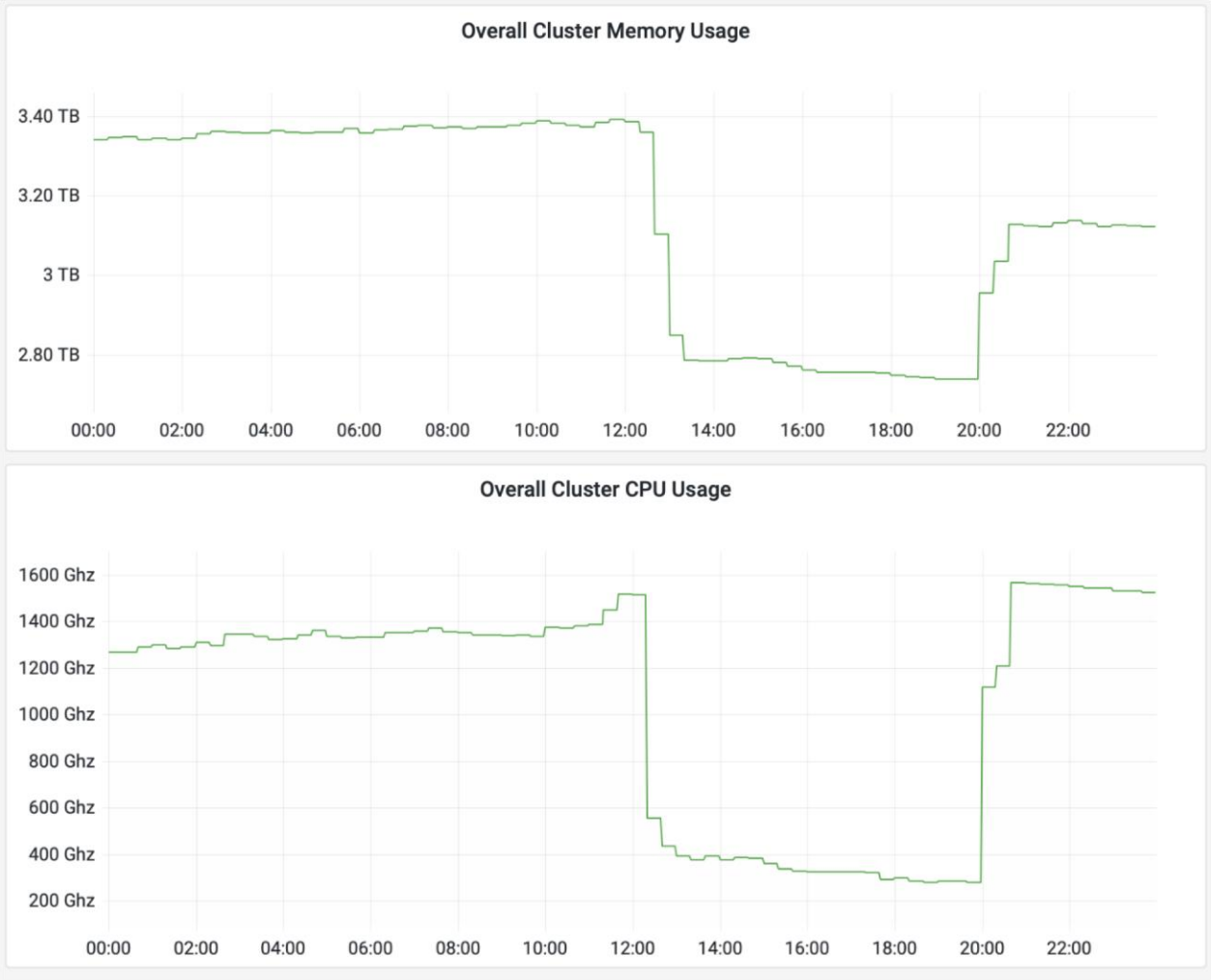
- Consider both direct (e.g. business) metrics, and indirect (e.g. system) metrics
- Hardware
  - Voltages, temperatures, fan speeds, component health
- OS
  - Memory usage, swap usage, disk space, CPU load
- Middleware
  - Memory, thread/db connection pools, connections, response time
- Applications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

Tools for Monitoring Deployments

- Nagios (c 2002): Agent-based architecture (install agent on each monitored host), extensible plugins for executing “checks” on hosts
- Track system-level metrics, app-level metrics, user-level KPIs



Monitoring can help identify operational issues



Grafana (AGPL, c 2014)

InfluxDB (MIT license, c 2013)

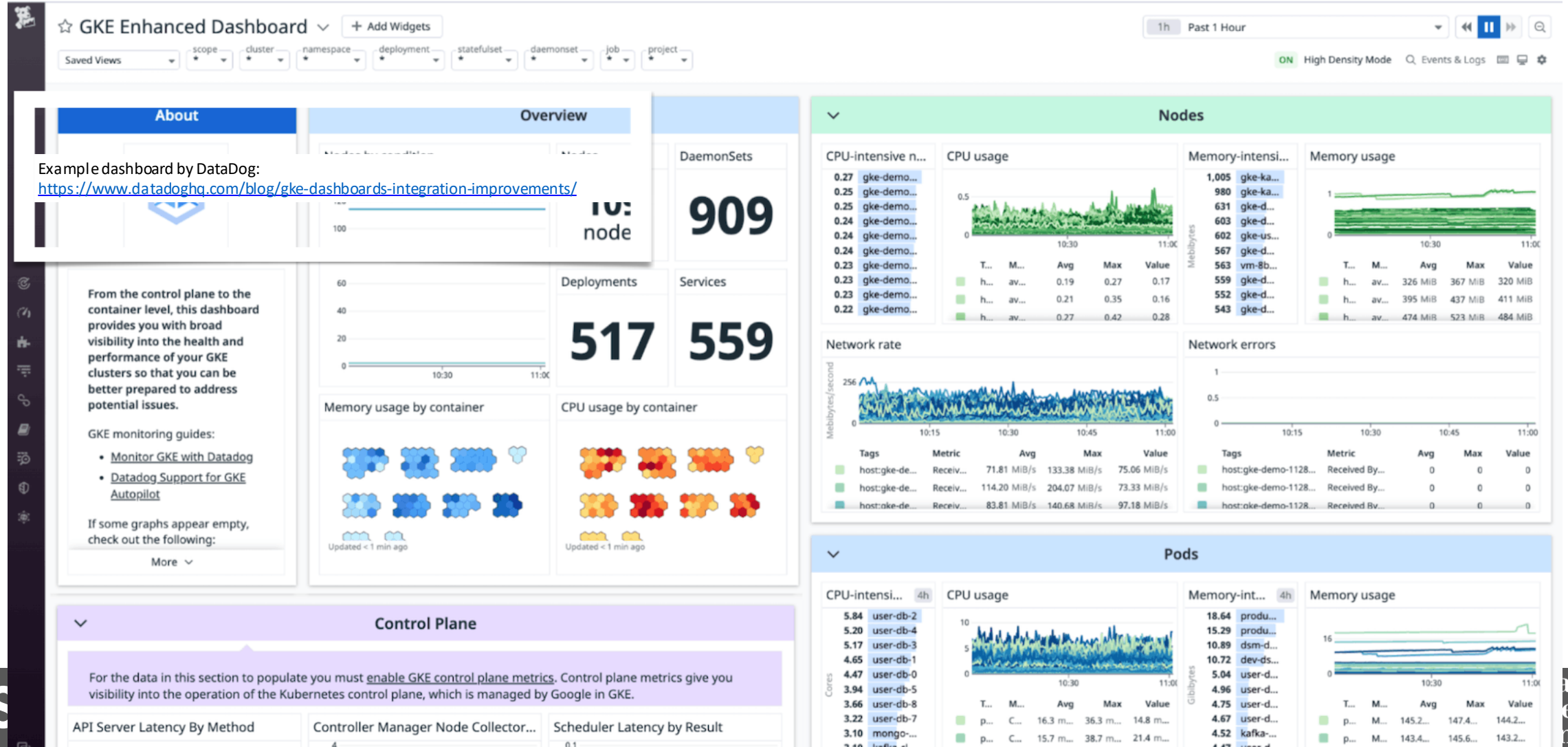
Continuous Delivery Tools Take Automated Actions

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

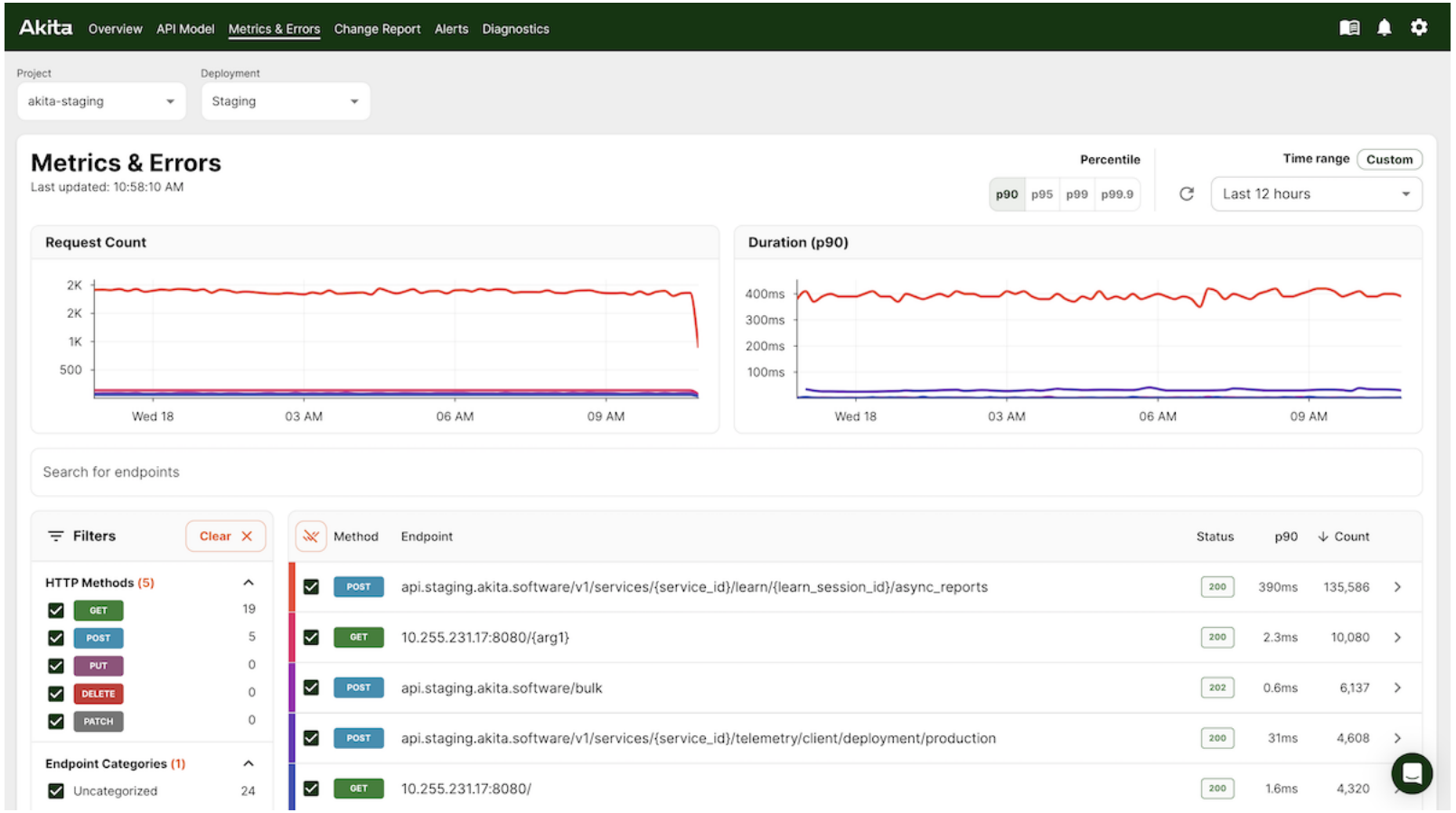
https://www.youtube.com/watch?v=qyzymU9ag

From Monitoring to Observability

- Understanding what is going on inside of our deployed systems

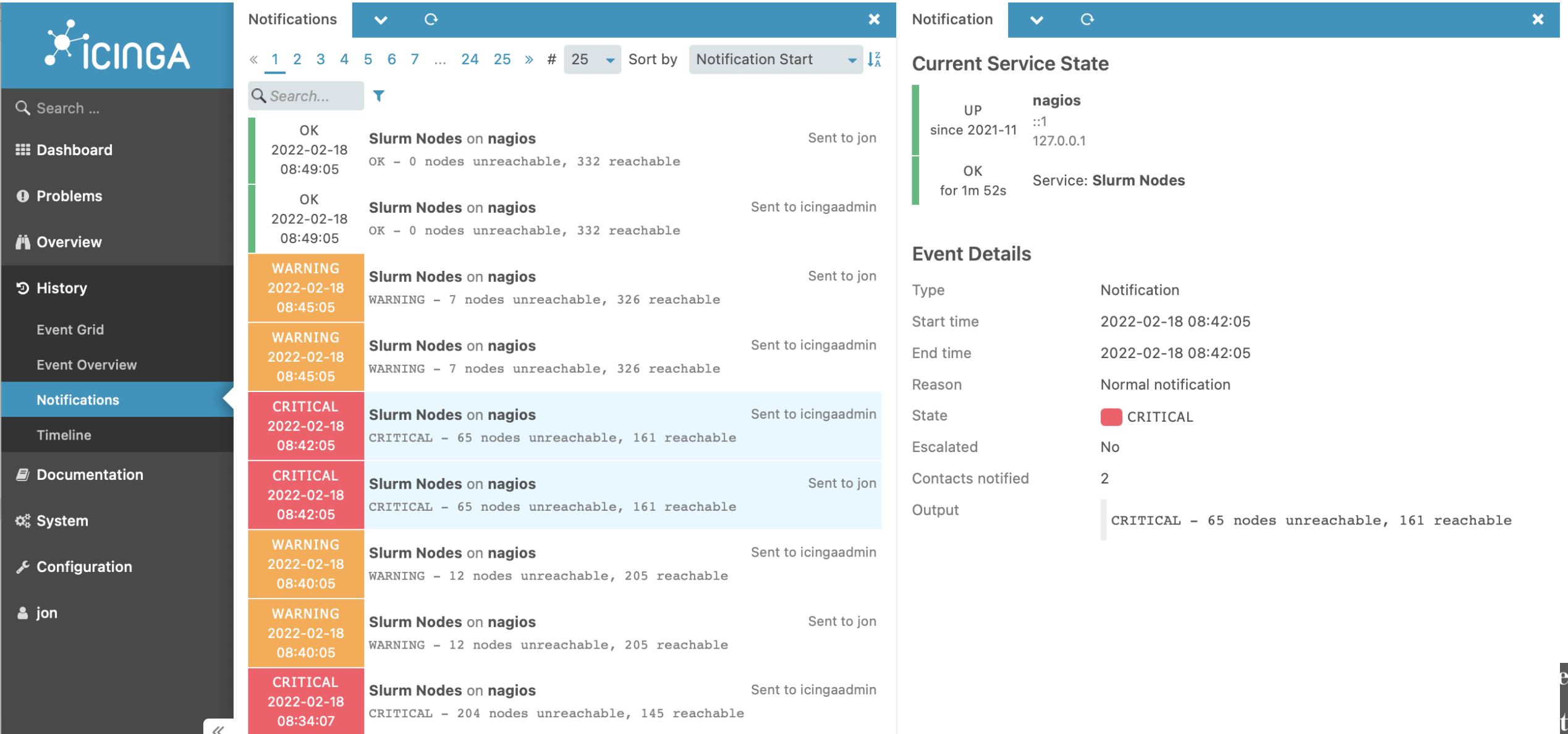


New Tools allow Observability inside of Apps, Too



Screenshot: https://www.akitasoftware.com/blog-posts/plug-and-play-endpoint-views-for-metrics-errors

Monitoring Services Take Automated Actions

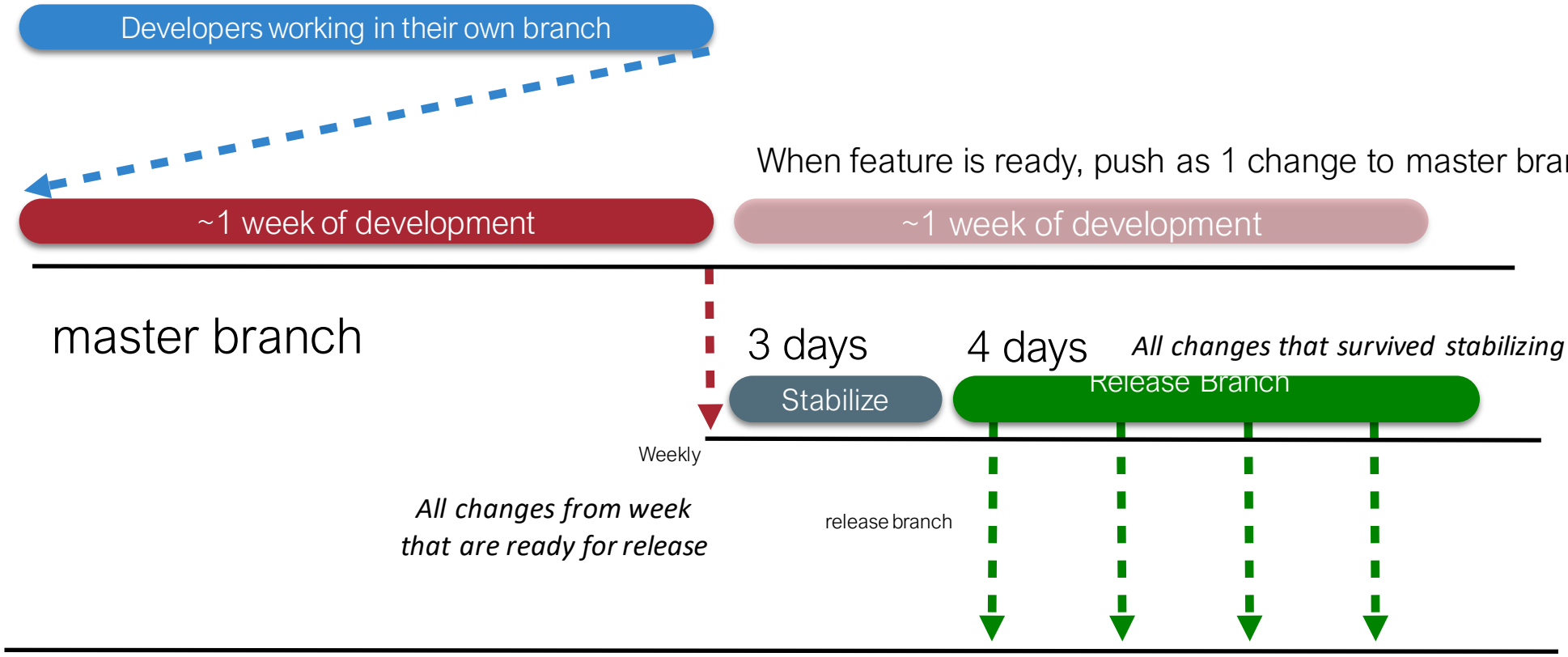


Beware of Metrics

- 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

Deployment Example: Facebook.com

- Pre-2016



Your change doesn't go out unless you're there that day at that time to support it!

3x Daily

"When in doubt back out"

Deployment Example

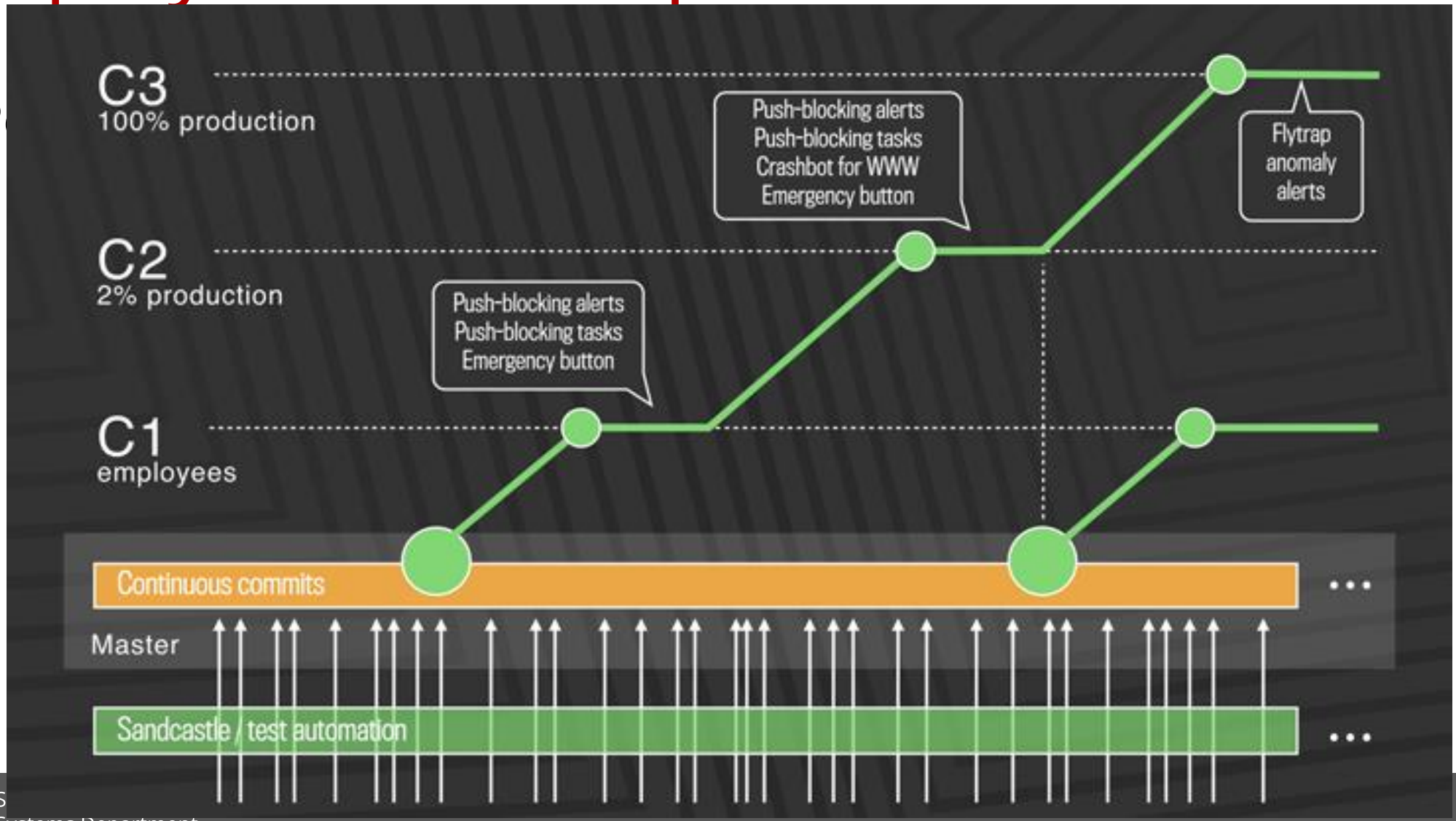
"Our main goal was to make sure that the new system made people's experience better — or at least, didn't make it worse. After a year of planning and development, over the course of three days **we enabled 100% of our production web servers to run code deployed directly from master**"

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

"Rapid release at massive scale" https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/

Deployment Example

- P



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