Week:
7 of 12

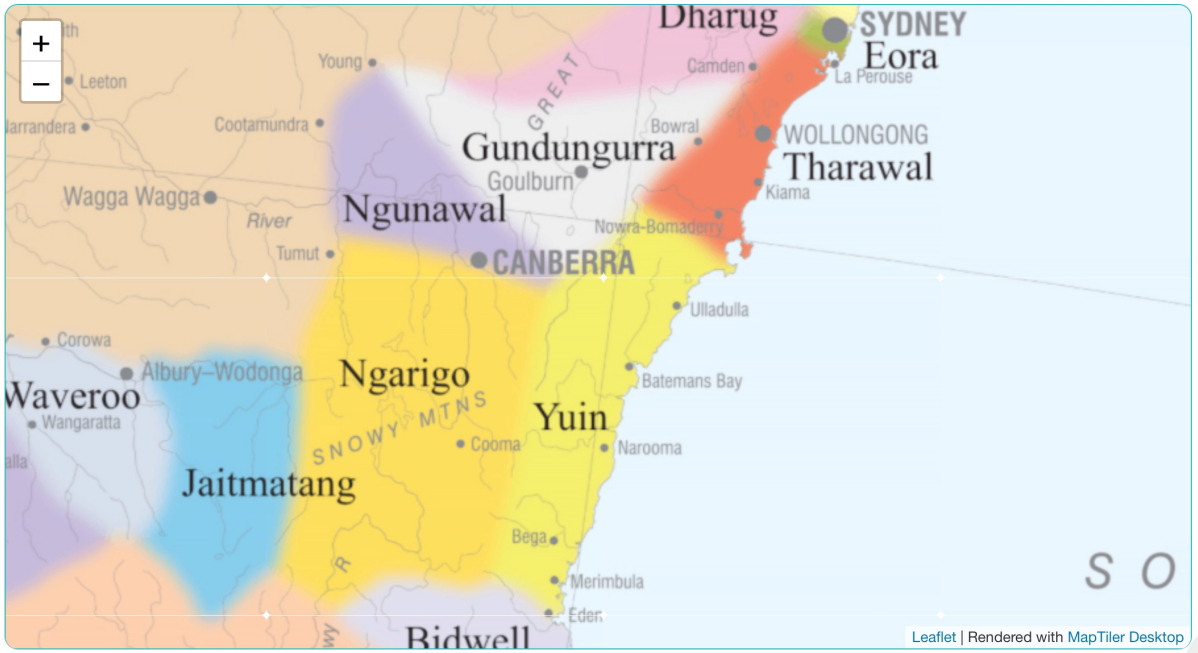
# COMP 2120 / COMP 6120

# DEVOPS

A/Prof Alex Potanin

# ANU Acknowledgment of Country

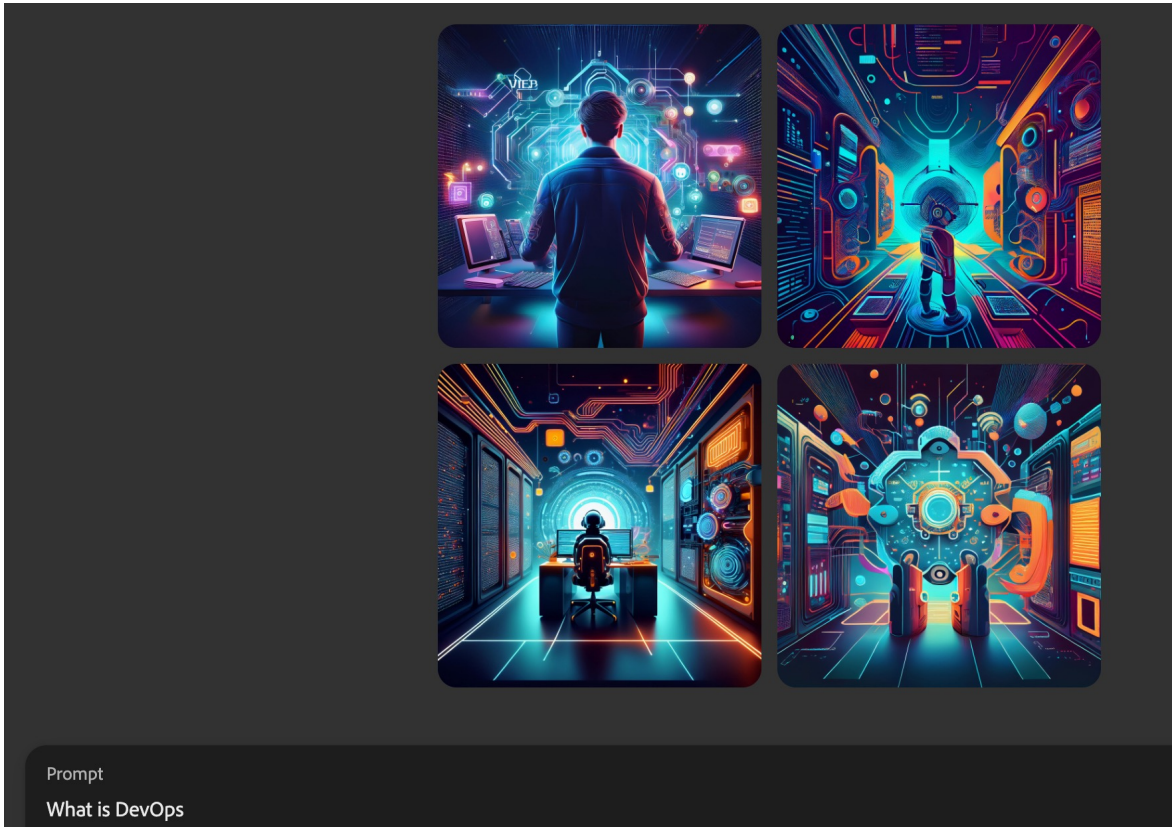

"We acknowledge and celebrate the First Australians on whose traditional lands we meet, and pay our respect to the elders past and present."




https://aiatsis.gov.au/explore/map-indigenous-australia

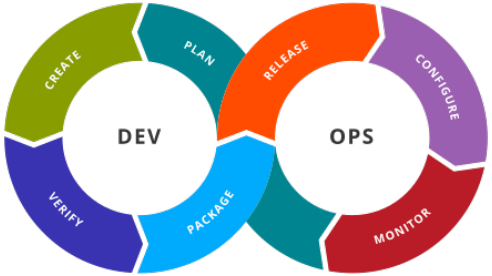
# Today

- What is DevOps
- CI: Continuous Integration
- CD: Continuous Deployment
- Infrastructure as Code
- Monitoring

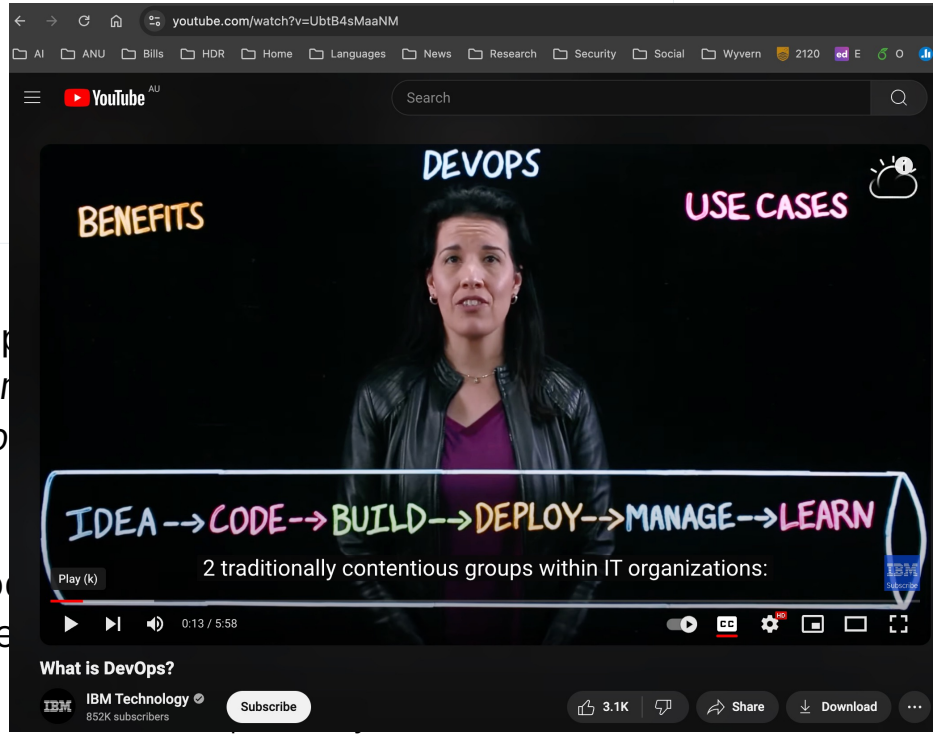



# What Is DevOps

# What is DevOps?



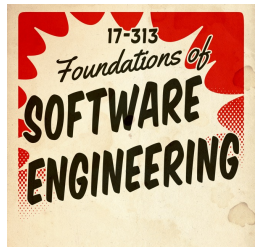


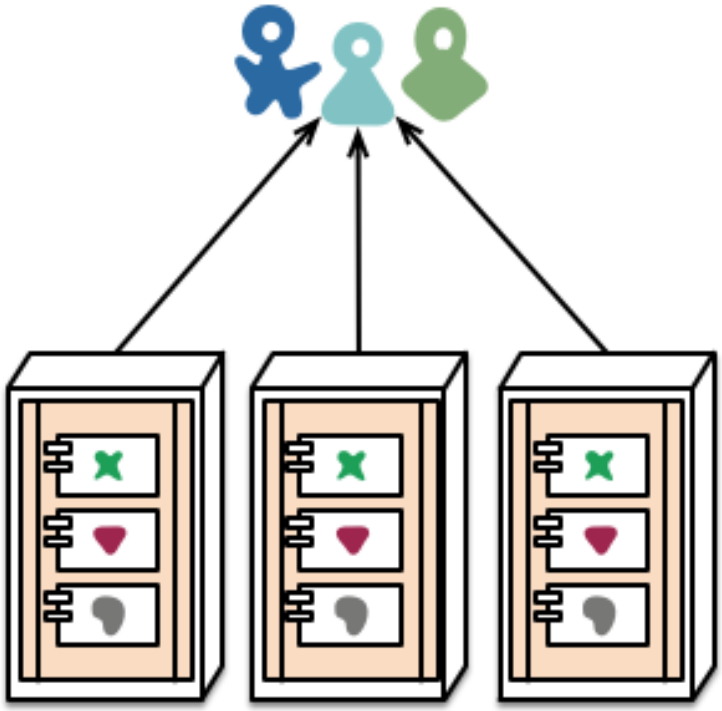



Bringing together two traditionally sep
- Development, typically *measured or*
- Operations, typically *measured thro*

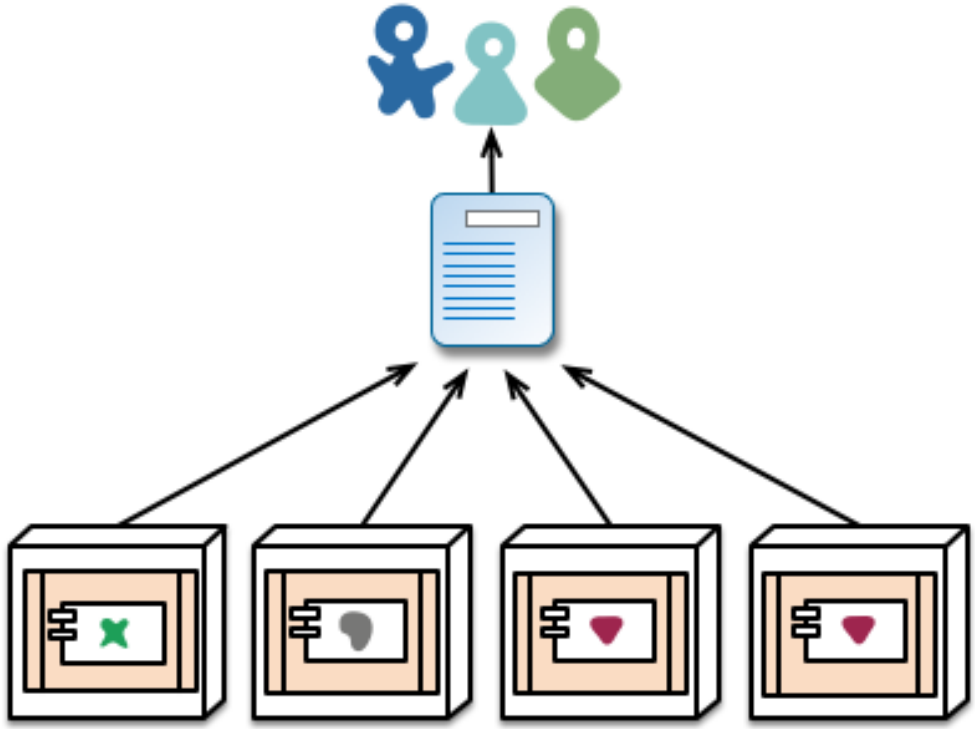
Benefits:
- *Increased* Velocity: how quickly pro
- *Increased* Quality: successful delive

# Deployment and Evolution



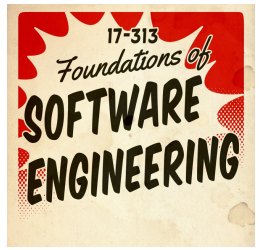
monolith - multiple modules in the same process

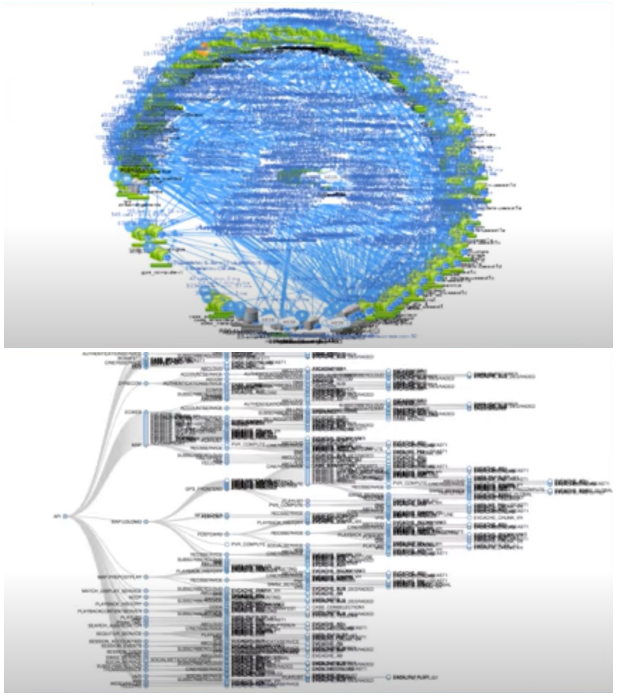


Source: http://martinfowler.com/articles/microservices.html

microservices - modules running in different processes

# Netflix: Microservice Architecture

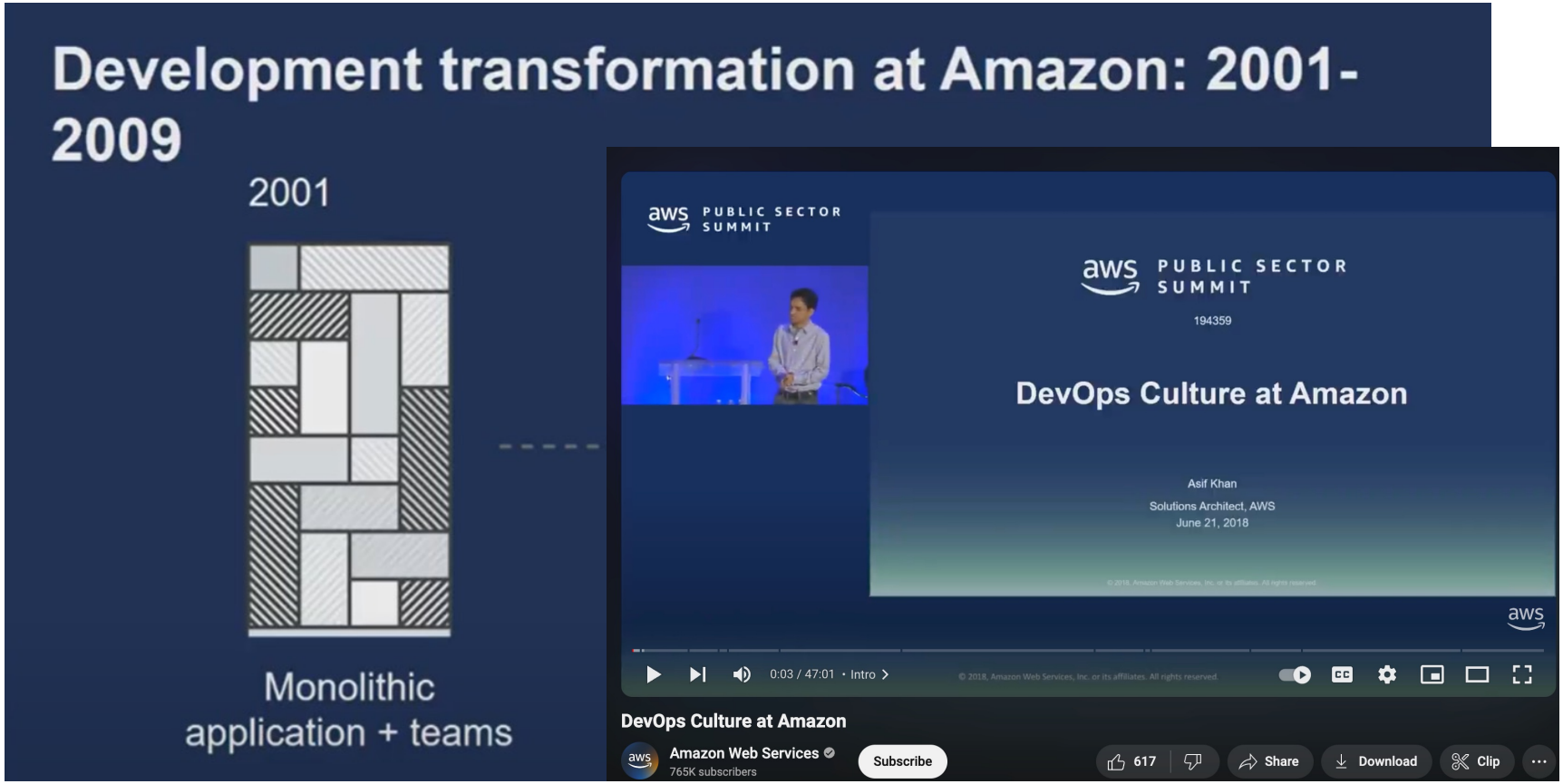


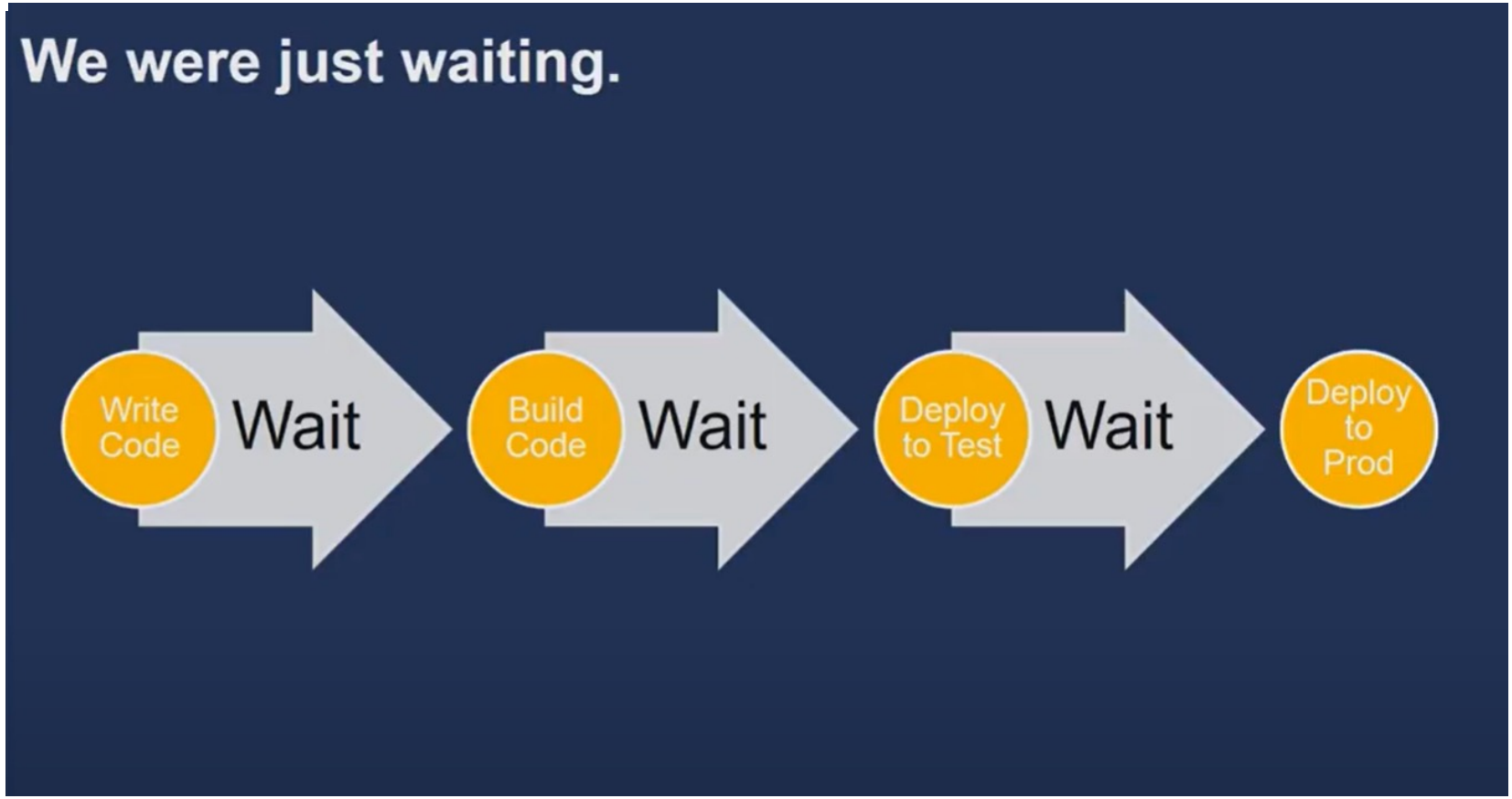
- 100s of microservices
- 1,000s of production changes per day
- 10,000s of virtual machines
- 100,000s of customer interactions per second
- 1,000,000s of metrics per minute (actually, 2 million)

- 81.5 million customers

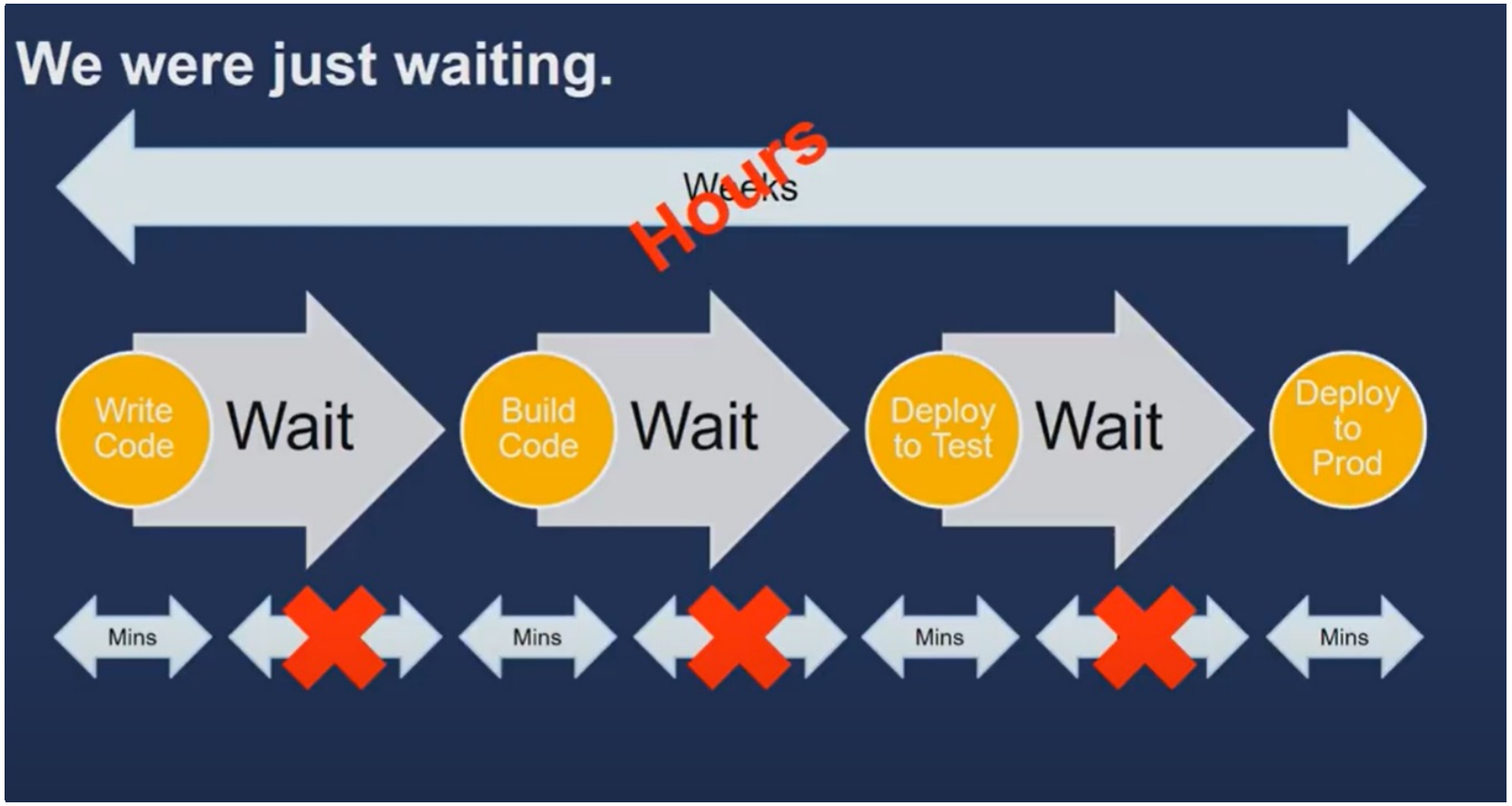
- 10s of operations engineers
- no single engineer knows the entire application




reference: https://www.youtube.com/watch?v=mBU3AJ3j1rg

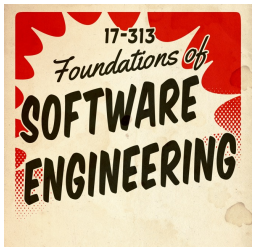



reference: https://www.youtube.com/watch?v=mBU3AJ3j1rg




reference: https://www.youtube.com/watch?v=mBU3AJ3j1rg

# How do we get to DevOps?




Goals:

1. *Technological:* Automated process for moving code from dev to release.

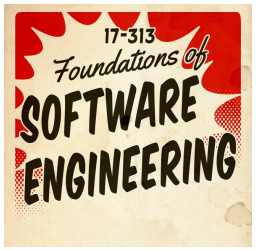
   Starting with check-in, build, unit test, build artifact,
   integration test, load test, as moves through stage to production,
   finally, with monitoring and other telemetry.

2. *Cultural:* Building cohesive, multidisciplinary teams.

   Typically, developers are the “first responders” when things go bad in production.
   Sense of “ownership” by the developer all the way from inception to release.

reference: https://www.youtube.com/watch?v=UbtB4sMaaNM

# What can it look like when it's done?




Netflix Spinnaker (open-source CI/CD fully automated pipeline):

- Takes code from code repository to production.
- Allows developers to specify required tests.
- Determines where, how code should be run in system (e.g., replication, placement.)
- Supports canary deployments, traffic management.
- Just publish the repo!

**5x** lower change failure rate

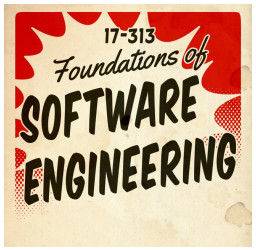
**440x** faster from commit to deploy

**46x** more frequent deployments

**44%** more time spent on new features and code

reference: Puppet State of the DevOps Report 2017

# What do we need to practice for DevOps?




Continuous Integration (CI)

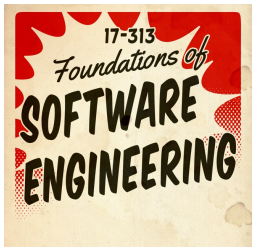
1. Constant testing as code is checked-in/pushed to the repository (e.g., GH hooks, etc.)
2. Verify the build process works (i.e., parsing, compilation, code generation, etc.)
3. Verify unit tests pass, style checks pass, other static analysis tools.
4. Build artifacts

Continuous Delivery & Deployment (CD)

1. Moving build artifacts from test -> stage -> prod environments.
   Environments always differ! (e.g., ENV, PII, data, etc.)
2. Gate code, if necessary, from advancing without manual approval.
   Useful when initially transitioning applications into a modern DevOps pipeline.

reference: https://www.youtube.com/watch?v=mBU3AJ3j1rg

# CI/CD




Continuous Integration (CI)

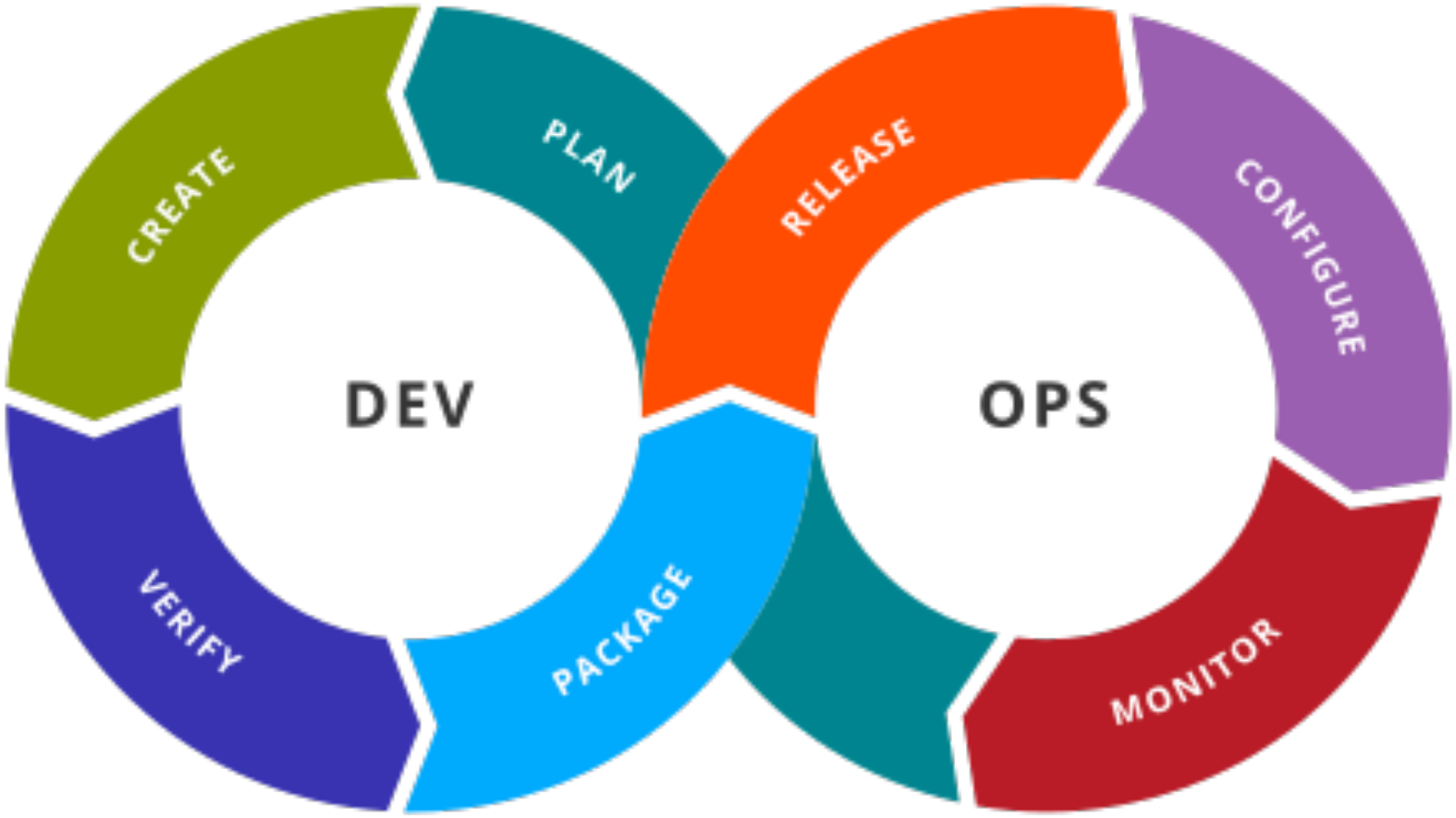
1. Commit and check-in code frequently (always can squash later)
2. Commits build on previous commits (know precisely where the build breaks)
3. Automated feedback and testing on commits
4. Artifact creation (e.g., container images, WAR files, etc.)
5. Ensure code, supporting infrastructure, documentation are all versioned together

Continuous Deployment (CD)

1. Artifacts automatically shipped into test, stage, production environments
2. Prevents “manual” deployment, avoids “manual” steps, early detection of problems
3. Can be tied to a “manual” promotion technique to advance through environments
4. Multi-stage deployment with automatic rollback on failure detection

reference: https://www.youtube.com/watch?v=mBU3AJ3j1rg

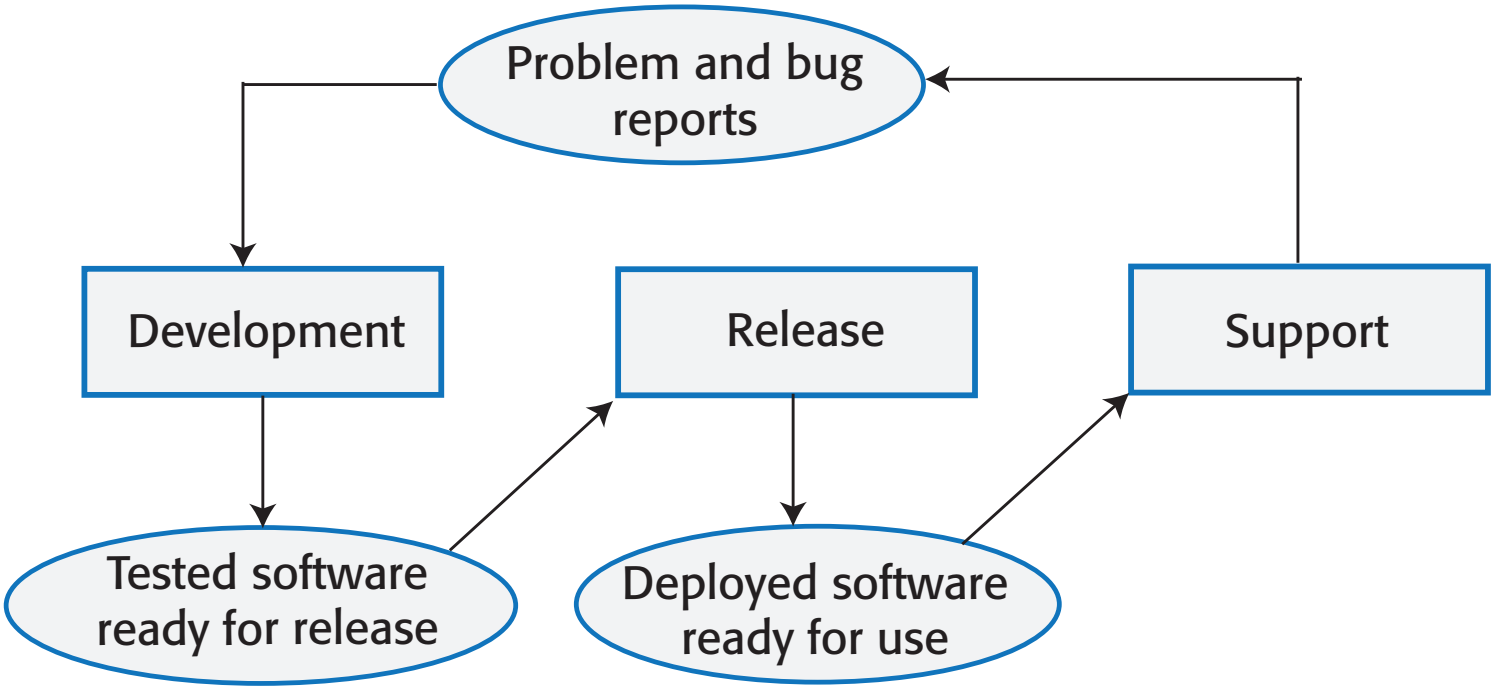
# DevOps Phases



# Software Support


- Traditionally, separate teams were responsible for software development, software release and software support.
- The development team passed over a ‘final’ version of the software to a release team. This team then built a release version, tested this and prepared release documentation before releasing the software to customers.
- A third team was responsible for providing customer support.
  - The original development team were sometimes also responsible for implementing software changes.
  - Alternatively, the software may have been maintained by a separate ‘maintenance team’.

# Development, release and support

# DevOps


- There are inevitable delays and overheads in the traditional support model.
- To speed up the release and support processes, an alternative approach called DevOps (Development + Operations) has been developed.
- Three factors led to the development and widespread adoption of DevOps:
  - Agile software engineering reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment.
  - Amazon re-engineered their software around services and introduced an approach in which a service was developed and supported by the same team. Amazon's claim that this led to significant improvements in reliability was widely publicized.
  - It became possible to release software as a service, running on a public or private cloud. Software products did not have to be released to users on physical media or downloads.

# DevOps

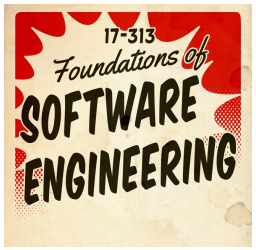

# DevOps principles


- ***Everyone is responsible for everything***
  All team members have joint responsibility for developing, delivering and supporting the software.
- ***Everything that can be automated should be automated***
  All activities involved in testing, deployment and support should be automated if it is possible to do so. There should be minimal manual involvement in deploying software.
- ***Measure first, change later***
  DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

# Benefits of DevOps


- **Faster deployment**
  Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.
- **Reduced risk**
  The increment of functionality in each release is small so there is less chance of feature interactions and other changes causing system failures and outages.
- **Faster repair**
  DevOps teams work together to get the software up and running again as soon as possible. There is no need to discover which team were responsible for the problem and to wait for them to fix it.
- **More productive teams**
  DevOps teams are happier and more productive than the teams involved in the separate activities. Because team members are happier, they are less likely to leave to find jobs elsewhere.

# What do we need to practice for DevOps?



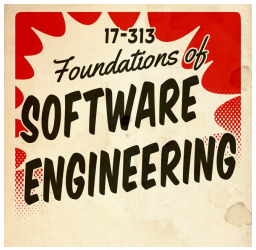

Infrastructure as Code

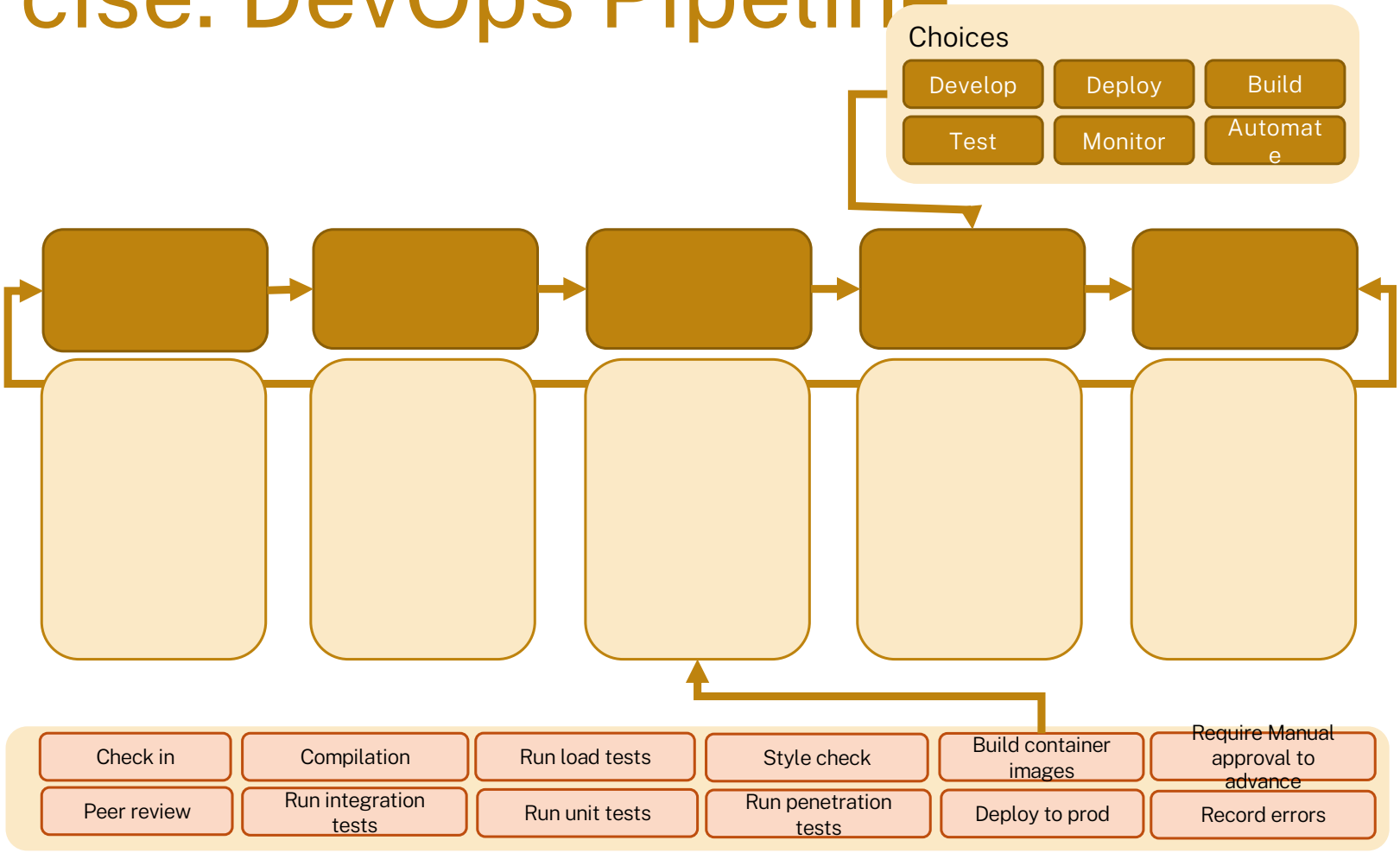
1. Required resources (e.g., cloud services, access policies, etc.) are created by code.
   No UI provisioning, no manual steps (avoid: easy to forget, time consuming!)
2. "Immutable Infrastructure"
   No update-in-place (e.g., SSH to server.)
   Replace with new instances, decommission old instances.
3. *Nothing to prod without it being in code, checked-in, versioned along side code!*

Observability (Monitoring, Logging, Tracing, Metrics)

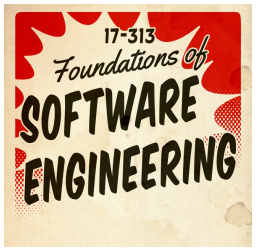
1. Be able to know how your application is running in production
2. Track and analyze low-level metrics on performance, resource allocation
3. Capture high-level metrics on application behavior
   1. What's "normal"?
   2. What's abnormal?

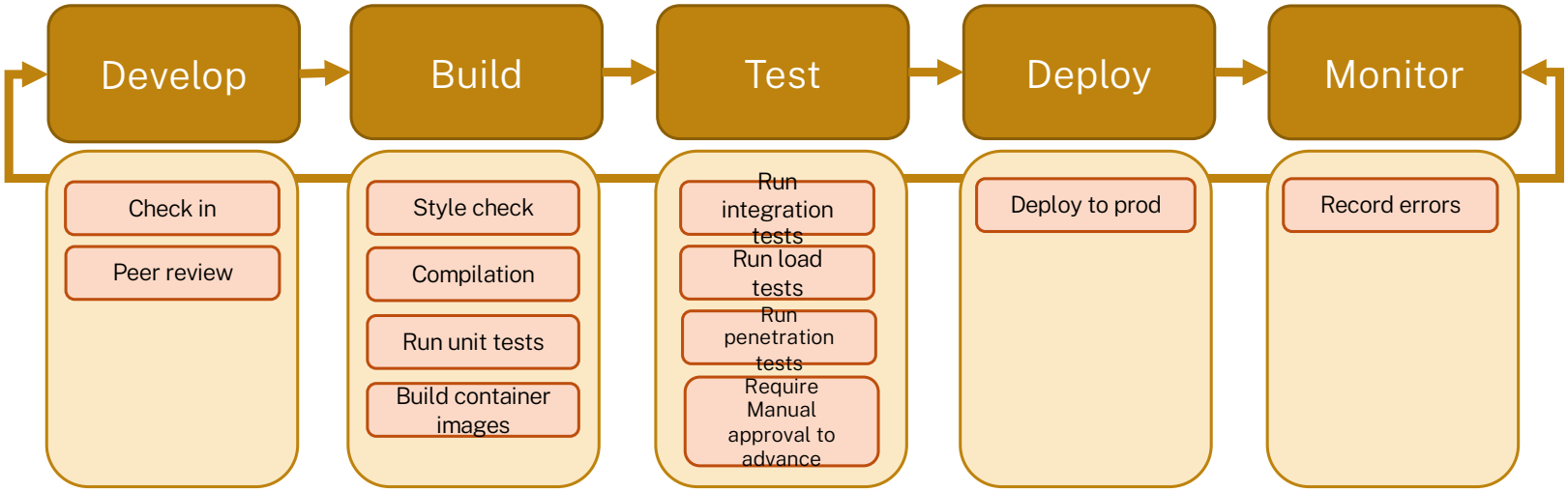
# Exercise: DevOps Pipeline



# A Typical DevOps Pipeline



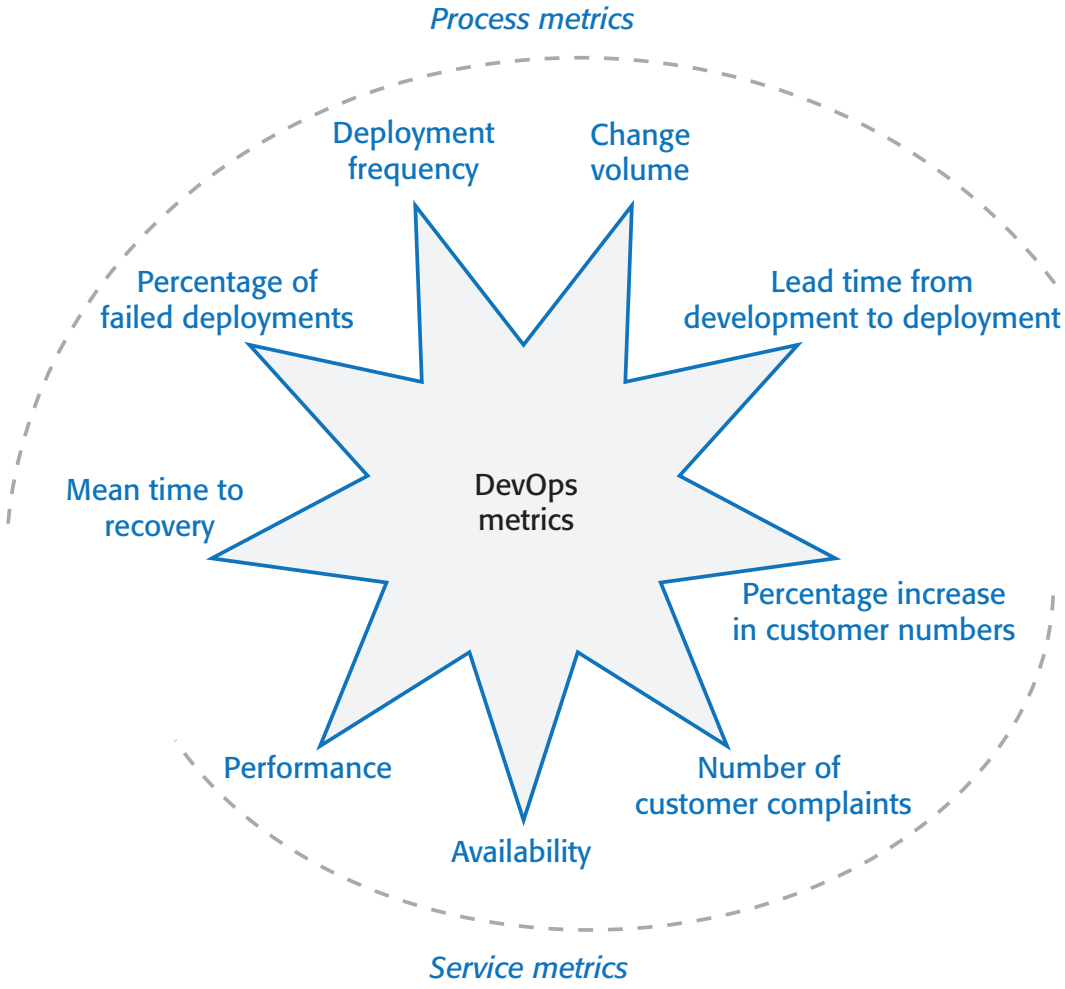
# DevOps Measurement


- After you have adopted DevOps, you should try to continuously improve your DevOps process to achieve faster deployment of better-quality software.
- There are four types of software development measurement:
  - Process measurement You collect and analyse data about your development, testing and deployment processes.
  - Service measurement You collect and analyse data about the software's performance, reliability and acceptability to customers.
  - Usage measurement You collect and analyse data about how customers use your product.
  - Business success measurement You collect and analyse data about how your product contributes to the overall success of the business.

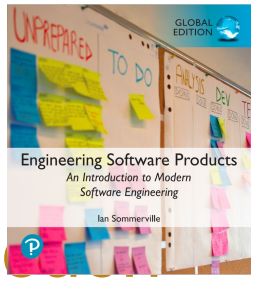
# Automating Measurement


- As far as possible, the DevOps principle of automating everything should be applied to software measurement.
- You should instrument your software to collect data about itself and you should use a monitoring system to collect data about your software's performance and availability.
- Some process measurements can also be automated.
  - However, there are problems in process measurement because people are involved. They work in different ways, may record information differently and are affected by outside influences that affect the way they work.

# Metrics used in the DevOps scorecard

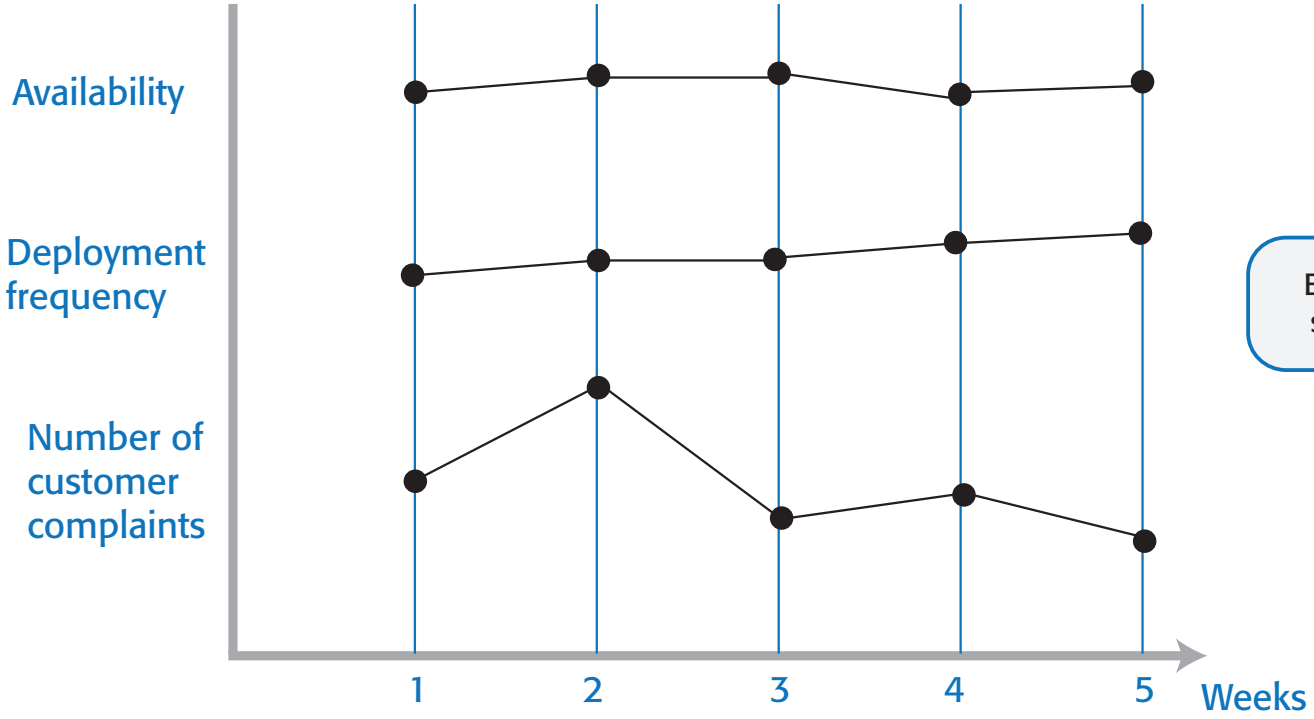



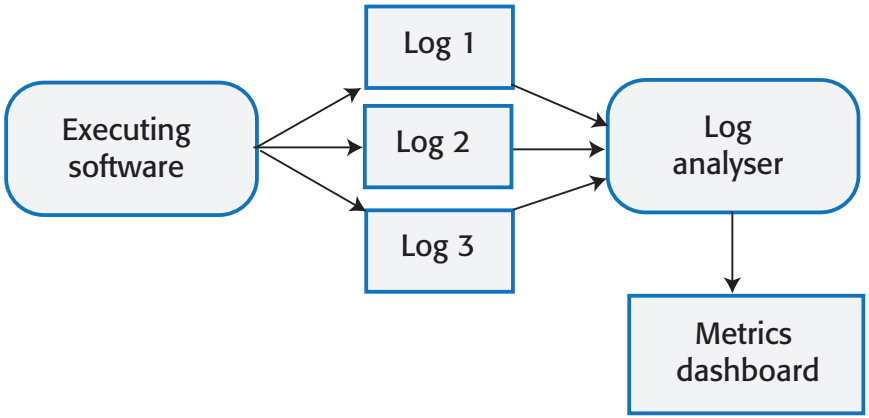
# Metrics scorecard



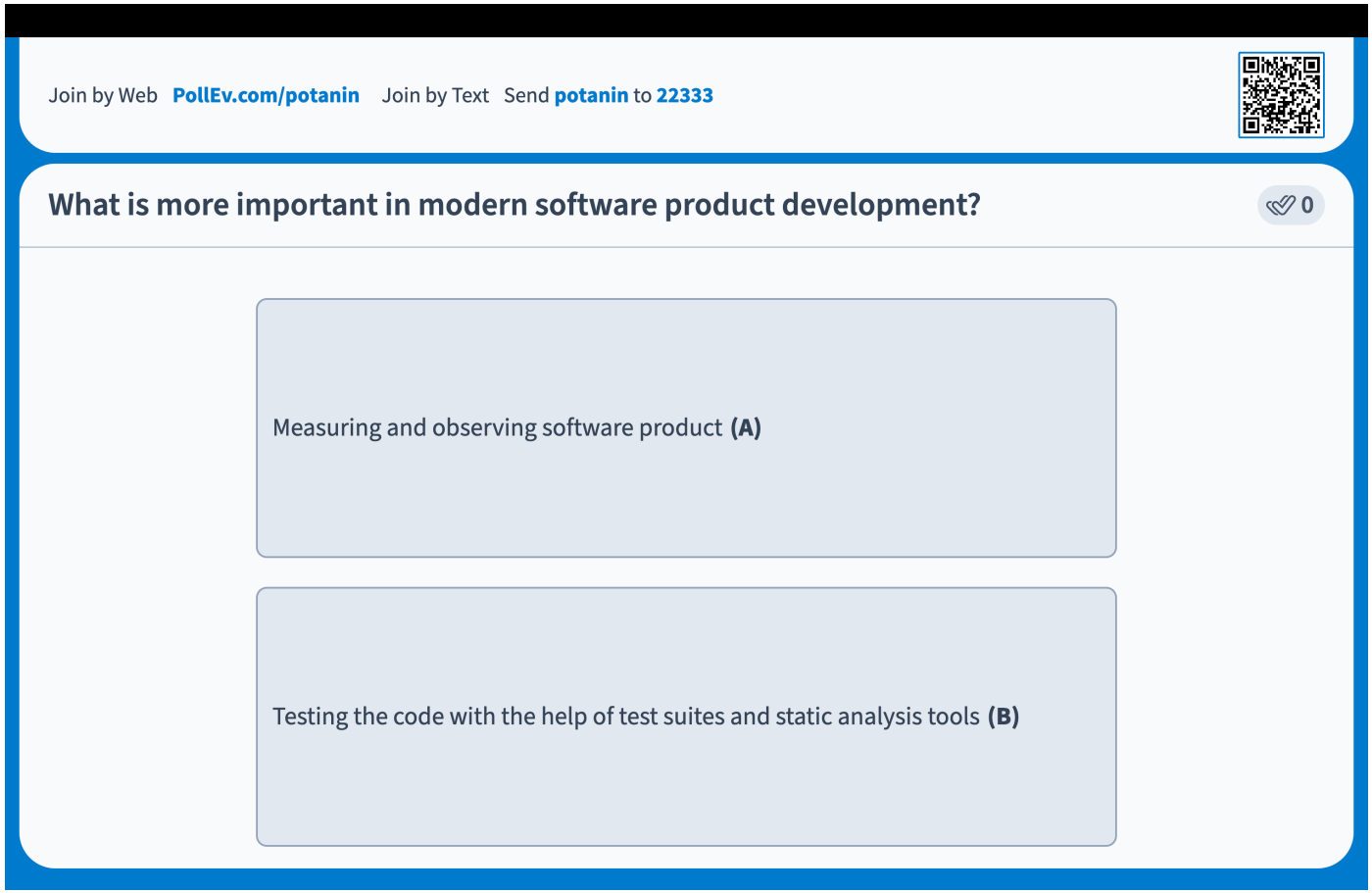

- Payal Chakravarty from IBM suggests a practical approach to DevOps measurement based around a metrics scorecard with 9 metrics:
  - These are relevant to software that is delivered as a cloud service. They include process metrics and service metrics
  - For the process metrics, you would like to see decreases in the number of failed deployments, the mean time to recovery after a service failure and the lead time from development to deployment.
  - You would hope to see increases in the deployment frequency and the number of lines of changed code that are shipped.
  - For the service metrics, availability and performance should be stable or improving, the number of customer complaints should be decreasing, and the number of new customers should be increasing.

# Metrics Trends (via Logging or Simila

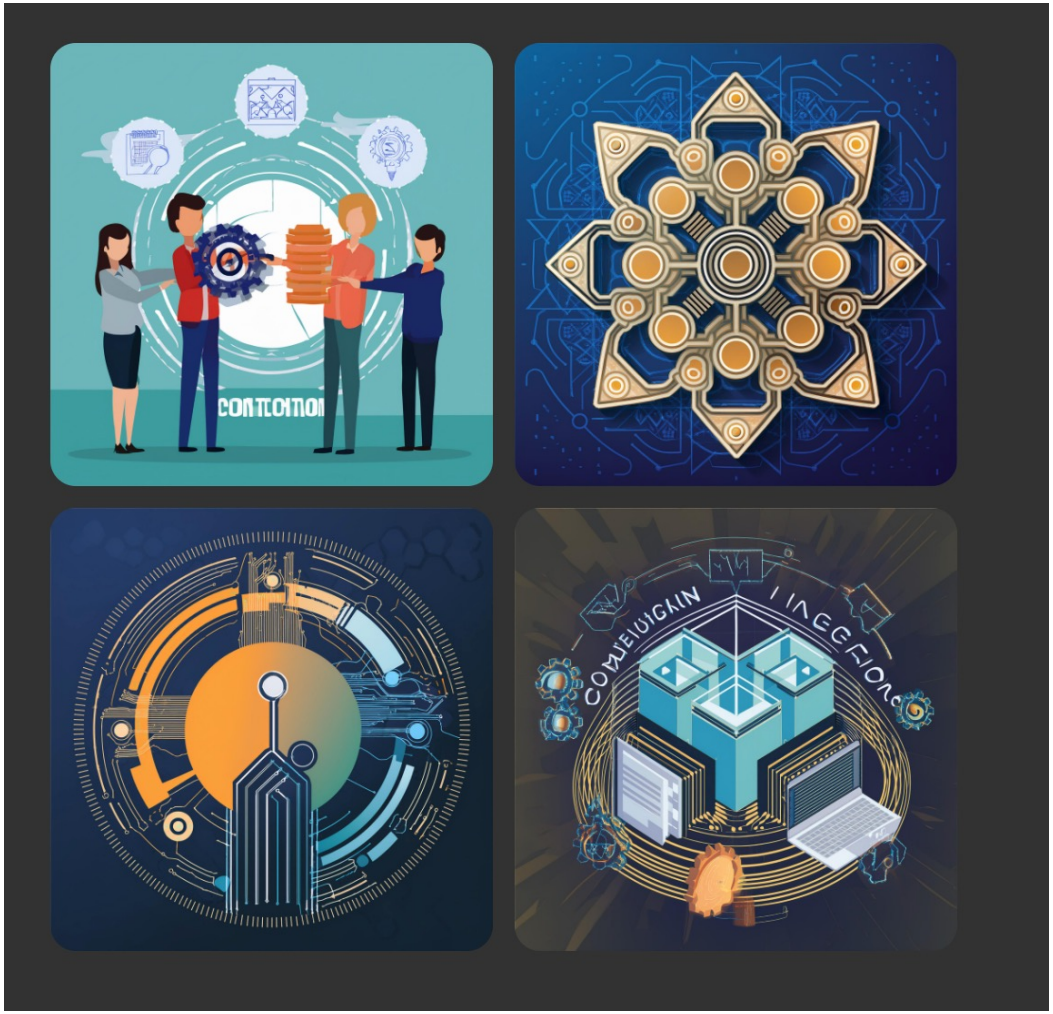



# Poll Everywhere Time!



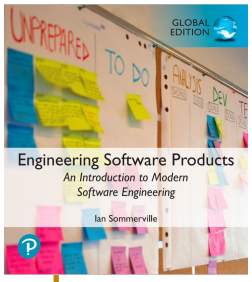
# CI: Continuous Integration



# DevOps automation


- By using DevOps with automated support, you can dramatically reduce the time and costs for integration, deployment and delivery.
- *Everything that can be, should be automated* is a fundamental principle of DevOps.
- As well as reducing the costs and time required for integration, deployment and delivery, process automation also makes these processes more reliable and reproducible.
- Automation information is encoded in scripts and system models that can be checked, reviewed, versioned and stored in the project repository.

# Aspects of DevOps automation




- ***Continuous integration***
  Each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.
- ***Continuous delivery***
  A simulation of the product's operating environment is created and the executable software version is tested.
- ***Continuous deployment***
  A new release of the system is made available to users every time a change is made to the master branch of the software.
- ***Infrastructure as code***
  Machine-readable models of the infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software's execution platform. The software to be installed, such as compilers and libraries and a DBMS, are included in the infrastructure model.

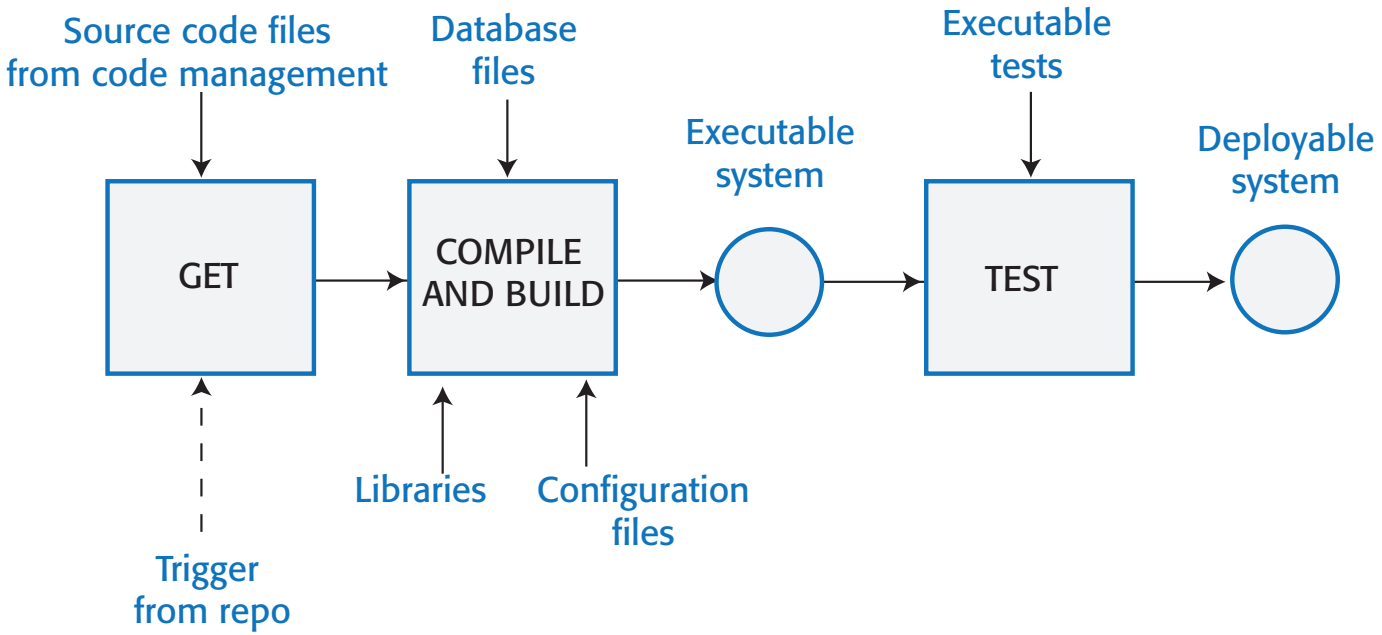
# System integration


- System integration (system building) is the process of gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system.
- Typical activities that are part of the system integration process include:
  - Installing database software and setting up the database with the appropriate schema.
  - Loading test data into the database.
  - Compiling the files that make up the product.
  - Linking the compiled code with the libraries and other components used.
  - Checking that external services used are operational.
  - Deleting old configuration files and moving configuration files to the correct locations.
  - Running a set of system tests to check that the integration has been successful.

# Continuous Integration (CI)


- Continuous integration simply means that an integrated version of the system is created and tested every time a change is pushed to the system's shared repository.
- On completion of the push operation, the repository sends a message to an integration server to build a new version of the product
- The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system.
- If you make a small change and some system tests then fail, the problem almost certainly lies in the new code that you have pushed to the project repo.
- You can focus on this code to find the bug that's causing the problem.

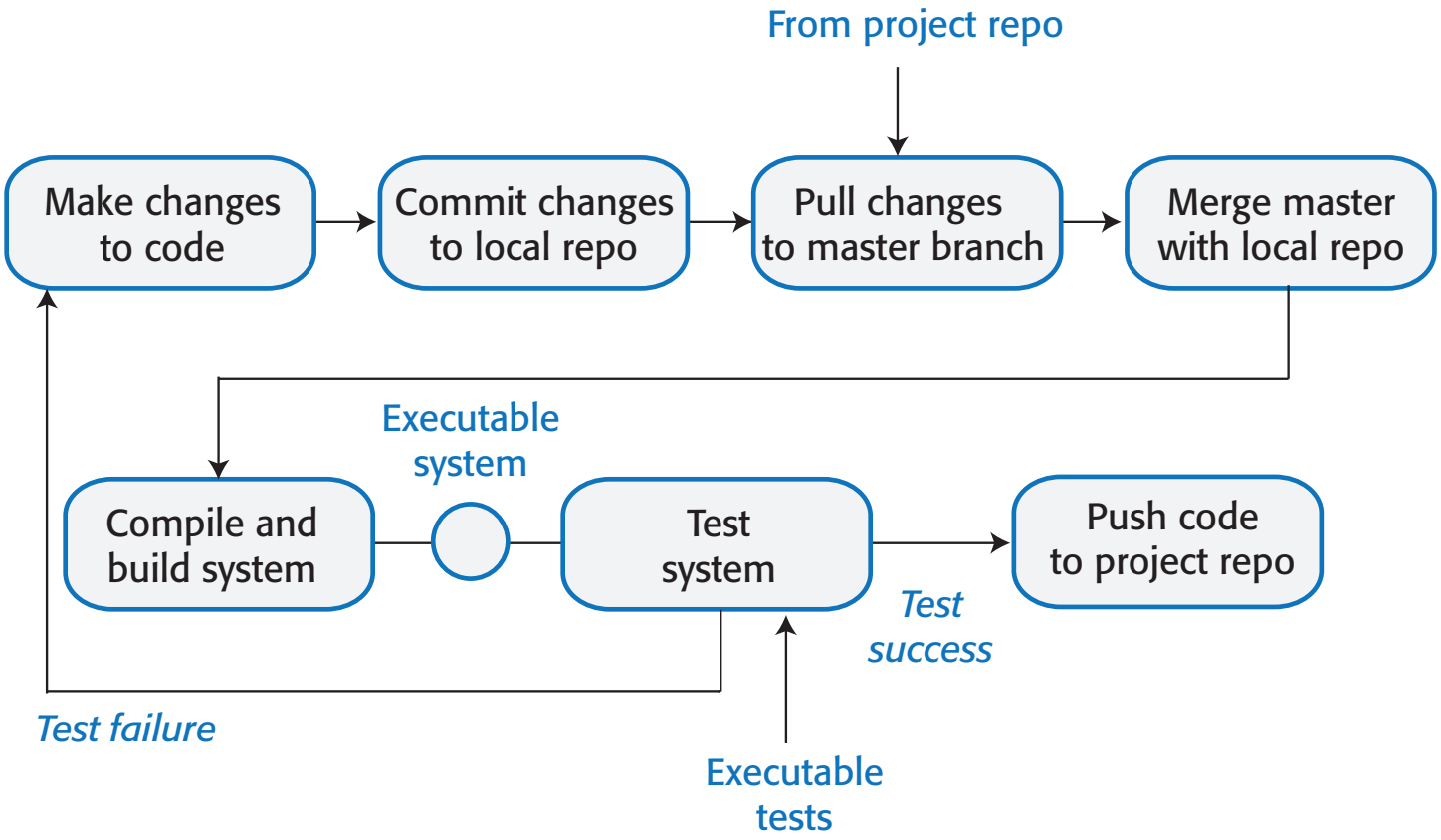
# Continuous Integration (CI)

# Breaking the build


- In a continuous integration environment, developers have to make sure that they don't 'break the build'.
- Breaking the build means pushing code to the project repository which, when integrated, causes some of the system tests to fail.
- If this happens to you, your priority should be to discover and fix the problem so that normal development can continue.
- To avoid breaking the build, you should always adopt an 'integrate twice' approach to system integration.
  - You should integrate and test on your own computer before pushing code to the project repository to trigger the integration server

# Local integration

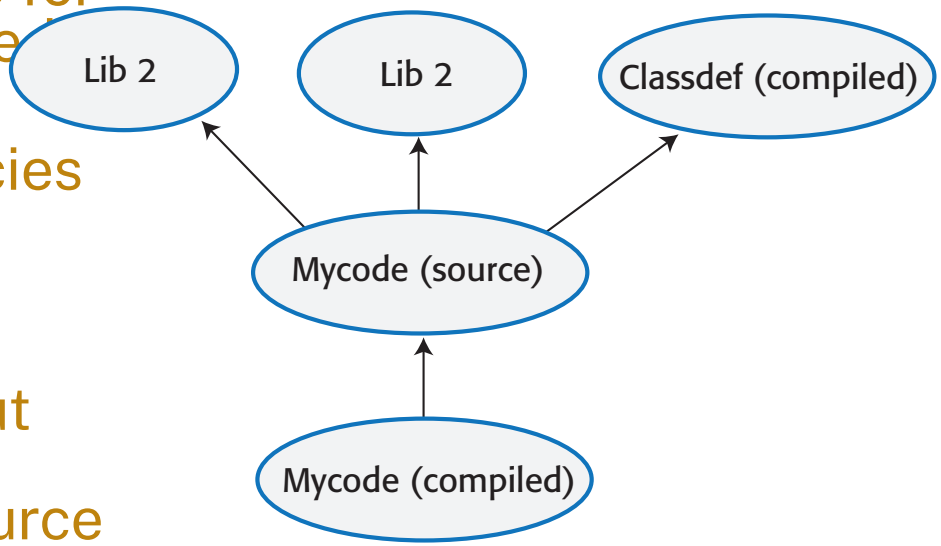



# System building

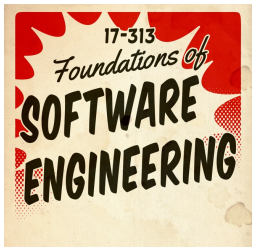

- Continuous integration is only effective if the integration process is fast and developers do not have to wait for the results of their tests of the integrated system.
- However, some activities in the build process, such as populating a database or compiling hundreds of system files, are inherently slow.
- It is therefore essential to have an automated build process that minimizes the time spent on these activities.
- Fast system building is achieved using a process of incremental building, where only those parts of the system that have been changed are rebuilt

# Dependencies


- Running a set of system tests depends on the existence of executable object code for both the program being tested and the system tests.
- In turn, these depend on the source code for the system and the tests that are compiled to create the object code.
- Figure on the right shows the dependencies involved in creating the object code for a source code files called Mycode.
- An automated build system uses the specification of dependencies to work out what needs to be done. It uses the file modification timestamp to decide if a source code file has been changed.

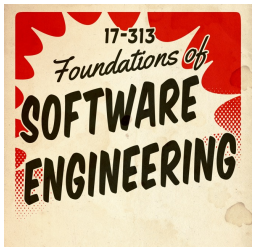


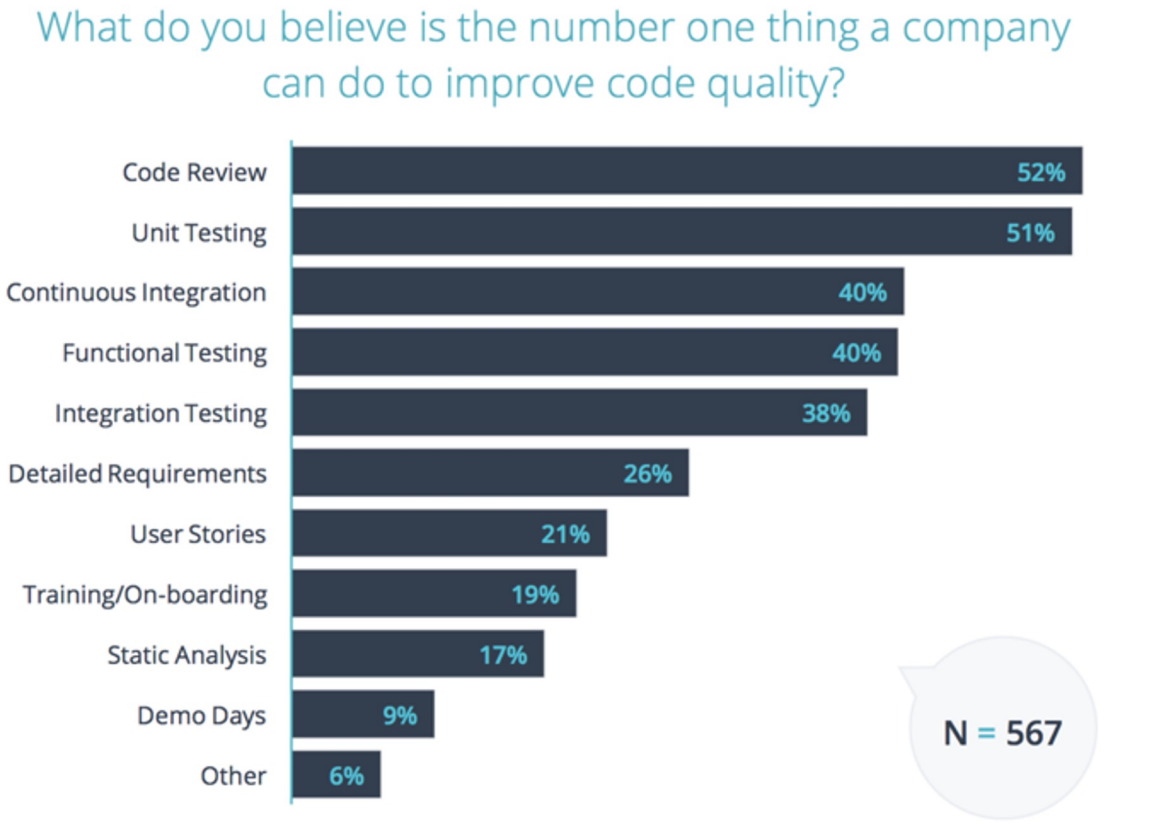
# Process Roundup



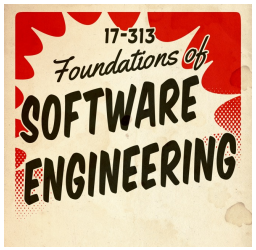

- Group Processes
- Continuous Integration
  - Code Review
  - Pair/Mob Programming
- Individual Processes
  - Asking Questions
  - How to run a meeting

# State of Code Review 2017



# History of CI





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


(2000) Martin Fowler posts “Continuous Integration” blog


(2001) First CI tool

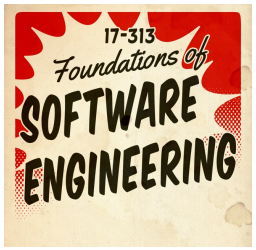

(2005) Hudson/Jenkins

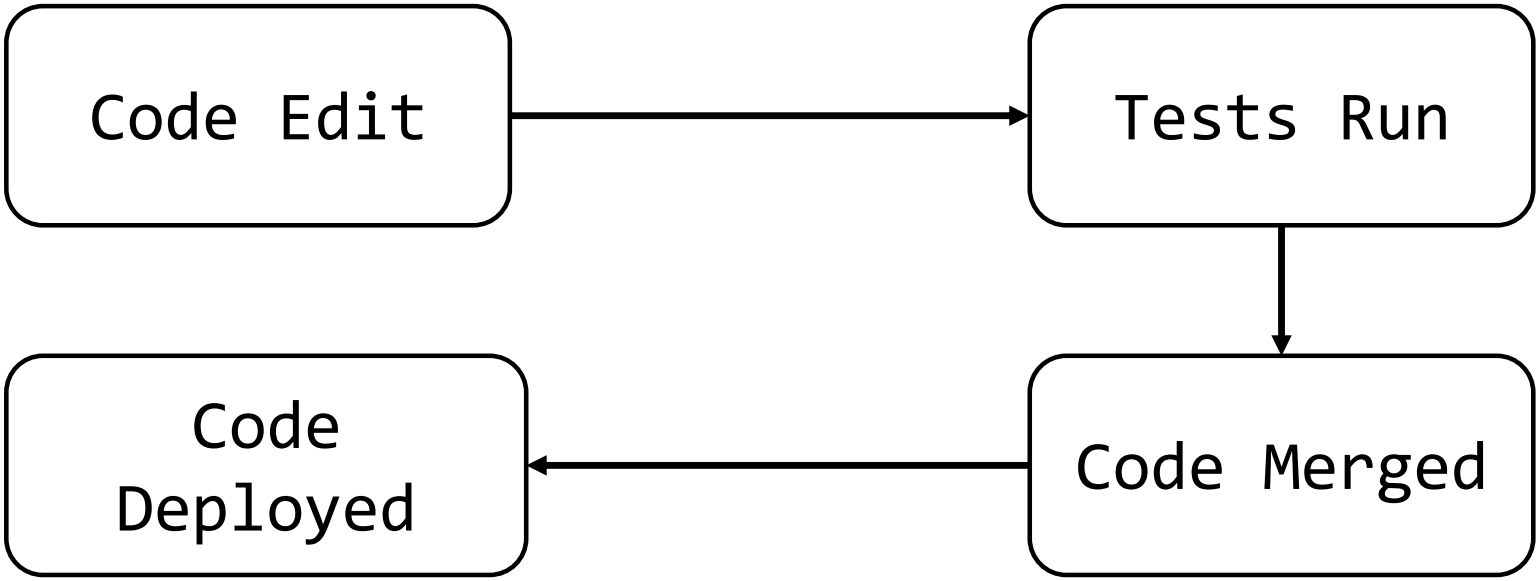

(2011) Travis CI


(2019) GitHub Actions



# CI/CD Pipeline overview

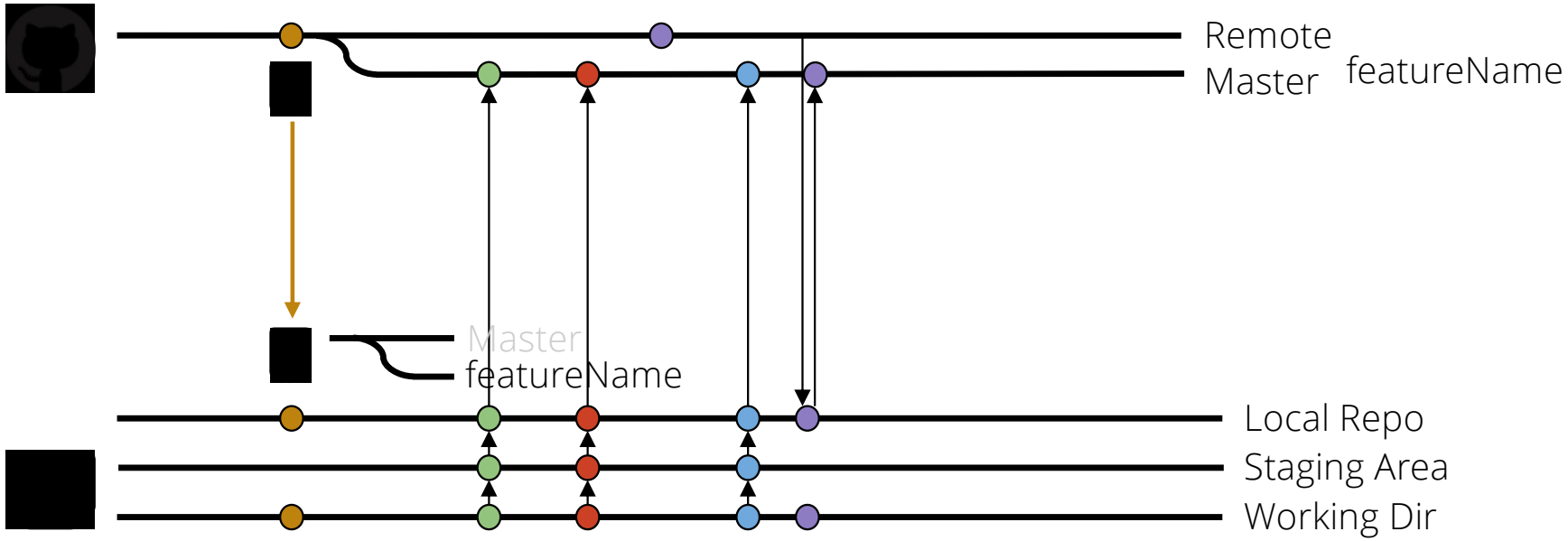





# Github Flow

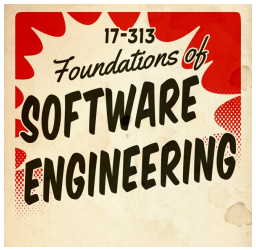
**Alternative:** Work with your PO to ensure that the user stories are as small and atomic as feasible. The workload should be small enough to be completed within a day or less. Try to make your CI pipeline lightweight/performant enough to allow commits to flow as often as needed. By landing commits to trunk often we facilitate the CI principles and keep the code integrated.




https://docs.github.com/en/get-started/quickstart/github-flow

https://medium.com/burdaforward/state-of-ci-cd-and-the-dreaded-git-flow-fce92d04fb07

# Sample CI Workflow




Create Pull Request

GitHub tells Travis CI build is mergeable

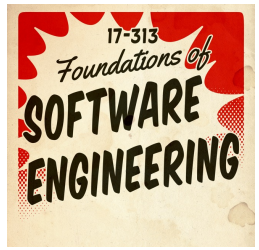
It builds and passes tests

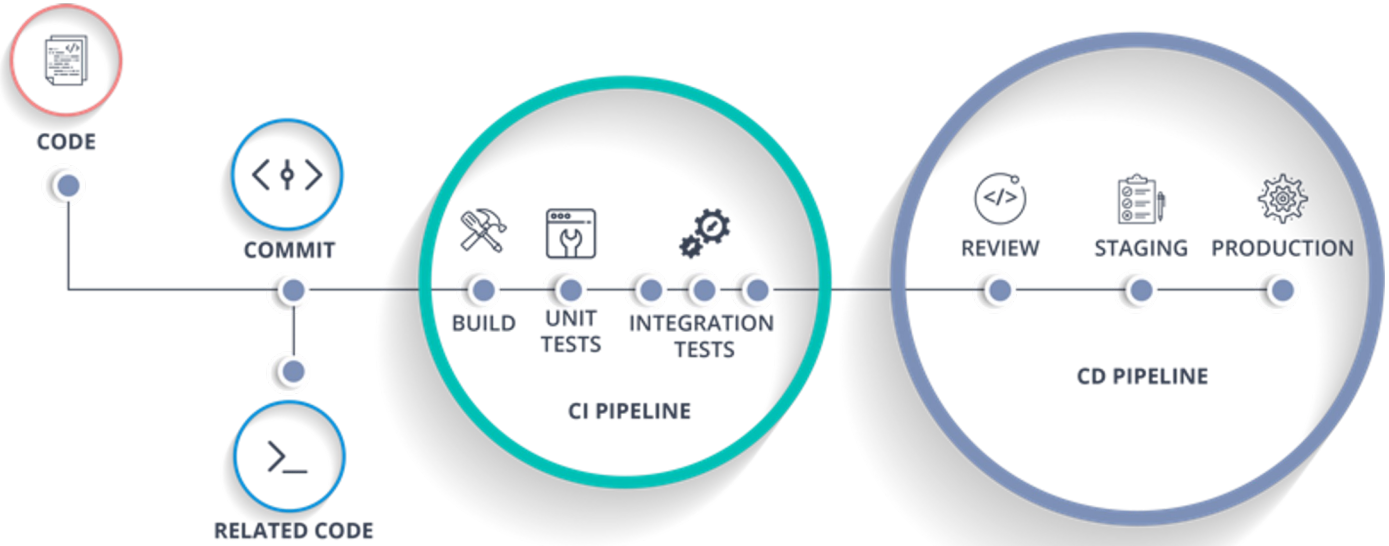

Travis updates PR

PR is merged

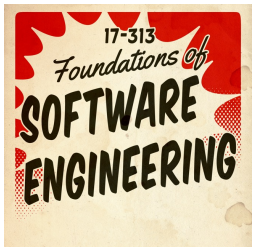
https://docs.github.com/en/actions/migrating-to-github-actions/migrating-from-travis-ci-to-github-actions

# Example CI/CD Pipeline

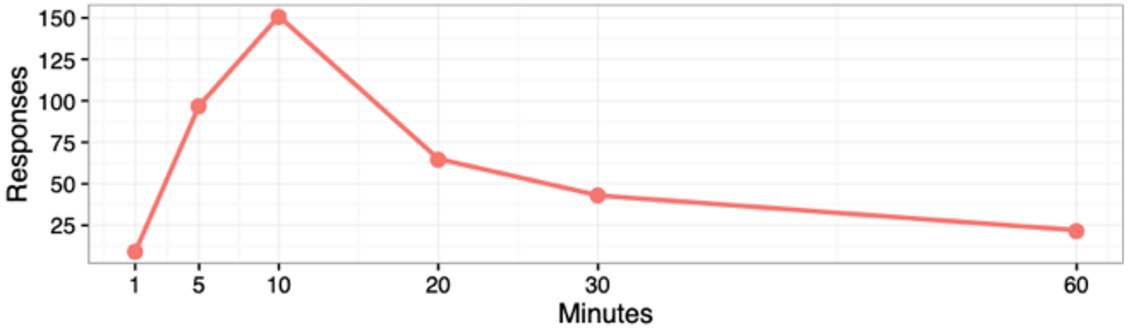


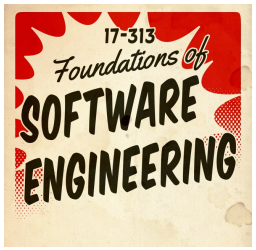
# CI Research



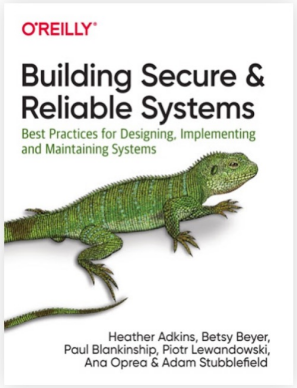

"My favorite way of thinking about build time is basically, you have tea time, lunch time, or bedtime..."

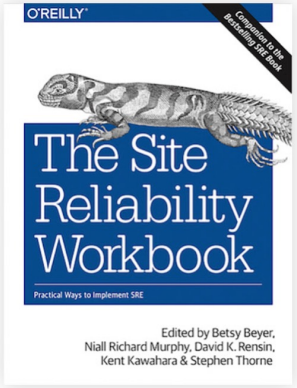


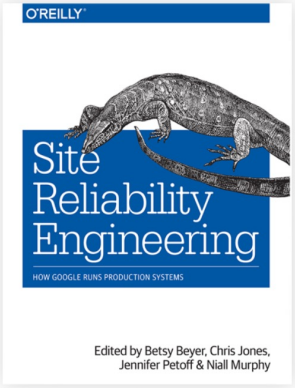
# DevOps: More Resources

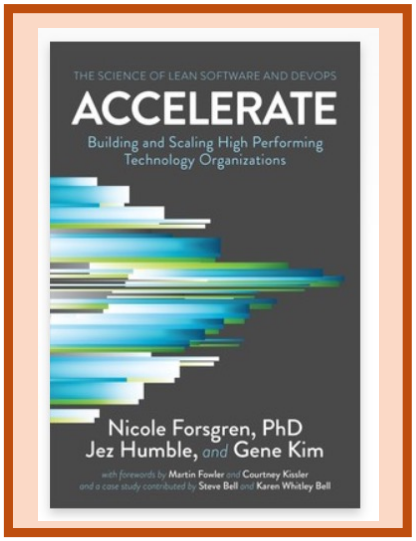



SRE Books

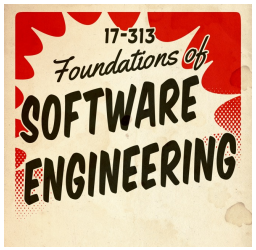


# Developers say:

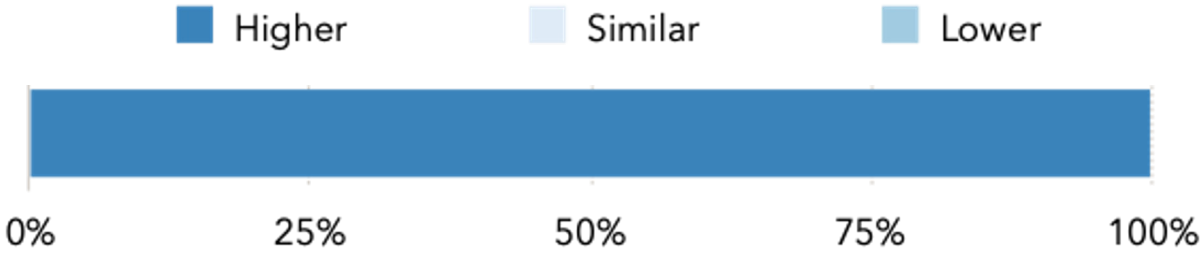



CI helps us catch bugs earlier
CI makes us less worried about breaking our builds
CI lets us spend less time debugging

"[CI] does have a pretty big impact on [catching bugs]. It allows us to find issues even before they get into our main repo, ... rather than letting bugs go unnoticed, for months, and letting users catch them."

# Developers report:

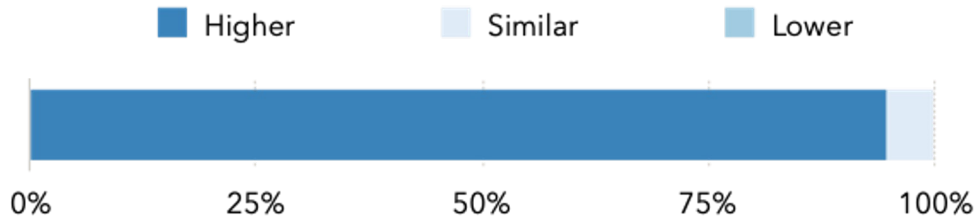

Do developers on projects with CI give (more/similar/less) value to automated tests?



# Developers report:


Do developers on projects with CI give (more/similar/less) value to automated tests?

Do projects with CI have (higher/similar/lower) test quality?

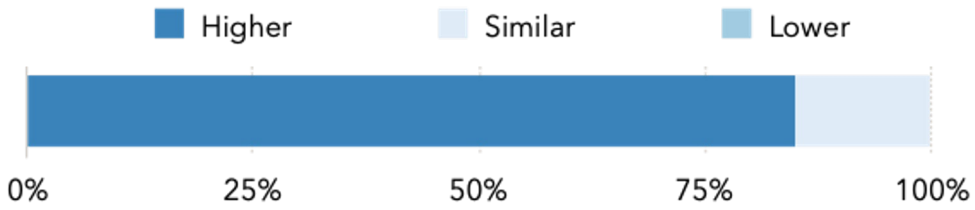


# Developers report:


Do developers on projects with CI give (more/similar/less) value to automated tests?

Do projects with CI have (higher/similar/lower) test quality?

**Do projects with CI have (higher/similar/lower) code quality?**



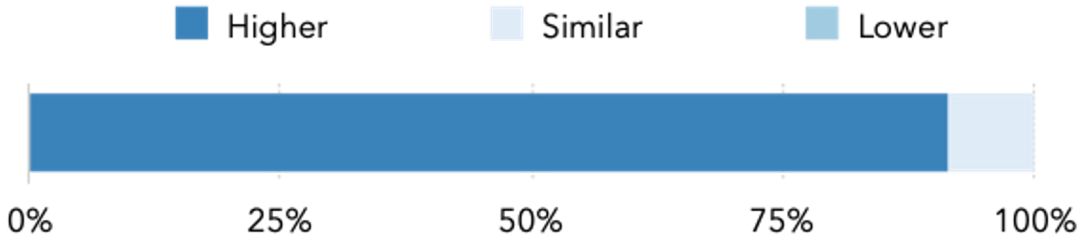
# Developers report:


Do developers on projects with CI give (more/similar/less) value to automated tests?

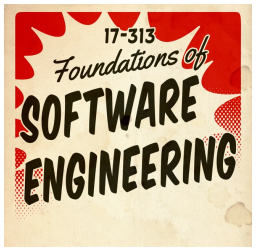
Do projects with CI have (higher/similar/lower) test quality?

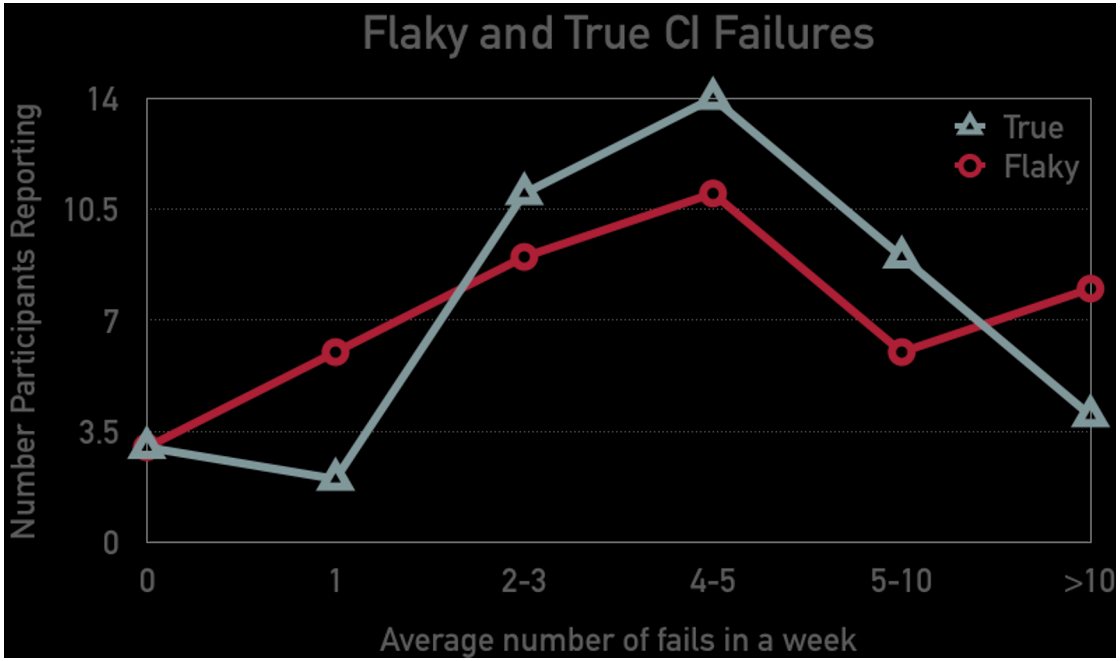
Do projects with CI have (higher/similar/lower) code quality?

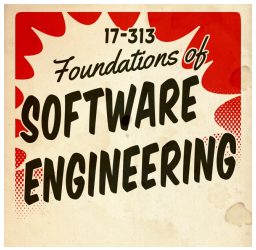
Are developers on projects with CI (more/similar/less) productive?



# Challenge: Flaky Tests



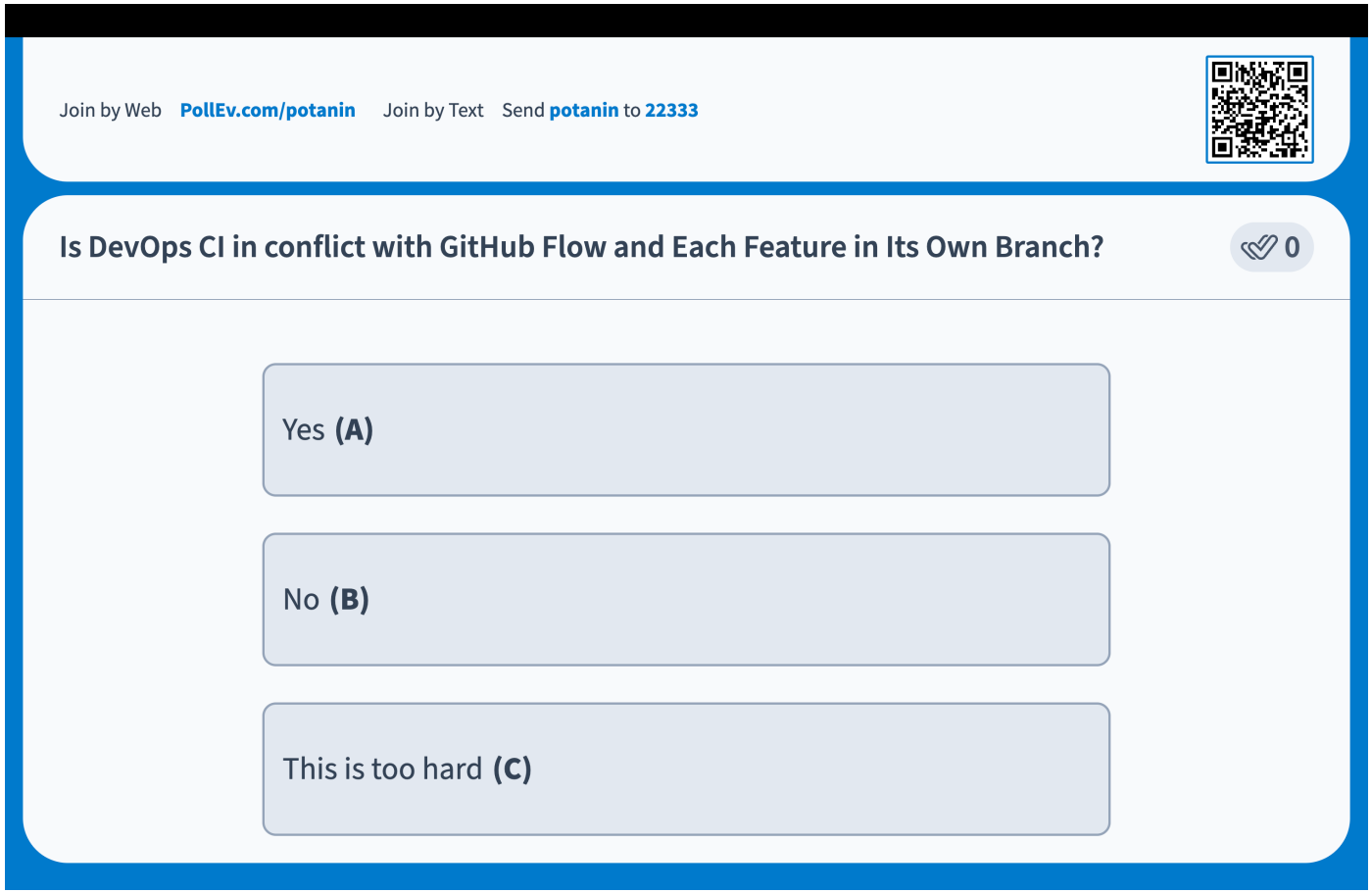






# Most of the benefits of CI come from running tests

# OBSERVATION

# Poll Everywhere Time!

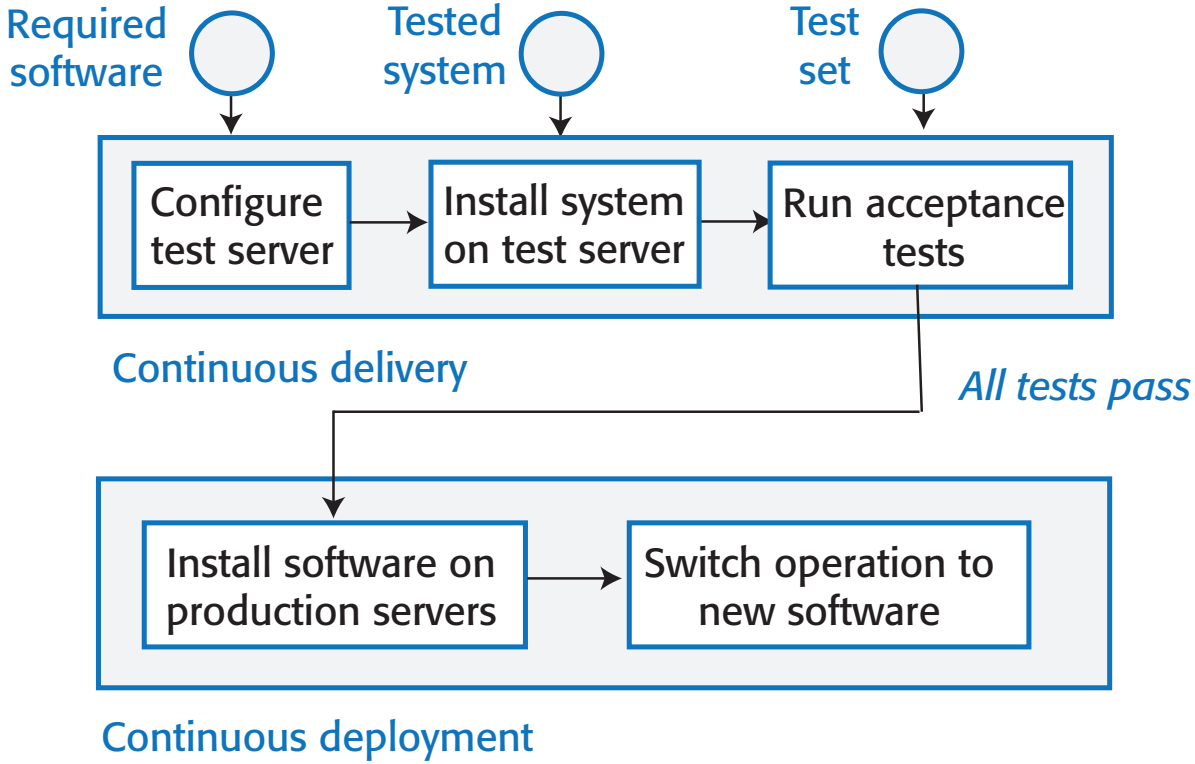


# CD: Continuous Deployment

# Continuous Delivery and Deployment (CD)


- Continuous integration means creating an executable version of a software system whenever a change is made to the repository. The CI tool builds the system and runs tests on your development computer or project integration server.
- However, the real environment in which software runs will inevitably be different from your development system.
- When your software runs in its real, operational environment bugs may be revealed that did not show up in the test environment.
- *Continuous delivery means that, after making changes to a system, you ensure that the changed system is ready for delivery to customers.*
- This means that you have to test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance.

# Continuous Delivery and Deployment (CD)

# The deployment pipeline


- After initial integration testing, a staged test environment is created.
- This is a replica of the actual production environment in which the system will run.
- The system acceptance tests, which include functionality, load and performance tests, are then run to check that the software works as expected. If all of these tests pass, the changed software is installed on the production servers.
- To deploy the system, you then momentarily stop all new requests for service and leave the older version to process the outstanding transactions.
- Once these have been completed, you switch to the new version of the system and restart processing.

# Benefits of Continuous Deployment (CD)

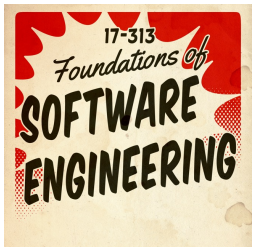

- ***Reduced costs***
  If you use continuous deployment, you have no option but to invest in a completely automated deployment pipeline. Manual deployment is a time-consuming and error-prone process. Setting up an automated system is expensive and time-consuming but you can recover these costs quickly if you make regular updates to your product.

- ***Faster problem solving***
  If a problem occurs, it will probably only affect a small part of the system and it will be obvious what the source of that problem is. If you bundle many changes into a single release, finding and fixing problems is more difficult.

- ***Faster customer feedback***
  You can deploy new features when they are ready for customer use. You can ask them for feedback on these features and use this feedback to identify improvements that you need to make.

- ***A/B Testing and Canary Deployments***
  This is an option if you have a large customer base and use several servers for deployment. You can deploy a new version of the software on some servers and leave the older version running on others. You then use the load balancer to divert some customers to the new version while others use the older version. You can then measure and assess how new features are used to see if they do what you expect.

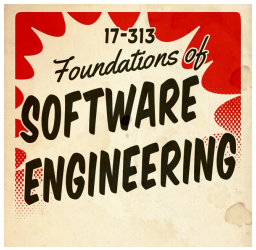
# Nightly Build




- Build code and run smoke test (Microsoft 1995)

- Benefits
  - it minimizes integration risk
  - It reduces the risk of low quality
  - it supports easier defect diagnosis
  - it improves morale

# Ring Deployment: Microsoft



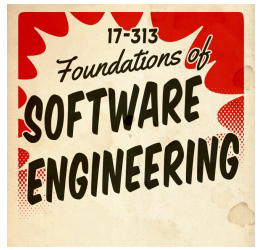

- Commits flow out to rings, de-flight if issue
- For example:
  - Ring 0 => Team
  - Ring 1 => Dogfood
  - Ring 2 => Beta
  - Ring 3 => Many
  - Ring 4 => All
- Windows 10 Insiders Program
  - Dev Channel (weekly builds of Windows 10)
  - Beta Channel (dev + validated updates by Microsoft)
  - Release Preview Channel (highest quality, validated updates)

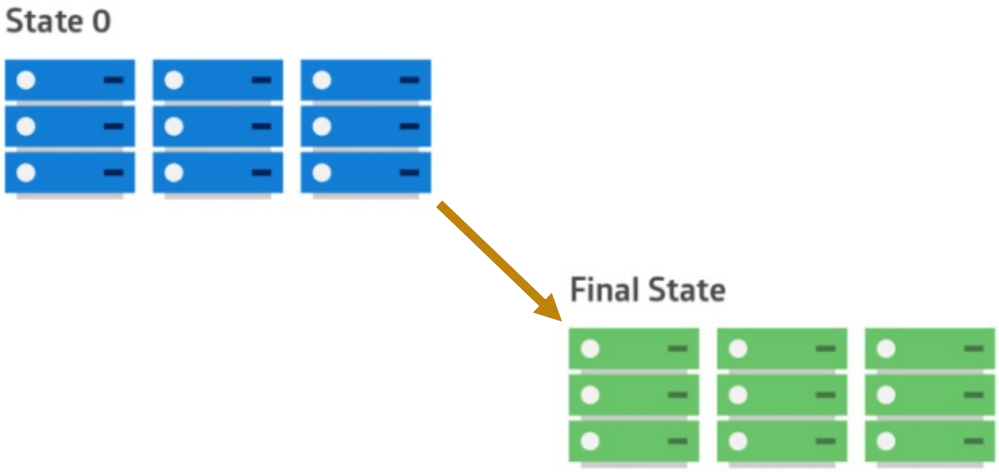
# Rapid Release/Mozilla


*If deployment requires on-prem deployment, say a web browser*

- There are four channels: *Nightly*, Alpha, Beta, Release Candidate
- Code flows every 2 weeks to next channel, unless fast tracked by release engineer.
- Involve corporate customer specific testing in testing (Practice also used by IBM, Redhat)
- same for Windows Edge browser Insiders Program:
  - Canary: nightly builds
  - Dev: weekly builds
  - Beta: 6 weeks

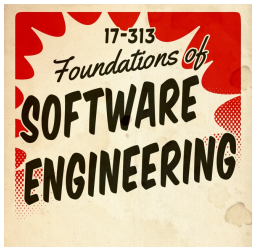
# “Big bang” deployments




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

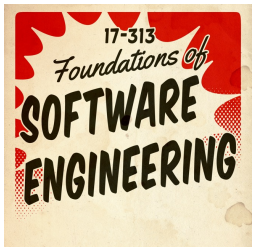
# Fast to Deploy, Slow to Release




Chuck Rossi at Facebook: *"Get your s*** in, fix it in production"*

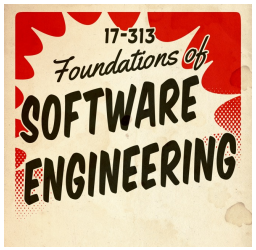

https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/

# Dark Launches at Instagram

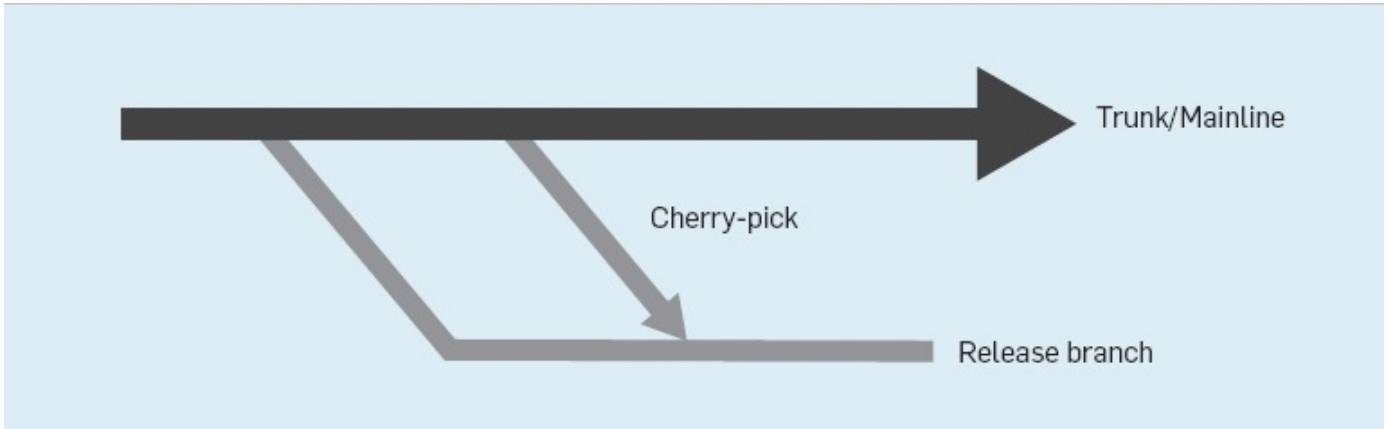



- **Early**: Integrate as soon as possible. Find bugs early. Code can run in production about 6 months before being publicly announced.
- **Often**: Reduce friction. Try things out. See what works. Push small changes just to gather metrics, feasibility testing. Large changes just slow down the team.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

# Facebook process (until 2016)

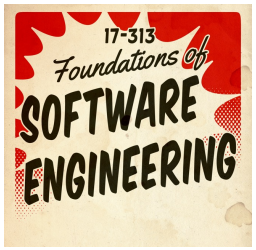


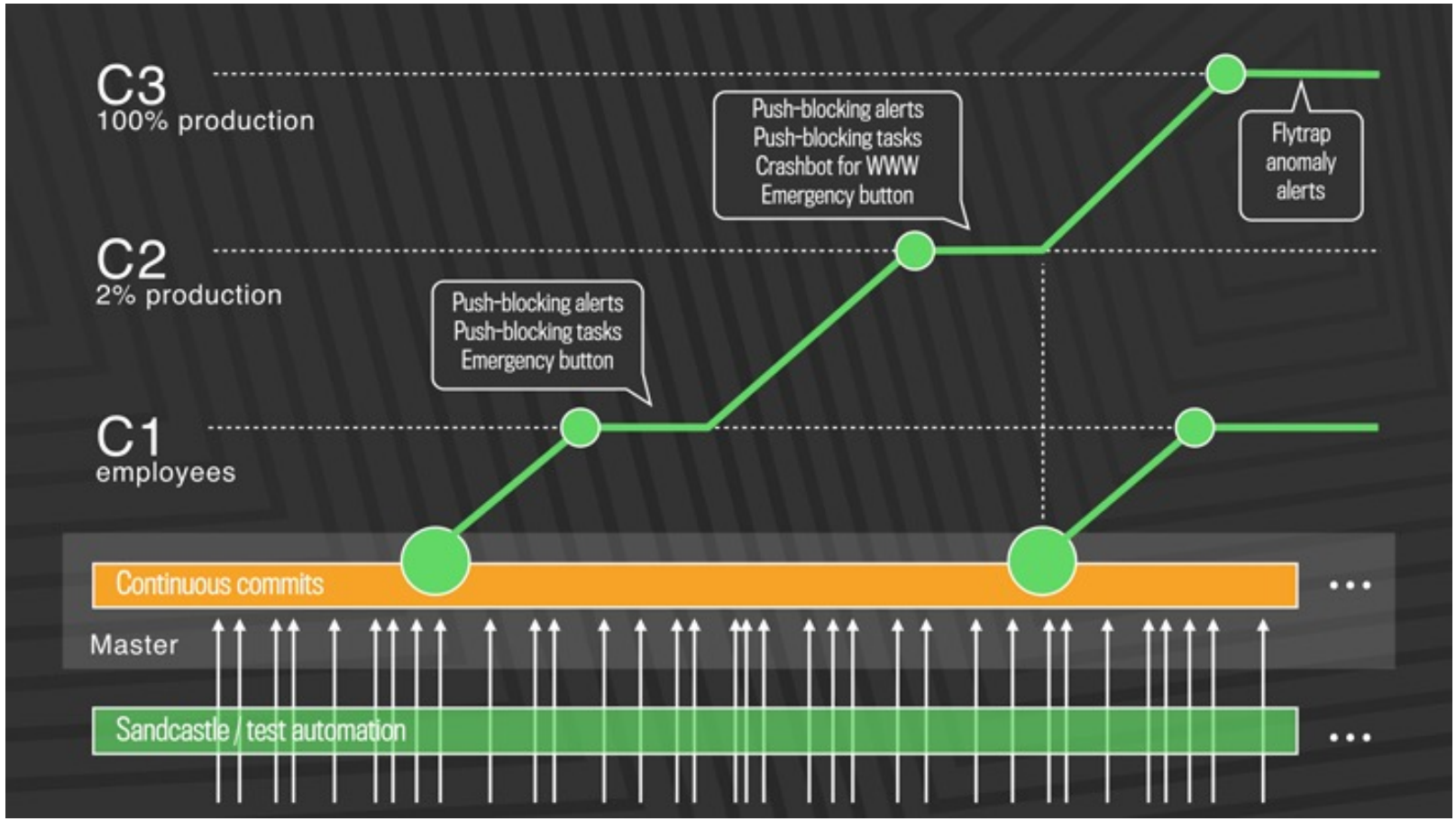

- Release is cut Sunday 6pm
- Stabilize until Tuesday, canaries, release. Tuesday push is 12,000 diffs.
- Cherry pick: Push 3 times a day (Wed-Fri) 300-700 cherry picks / day.



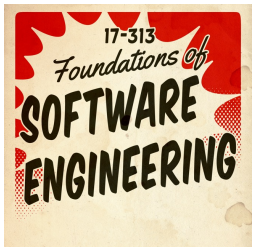

reference: https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/

# Facebook quasi-continuous release



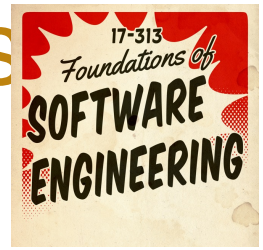
# Rolling deployments

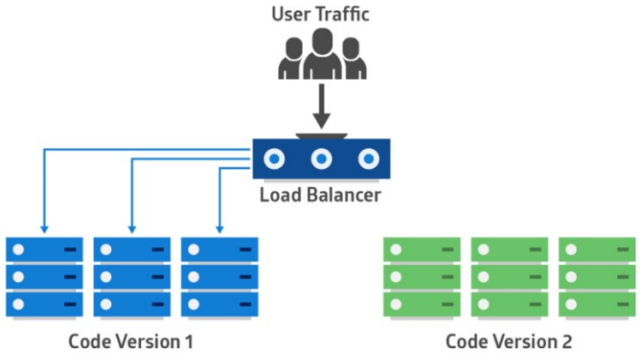


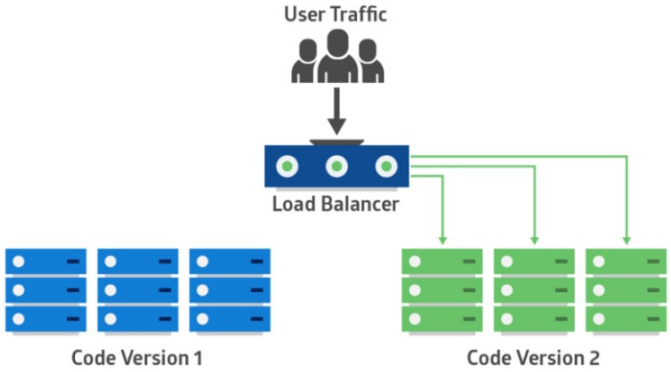


reference: https://dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3

# Red/Black (Blue/Green) deployments

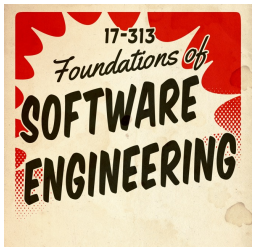


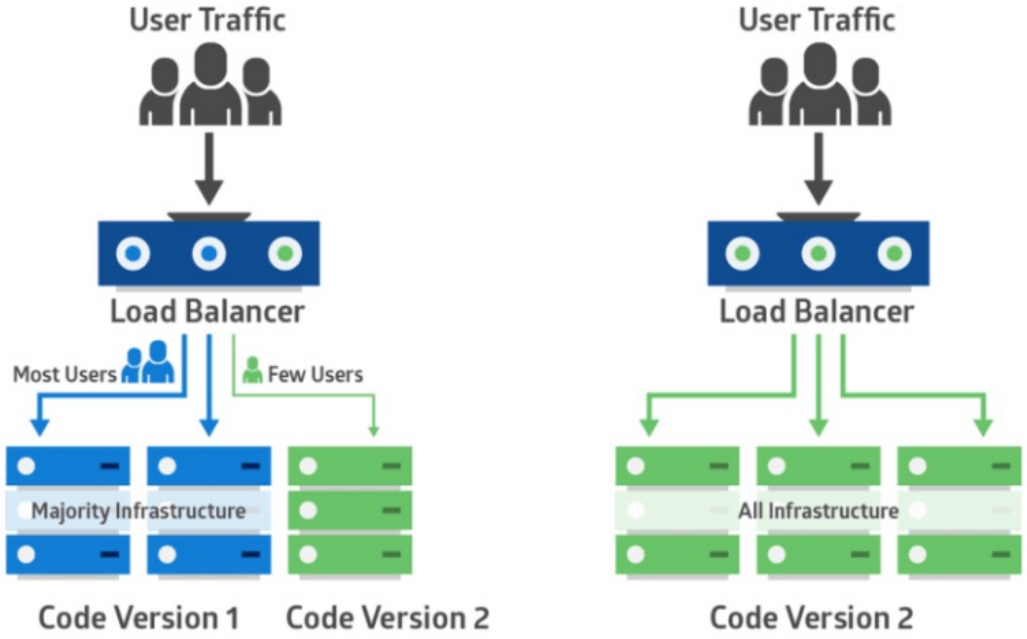




reference: https://dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3

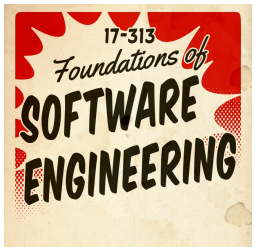
# Canary deployments

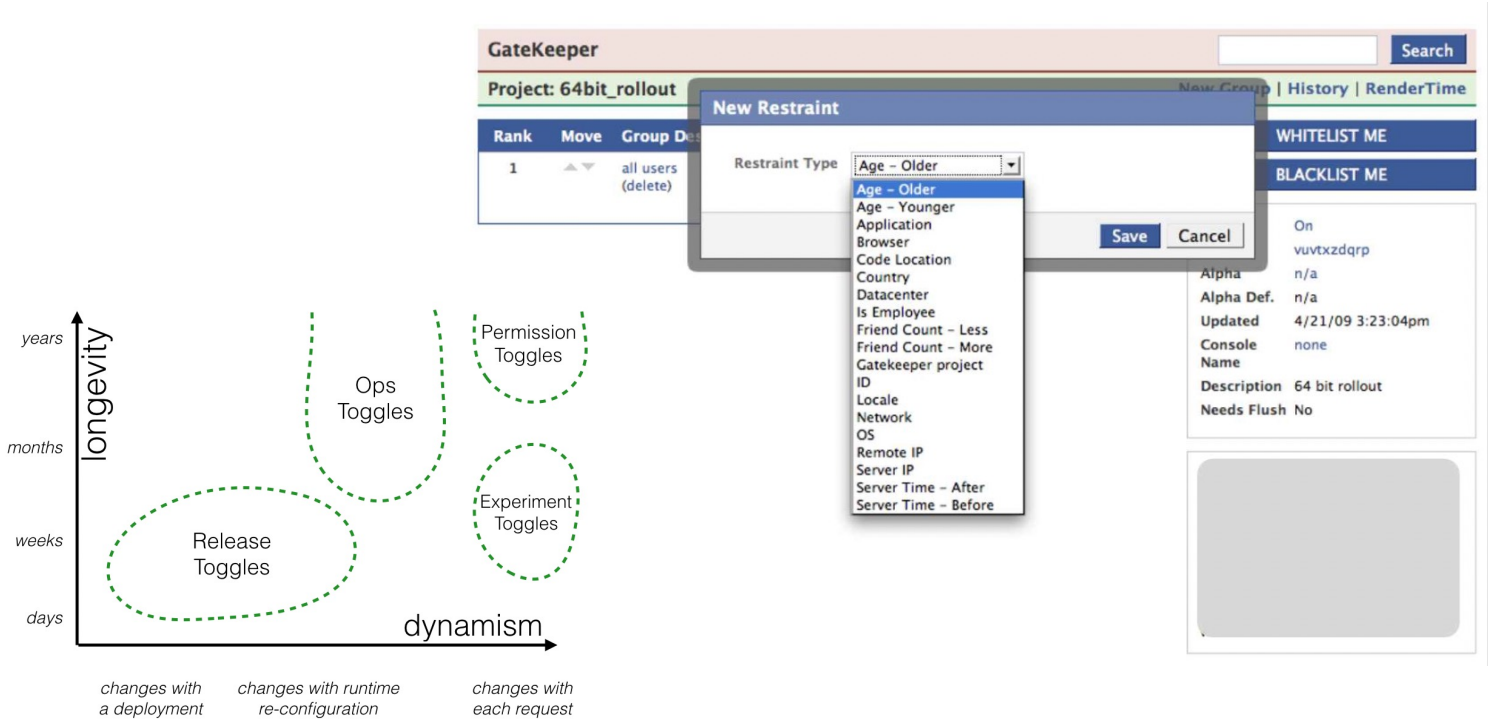


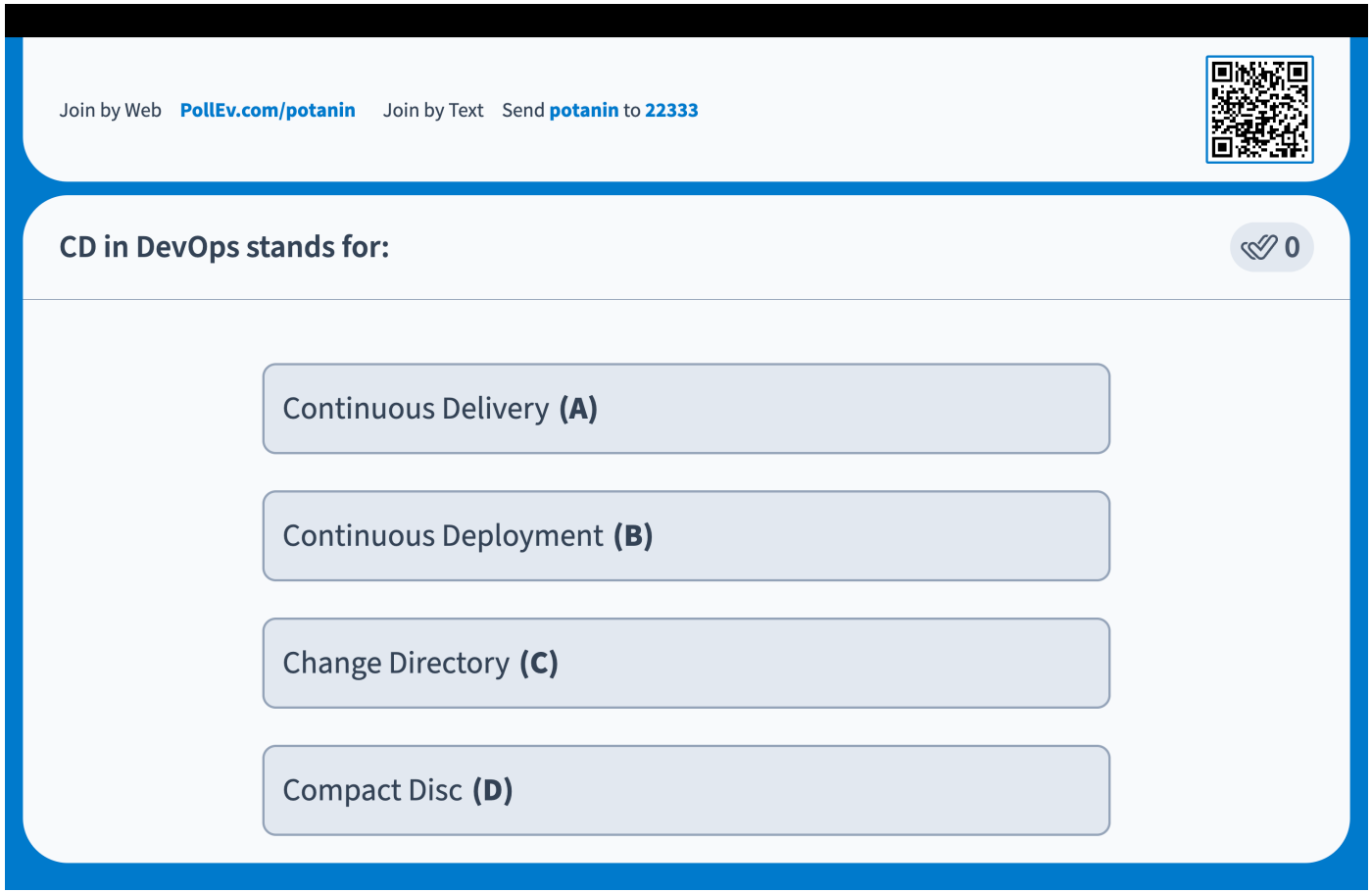

https://dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3

# Feature Flags



# Poll Everywhere Time!

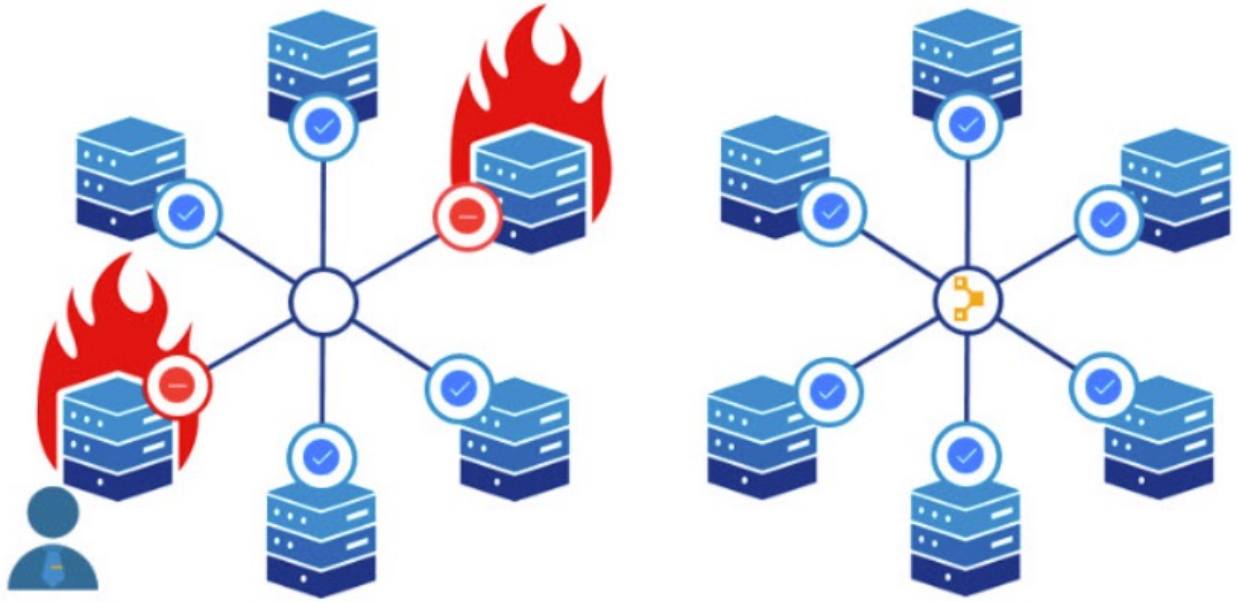


# Infrastructure as Code

# Infrastructure as Code

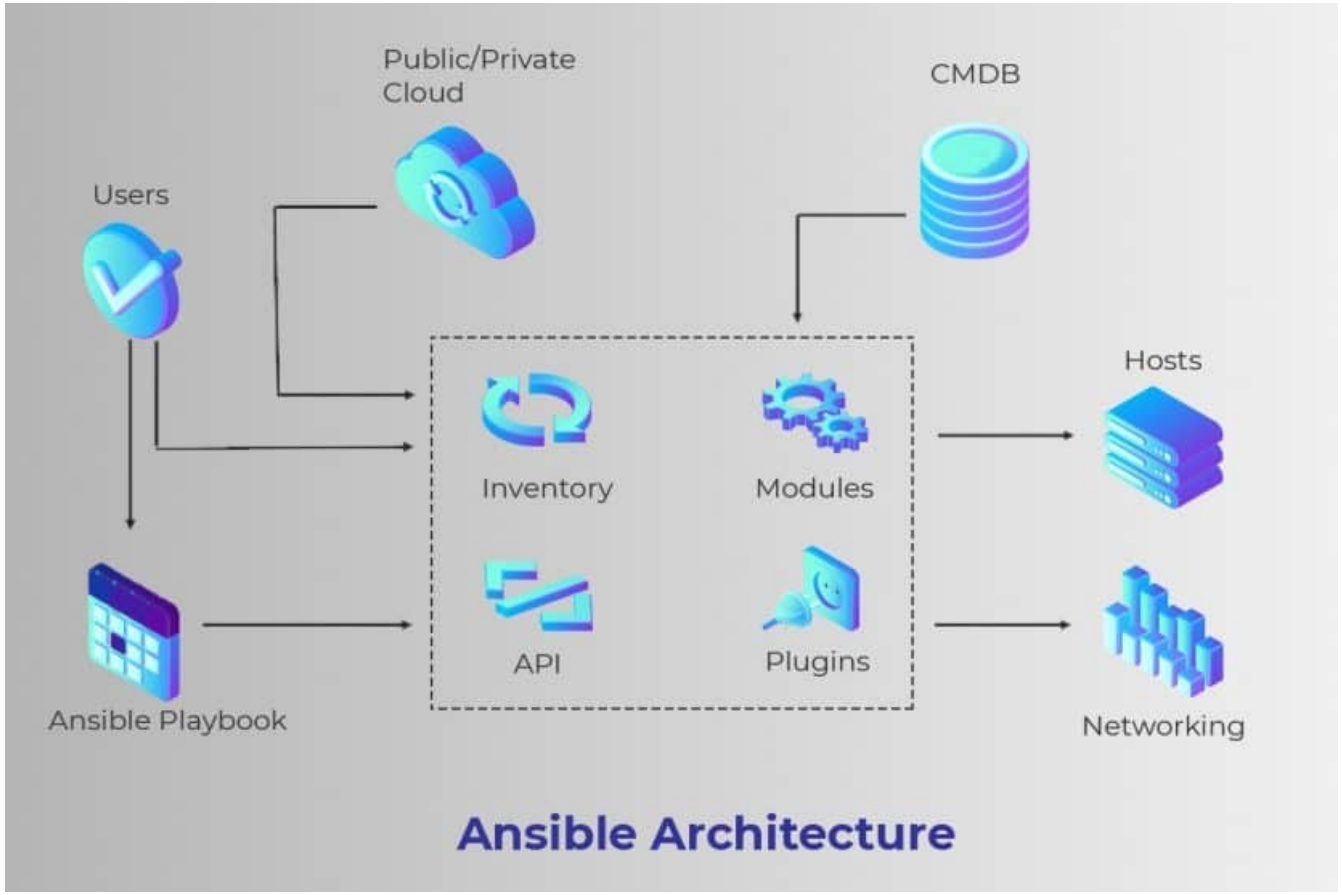

- In an enterprise environment, there are usually many different physical or virtual servers (web servers, database servers, file servers, etc.) that do different things. These have different configurations and run different software packages.
- It is therefore difficult to keep track of the software installed on each machine.
- The idea of infrastructure as code was proposed as a way to address this problem. Rather than manually updating the software on a company's servers, the process can be automated using a model of the infrastructure written in a machine-processable language.
- Configuration Management (CM) tools such as Puppet can automatically install software and services on servers according to the infrastructure definition

# Infrastructure as Code (Puppet)



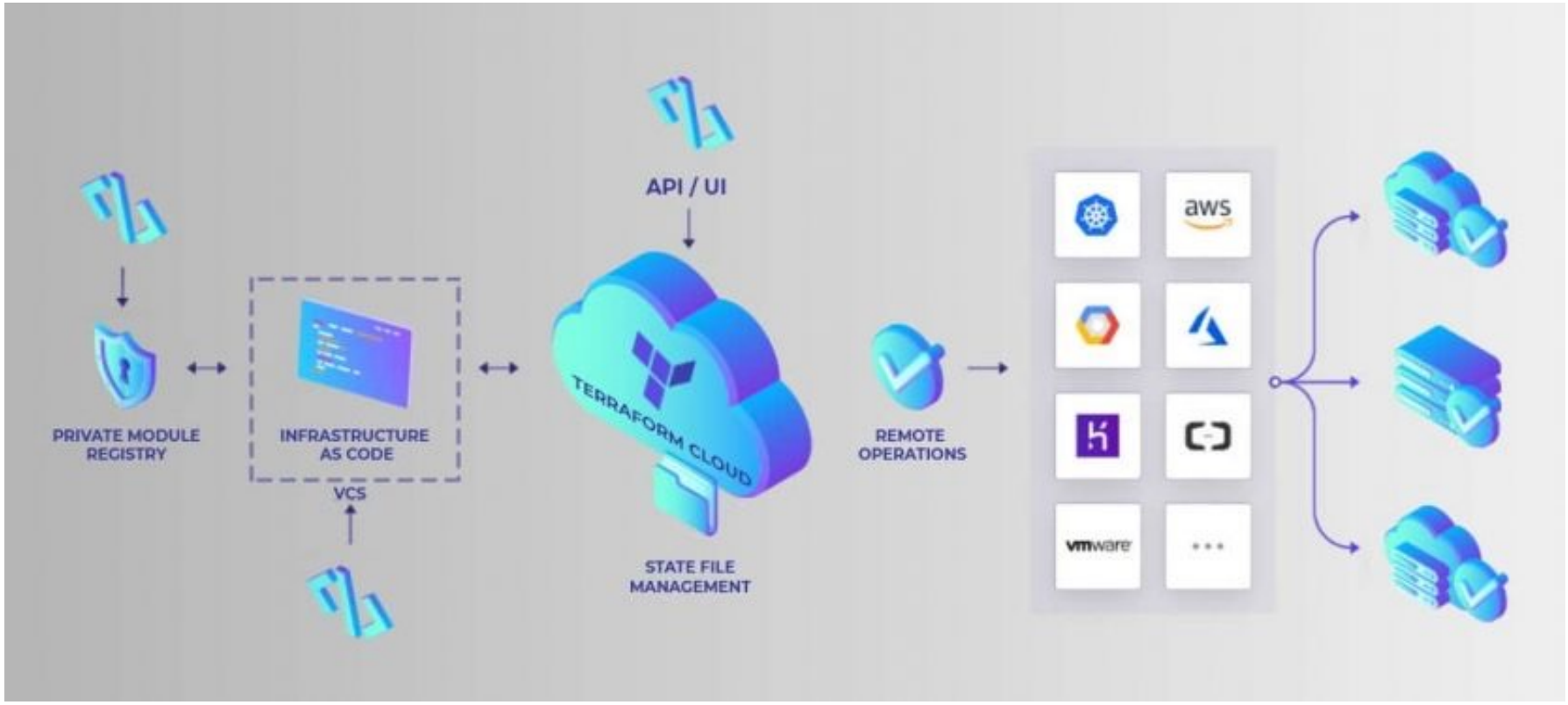
https://phoenixnap.com/blog/what-is-puppet

# Infrastructure as Code (Ansible)




https://phoenixnap.com/blog/ansible-vs-terraform-vs-puppet

# Infrastructure as Code (Terraform)




https://phoenixnap.com/blog/ansible-vs-terraform-vs-puppet

# Benefits of infrastructure as code


- Defining your infrastructure as code and using a configurat management system solves two key problems of continuous deployment.
  - Your testing environment must be exactly the same as your deployment environment. If you change the deployment environment, you have to mirror those changes in your testing environment.
  - When you change a service, you have to be able to roll that change out to all of your servers quickly and reliably. If there is a bug in your changed code that affects the system's reliability, you have to be able to seamlessly roll back to the older system.
- The business benefits of defining your infrastructure as code are lower costs of system management and lower risks of unexpected problems arising when infrastructure changes are implemented.

# Characteristics of infrastructure as code


- ***Visibility***
  Your infrastructure is defined as a stand-alone model that can be read, discussed, understood and reviewed by the whole DevOps team.

- ***Reproducibility***
  Using a configuration management tool means that the installation tasks will always be run in the same sequence so that the same environment is always created. You are not reliant on people remembering the order that they need to do things.

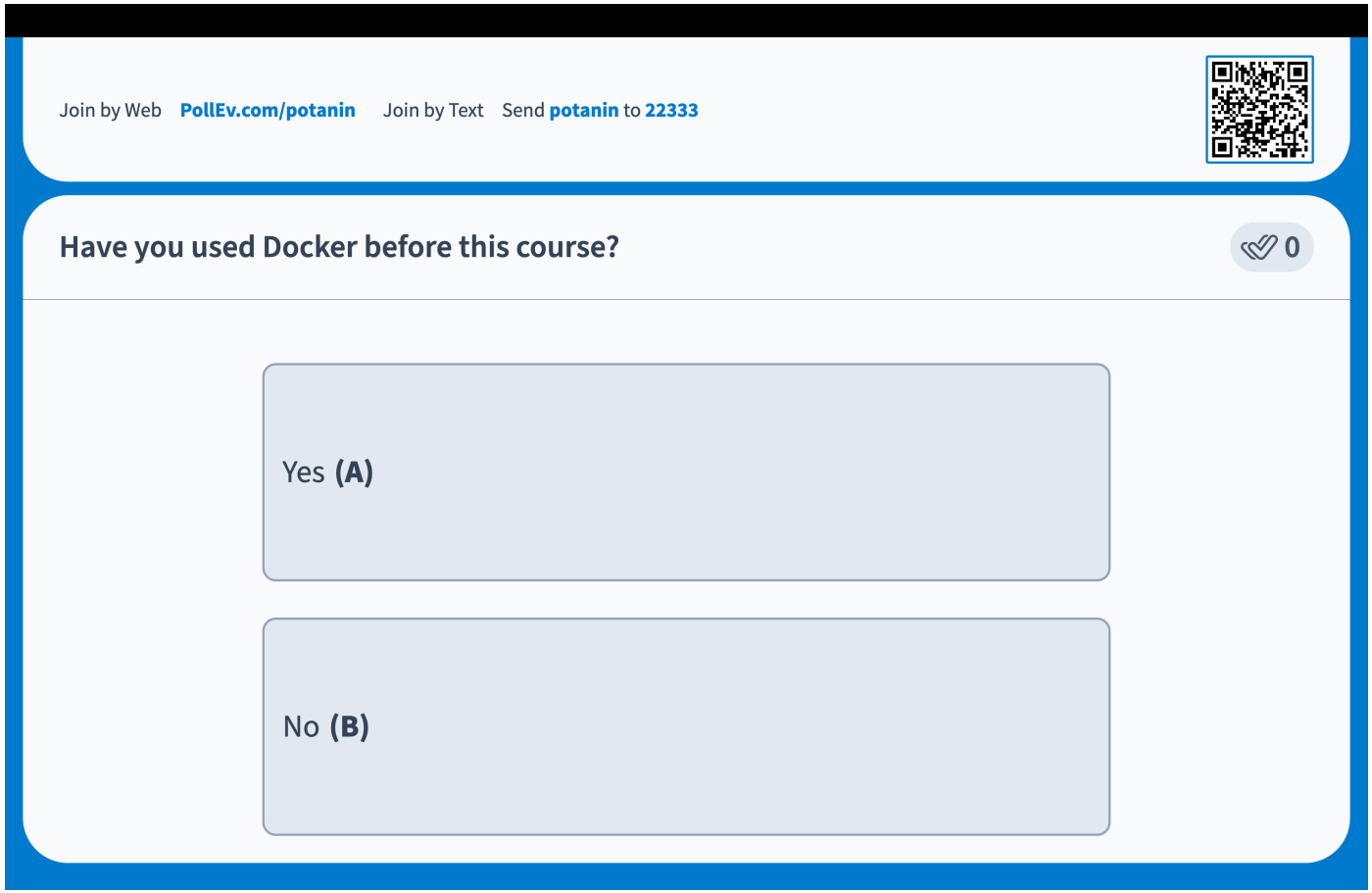
- ***Reliability***
  The complexity of managing a complex infrastructure means that system administrators often make simple mistakes, especially when the same changes have to be made to several servers. Automating the process avoids these mistakes.

- ***Recovery***
  Like any other code, your infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems you can easily revert to an older version and reinstall the environment that you know works.

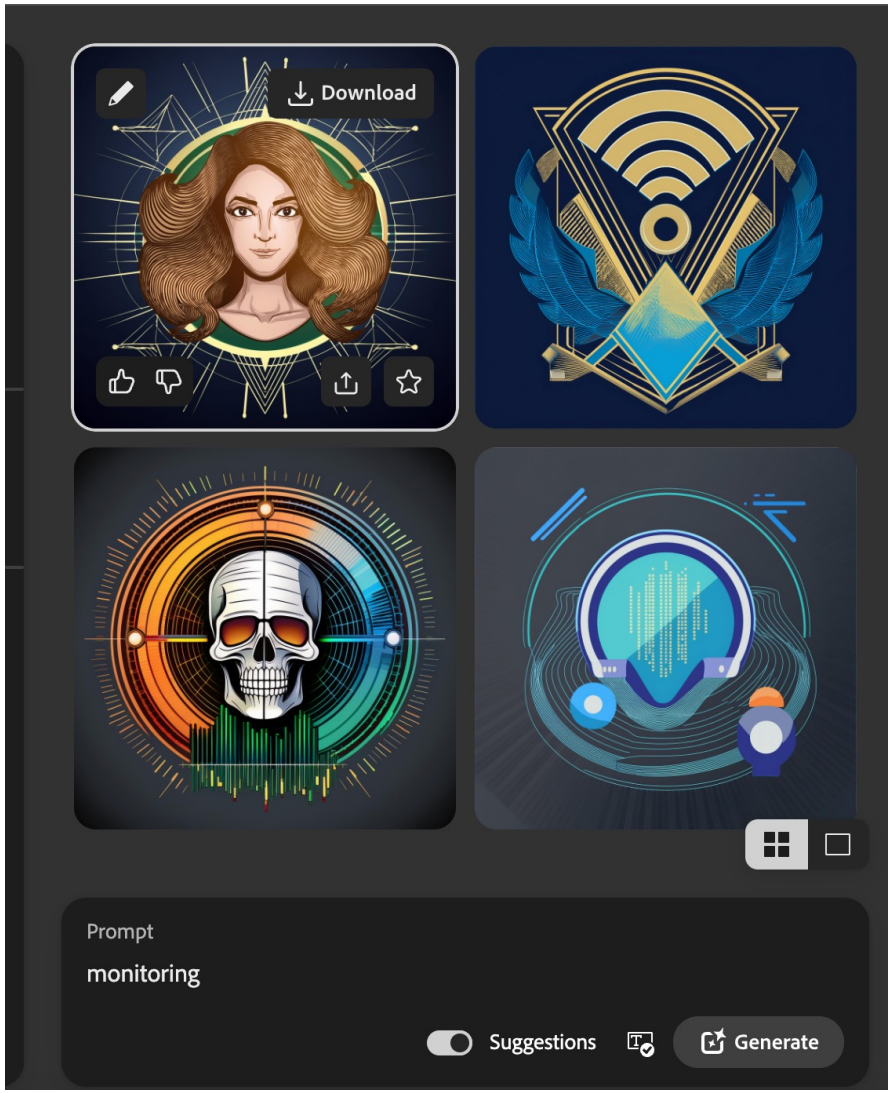
# Containers


- A container provides a stand-alone execution environment running on top of an operating system such as Linux.
- The software installed in a Docker container is specified using a Dockerfile, which is, essentially, a definition of your software infrastructure as code.
- You build an executable container image by processing the Dockerfile.
- Using containers makes it very simple to provide identical execution environments.
    - For each type of server that you use, you define the environment that you need and build an image for execution. You can run an application container as a test system or as an operational system; there is no distinction between them.
    - When you update your software, you rerun the image creation process to create a new image that includes the modified software. You can then start these images alongside the existing system and divert service requests to them.

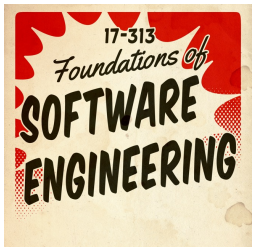
# Poll Everywhere Time!



# Monitoring



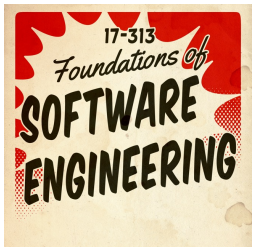
# What is Observability?




"As a philosophy, ***observability*** is our ability as developers to know and discover what is going on in our systems. In practice, it means adding telemetry to our systems in order to measure change and track workflows."

The New Stack, "What is observability?" 28 Feb 2020
https://thenewstack.io/what-is-observability/

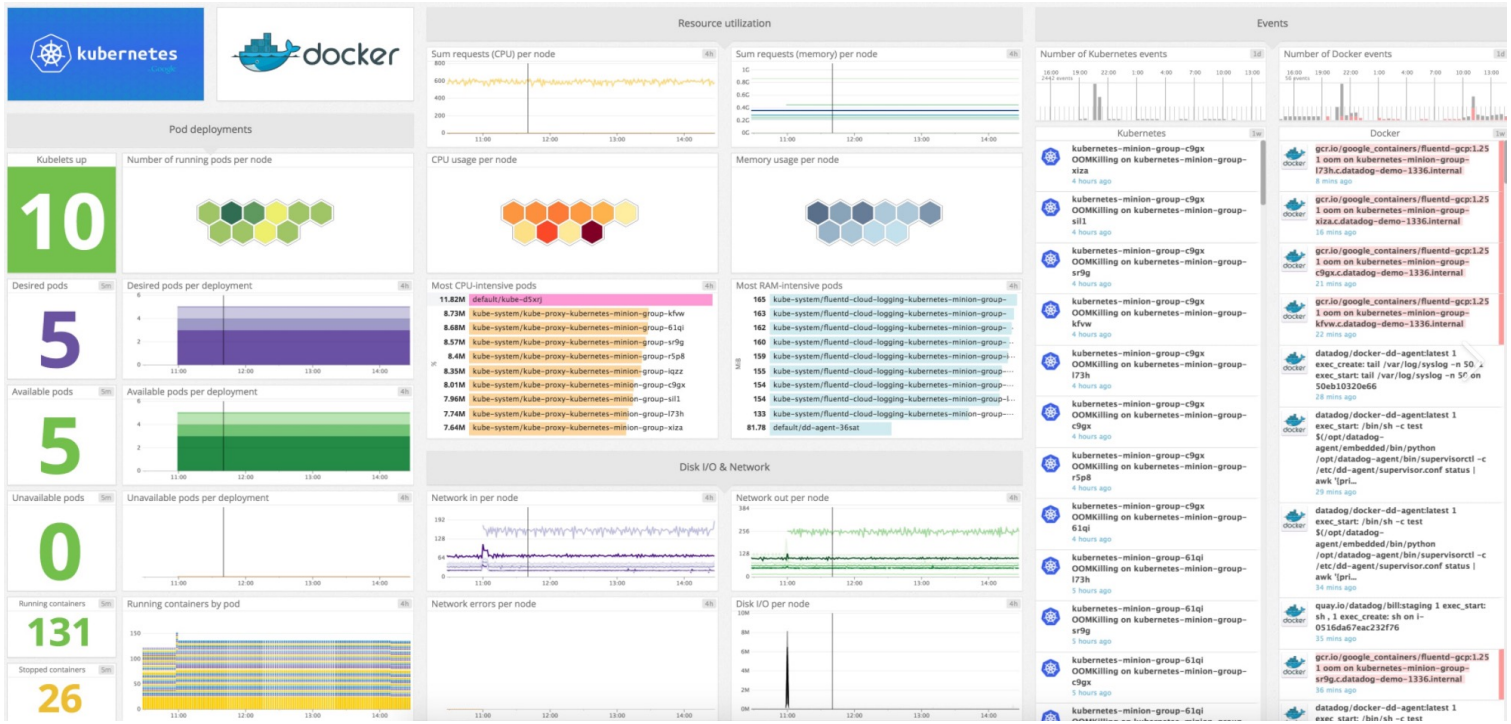
# Observability: Dashboards




1. What's happening now?
2. What does "normal" behavior look like?
3. What does it look like when something's gone (or is going) wrong?
4. Can I correlate events to changes in the actual graphs?

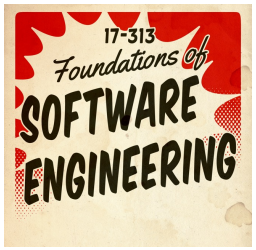
reference: https://www.youtube.com/watch?v=mBU3AJ3j1rg

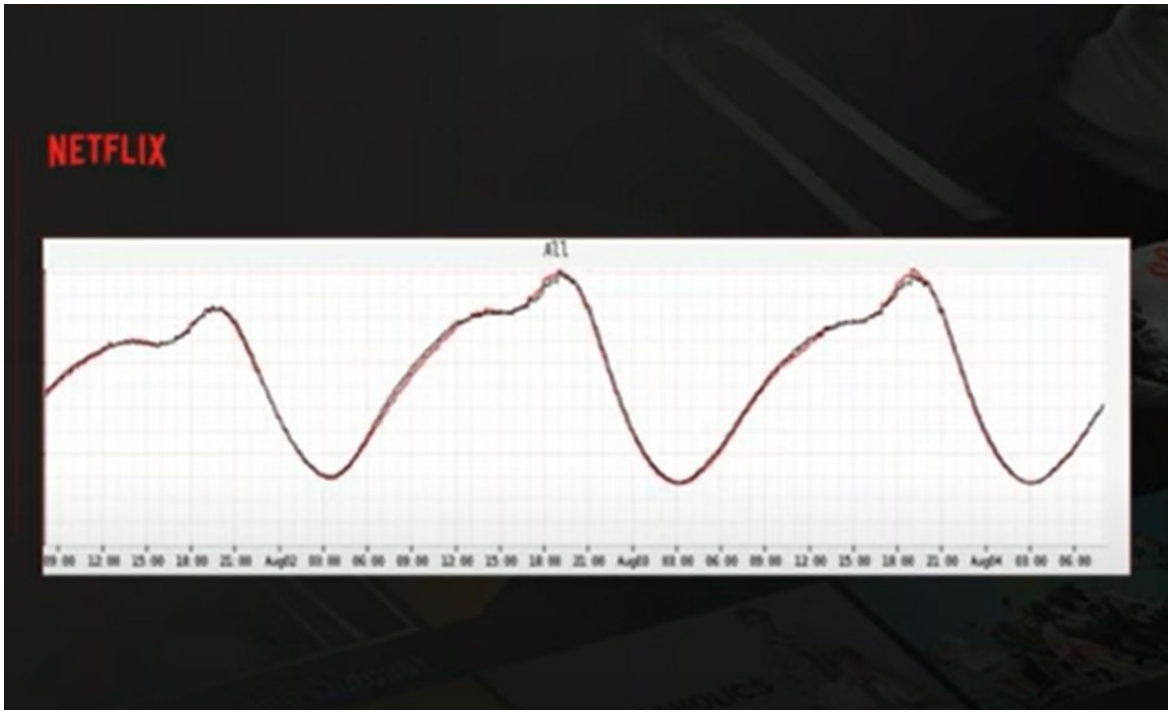
# Observability: Dashboard Example




https://datadog-prod.imgix.net/img/blog/monitoring-kubernetes-with-datadog/kubernetes-dashboard.png?fit=max

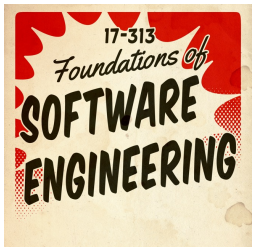
# Observability: Defining “Normal”

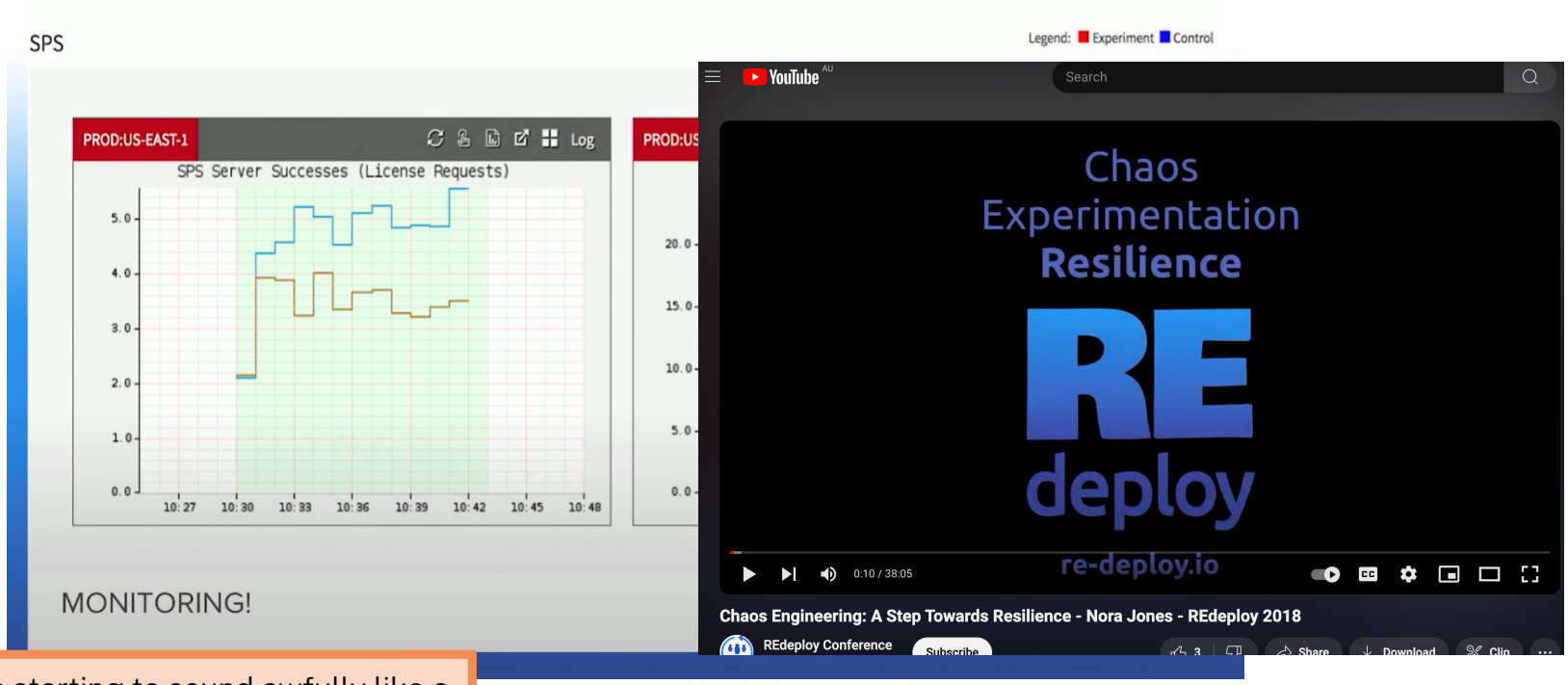



reference: https://www.youtube.com/watch?v=vq4QZ4_YDok

# Observability: When things aren't "Normal"




This is starting to sound awfully like a *quality attribute*....

reference: https://www.youtube.com/watch?v=qyzymLlj9ag

# Poll Everywhere Time!