The InfoQ eMag / Issue #103 / December 2021

Paths to Production: Deployment Pipelines as a Competitive Advantage

InfoQ

Paving the Road to Production

Safe and Fast Deploys at Planet Scale

Improving Speed and Stability of Software Delivery Simultaneously at Siemens Healthineers

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

InfoQ @ InfoQ InfoQ InfoQ

Paths to Production: Deployment Pipelines as a Competitive Advantage

IN THIS ISSUE

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITORS Maureen Spencer DESIGN Dragos Balasoiu / Ana Ciobotaru
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Graham Jenson

is an infrastructure engineer at Coinbase helping developers securely and reliably deploy software. He has worked on making infrastructure-as-code collaborative, creating large scale cloud systems, and building out the Coinbase monorepo.

Mathias Schwartz

has been an infrastructure engineer at Uber for more than 5 years. He and his team is responsible for the deployment platform for stateless services used across all of Uber engineering. Mathias has a PhD in Computer Science from the Programming Languages group at Aarhus University.

Vladyslav Ukis

graduated in Computer Science from the University of Erlangen-Nuremberg, Germany, and later from the University of Manchester, UK. He joined Siemens Healthineers after each graduation and has been working on Software Architecture, Enterprise Architecture, Innovation Management, Private and Public Cloud Computing, Team Management, Engineering Management and Digital Transformation. Since 2021, he has additionally been holding a reliability lead role for all the Siemens Healthineers Digital Health products.

A LETTER FROM THE EDITOR

Steef-Jan Wiggers

is one of InfoQ's senior cloud editors and works as a Technical Integration Architect at HSO in The Netherlands. His current technical expertise focuses on integration platform implementations, Azure DevOps, and Azure Platform Solution Architectures. Steef-Jan is a board member of the Dutch Azure User Group, a regular speaker at conferences and user groups, writes for InfoQ, and Serverless Notes. Furthermore, Microsoft has recognized him as Microsoft Azure MVP for the past eleven years.

Enabling developers to push code to production at an ever-increasing velocity has become a competitive advantage. By rapidly deploying applications, companies can easily keep up with changes in the business and surrounding market and thus maintain competitiveness. Automating deployments allow developers to reduce errors, increase productivity, and deploy more frequently.

In the early days, deployments from development environments to production were predominantly a manual process or consisted of utilizing a chain of custom scripts. Both developers and operations teams had to spend a lot of time on laborious manual chores like code testing and release. With the introduction of continuous integrations and deployments capabilities through tooling, this process became more automated. Moreover, pipelines were introduced with the primary purpose of keeping the software development process organized and focused.

Over the years, pipelines became more sophisticated and more critical for companies' IT departments that went through digital transformations. More software became available online as services, APIs, and products requiring a quick update and maintenance cycle. Furthermore, companies embraced the DevOps processes around pipelines and - once applied correctly - gained a competitive advantage, according to the research from the “Accelerate” book by Nicole Forsgren, Jez Humble, and Gene Kim.

In this eMag, you will be introduced to the paths to production and how several global companies supercharge developers and keep their competitiveness by balancing speed and safety. We've hand-picked three full-length articles to showcase that.

In the first article, Graham Jenson, Infrastructure Tech Lead at Coinbase, describes the story of how his company benefited from its deployment pipelines and how the organization got to where it stands now. Specifically, it describes Coinbase's experience creating "paved roads," which are curated technology and platform paths walked by developers to get their code into production. Furthermore, it discusses deploying pipelines at Coinbase for the past five years.

In the second article, Mathias Schwarz, a software engineer at Uber, explains how his company deploys its services for products like UberRides and UberEats globally. The article talks about Uber's journey from a small-scale company where engineers managed individual servers to the zone-based abstraction of "Micro Deploy," where they could automatically manage servers - and the Up system, which the company uses to automate everything at the regional level.

And finally, in the third article, Vladyslav Ukis, a Software Development Lead at Siemens Healthineers, focuses on the software delivery process at Siemens Healthineers Digital Health. The process is subject to strict regulations valid in the medical industry, and the article describes their journey of transforming the process towards speed and stability. Both measures improved simultaneously during the transformation, confirming research from the "Accelerate" book.

We hope you enjoy this edition of the InfoQ eMag. Please share your feedback via editors@infoq.com or on Twitter.

Paving the Road to Production

by **Graham Jenson**, Infrastructure Engineer at Coinbase

Outline

Coinbase has gotten much from its deploy pipelines. We deploy thousands of servers across hundreds of projects per day in order to serve our millions of customers and their billions in assets. Although I'm super proud of where we are today, that's not the whole story.

This article is about where Coinbase is now. It explores the journey we took to get here. Specifically, it describes our paved roads and how they've had to change over time in response to our company growing.

Roads

Roads might be the most expensive thing you touch every day, with a typical road costing around $3 million per mile. The only way that this immense cost could possibly be worth it is if we share the roads. Everybody shares the same road; whether you drive a Toyota or a Tesla, you expect the same quality. You have a contract with that road. There's enough lanes for the required traffic.

You can drive safely at or above the speed limit. The car you buy from a manufacturer is allowed to be driven on the road.

You probably only even notice a road if it breaks that contract. The road is not wide enough for traffic. It's not safe enough with all these potholes. The drainage is broken so the road is flooded. It's the same with software infrastructure. If there are many different paths to production, your applications don't share a common road; it will be more expensive to provide a good, reliable experience.

Imagine a company where every project uses a different source control system, a different build and CI tool, a different deploy pipeline. You would need to hire an infrastructure engineer to support each project. That would be like every car manufacturer building roads that only their cars can drive on.

Where the Rubber Hits the Road

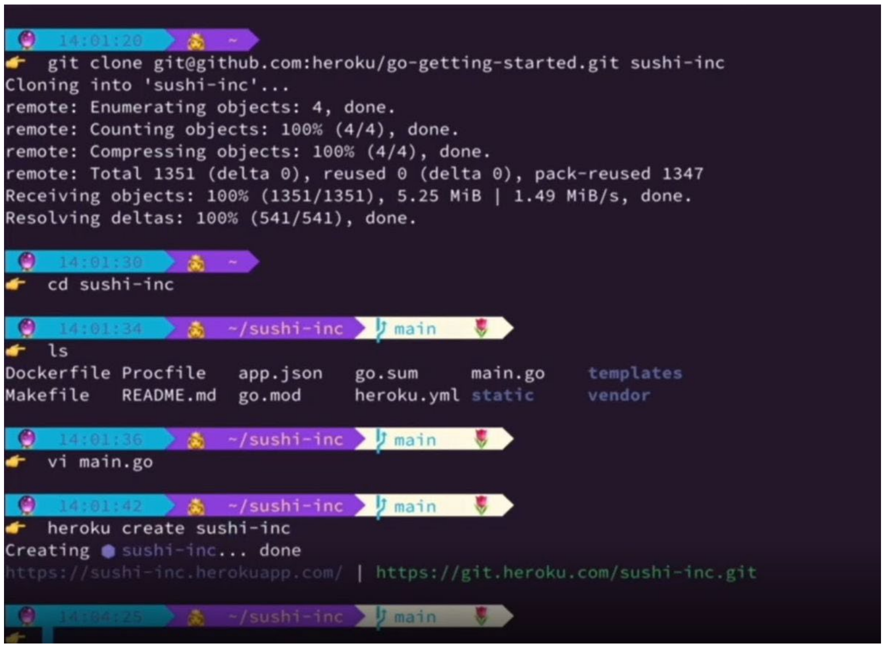
A developer shows up at your company and wants to get a new application into production- what are the steps they have to go through to get their code deployed? How is the application developed? How is it built and tested? How are the resources provisioned? How is the application configured? How is it deployed?

These questions need answers because they make up the contract between application and infrastructure engineering. It's the surface area between the two. It's where the rubber meets the road. The goal is to reduce this area into a single lane with clear direction, a paved road.

Let's Hit the Road

Coinbase was founded in 2012. It started out as a rails app deploying onto Heroku. Heroku is a platform as a service. It's a pretty fast way to get an application spun up and seen by the world. To deploy, you get pushed to a remote branch. This will kick off a build pack, which is like a Docker file that builds and containerizes your application and launches it into the cloud. The application is configured on Heroku's admin panel with information like the number of servers to run and the environment variables.

At a point in time, when the company's priority is to get your product in front of people, why would you want anything more complicated? The paved road to Heroku then is about creating a Git repository for your project. Actually, create the application. Add the project to the CI server. Ask Bob to create the Heroku application. Ask Bob to create the Heroku resources. Ask Bob to create the Heroku configuration. Go through security and code review, and get pushed to deploy.



```
 14:01:20   ~
   git clone git@github.com:heroku/go-getting-started.git sushi-inc
Cloning into 'sushi-inc'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 1351 (delta 0), reused 0 (delta 0), pack-reused 1347
Receiving objects: 100% (1351/1351), 5.25 MiB | 1.49 MiB/s, done.
Resolving deltas: 100% (541/541), done.

 14:01:30   ~
   cd sushi-inc

 14:01:34   ~/sushi-inc  main
   ls
Dockerfile Procfile   app.json   go.sum     main.go    templates
Makefile   README.md  go.mod     heroku.yml static     vendor

 14:01:36   ~/sushi-inc  main
   vi main.go

 14:01:42   ~/sushi-inc  main
   heroku create sushi-inc
Creating ⬢ sushi-inc... done
https://sushi-inc.herokuapp.com/ | https://git.heroku.com/sushi-inc.git

 14:04:25   ~/sushi-inc  main
```

Bob is a blocker; he has the Heroku login, which means if we want to create a new application or change a configuration, we have to ask him. Getting back to our metaphor, Heroku is like a toll road. We are sometimes paying a lot to use someone else's paved road. We don't control quality, security, or safety, and it introduces blockers like Bob. On the other hand, we don't have to spend a lot of time learning to pour concrete by building our own deploy pipelines.

The Gravel Road

Heroku isn't perfect. Although a lot of companies start there, they soon move off of it, either because of the cost, the security, or the lack of control over the environment. In late 2014 when Coinbase had a few projects and teenage engineers, we started the discussion around what it would look like to replace Heroku with our own paved road.

There were a few things we didn't want to lose when moving off of Heroku. Using Git commits as the primary index for what is deployed means you can find out the exact code that's in production, and also, keep parts of the Twelve-Factor App philosophy, like stateless processes, one project per repository, and explicitly declared dependencies. This will help us build scalable and resilient applications. We also wanted to add some more features, like ensuring that deploys were properly scanned and reviewed before going out, the ability to scale horizontally to dynamic events, and allowing our users to control their own configuration.

We decided to move to AWS for the scale it can give us in spinning up hundreds of machines in minutes, and the security controls it has over its many products. Taking further inspiration from projects like Etsy's Deployinator, we started to have discussions about the culture that we wanted to build. We wanted to default to open, and let developers contribute and deploy to any project in the company. We wanted to deploy on the first day. We want to remove the fear of deploying by having every new developer deploy coinbase.com on their very first day at the company. We wanted to work hard to be dumb by showing relevant information, answering questions before they are asked, and being intuitive to use.

On the technology side, containerizing our applications made sense. We decided to use Docker and Docker Compose to define applications processes that need running. These processes would be split up and deployed to many AWS Auto Scaling groups, with declarative descriptions of how to handle a lifecycle when scaling management of many cloud instances. We also wanted better control over our sometimes very sensitive environment variables. This was before HashiCorp Vault was created. We wanted something similar, a secure store.

Codeflow

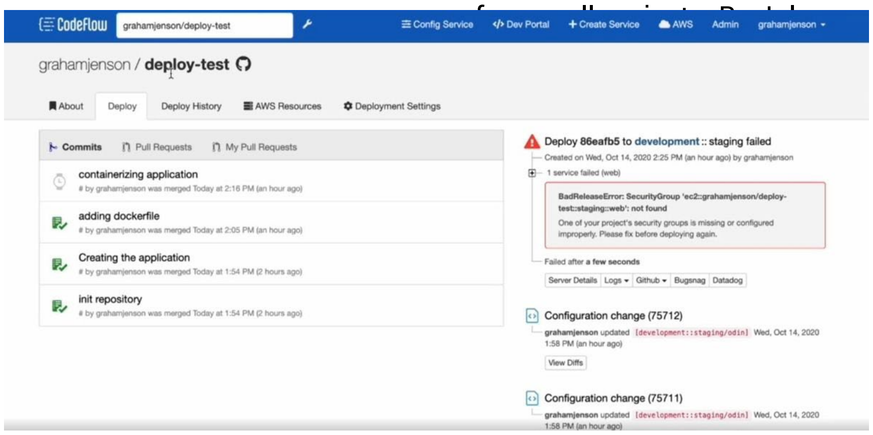
What we really liked about Heroku was the easy-to-use and informative interface. We ended up building our own interface and calling it Codeflow. Codeflow is a pretty simple CRUD Ruby application. A user would log in using GitHub authentication, and find the project they wanted to deploy. They would select it, and be presented with a list of commits and the history of deploys. You can then configure the project by selecting the deploy target and editing the Docker Compose, the environment variables, or the Auto Scaling group configuration. The user can select a commit, which shows whether its tests and security scanners have passed, if it's been properly reviewed, and if the build is ready to be deployed.

Each deploy target has a button, which when clicked, will create a deploy. When deployed, Codeflow will bundle up the configuration files, build an initialization script, and spin up all of the new Auto Scaling groups with attached security groups, IAM roles, and load balancers. These Auto Scaling groups will create new compute instances. Once those instances become healthy, the old instances are attached and deleted, finishing the deploy.

With Codeflow, our paved road now is about creating a Git repository for your project. Create the application. Write a Docker file to containerize that application. Add the project to the CI server. Ask Bob to create AWS resources like security groups, IAM roles, and load balancers. Add the project to Codeflow with the Docker Compose, the Auto Scaling group configuration, and all the environment variables. Go through security review and code review. Click to deploy.

If a developer follows these guidelines, they'll get their project into production. The only place Bob still has a job is creating AWS resources. This is only needed infrequently. Since we codified all of our resources with Terraform, this is not so much of a blocker yet.

There are many other aspects of our infrastructure which are not part of the paved road; things like building our hardened AMI, or how our deployers manage the lifecycle of Auto Scaling groups. These details are the substrate of the road, the aspects that our



containerizing application
# by grahamjenson was merged Today at 2:16 PM (an hour ago)

adding dockerfile
# by grahamjenson was merged Today at 2:05 PM (an hour ago)

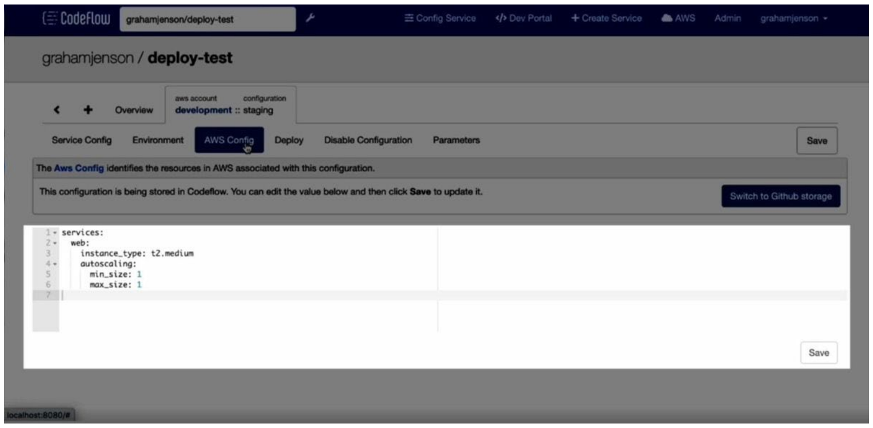
Creating the application
# by grahamjenson was merged Today at 1:54 PM (2 hours ago)

init repository
# by grahamjenson was merged Today at 1:54 PM (2 hours ago)

Deploy 86eafb5 to development :: staging failed
Created on Wed, Oct 14, 2020 2:25 PM (an hour ago) by grahamjenson
1 service failed (web)
BadReleaseError: SecurityGroup 'ec2::grahamjenson/deploy-test::staging::web': not found
One of your project's security groups is missing or configured improperly. Please fix before deploying again.
Failed after a few seconds

Configuration change (75712)
grahamjenson updated [development::staging/odin] Wed, Oct 14, 2020 1:58 PM (an hour ago)

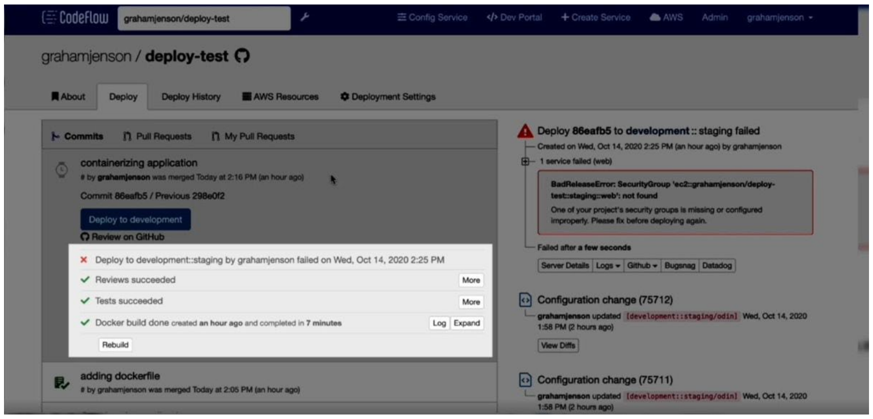
Configuration change (75711)
grahamjenson updated [development::staging/odin] Wed, Oct 14, 2020 1:58 PM (an hour ago)



The Aws Config identifies the resources in AWS associated with this configuration.

This configuration is being stored in Codeflow. You can edit the value below and then click **Save** to update it.

```
services:
  web:
    instance_type: t2.medium
    autoscaling:
      min_size: 1
      max_size: 1
```



containerizing application
# by grahamjenson was merged Today at 2:16 PM (an hour ago)
Commit 86eafb5 / Previous 298e0f2

✗ Deploy to development::staging by grahamjenson failed on Wed, Oct 14, 2020 2:25 PM
✓ Reviews succeeded
✓ Tests succeeded
✓ Docker build done created an hour ago and completed in 7 minutes

adding dockerfile
# by grahamjenson was merged Today at 2:05 PM (an hour ago)

Deploy 86eafb5 to development :: staging failed
Created on Wed, Oct 14, 2020 2:25 PM (an hour ago) by grahamjenson
1 service failed (web)
BadReleaseError: SecurityGroup 'ec2::grahamjenson/deploy-test::staging::web': not found
One of your project's security groups is missing or configured improperly. Please fix before deploying again.
Failed after a few seconds

Configuration change (75712)
grahamjenson updated [development::staging/odin] Wed, Oct 14, 2020 1:58 PM (2 hours ago)

Configuration change (75711)
grahamjenson updated [development::staging/odin] Wed, Oct 14, 2020 1:58 PM (2 hours ago)

developers shouldn't need to worry about. The only time they'll ever see them is if they hit a pothole.

Moving the Road

By March 2015, we had a prototype of Codeflow deploying com from Heroku onto Codeflow. This was very stressful.

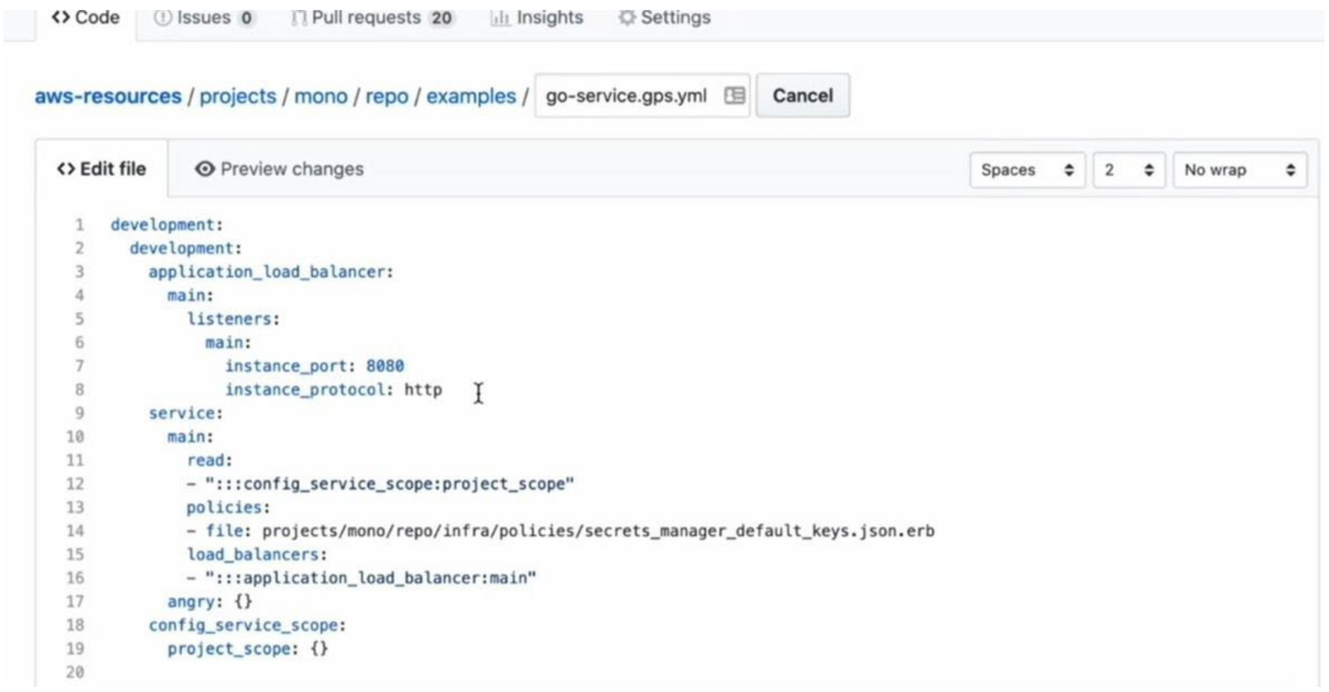
We made lists of things we needed to do before, lists of things we needed to do during, and lists of cleanup. Lists are great. They make it much easier to collaborate when you have many stakeholders all working together on a single project. They also highlight risky areas and clarify what to do if something goes wrong.

Coinbase.com was founded in 2012, and was on Heroku until July 2015. It took six months of planning, development, and execution to get us onto our own paved road. My point here is that there's no rush. Roads are difficult and expensive to build, even more so to change. Doing lots of groundwork here will save you in the long run.

Desire Paths

Desire paths are created by people walking a route, over and over, eroding the soil and eventually creating a new trail. They appear in places where there are no or inefficient existing paths. You can stop the erosion by building fences or you could learn from them and pave a more usable path.

Coinbase was a Ruby company but people wanted to start using Golang. There's really no blocker to using Golang. If you could containerize your application, then our deploy pipelines would work just fine. Also, there was no hard rule saying you couldn't use Golang, especially if you thought it was the right tool for the job. The only thing we had to convey is that it wouldn't be supported. They'd be going off-road.



```
development:
  development:
    application_load_balancer:
      main:
        listeners:
          main:
            instance_port: 8080
            instance_protocol: http
    service:
      main:
        read:
        - ":::config_service_scope:project_scope"
        policies:
        - file: projects/mono/repo/infra/policies/secrets_manager_default_keys.json.erb
        load_balancers:
        - ":::application_load_balancer:main"
      angry: {}
    config_service_scope:
      project_scope: {}

```

As more projects started using Golang, developers were working out patterns and solutions. The path was becoming more well-trodden. As this happened, we started to offer more support. After a few years of work, the road was paved and the developers were happy. In fact, Golang is now the preferred language at Coinbase.

As road makers, we don't really decide where the roads go. Every company will have a different path. My tool might not be right for you, because your problems aren't the same as mine. The team building the road just needs to look where people are going and pave underneath their feet.

Single Lane Road

In 2017 to 2018, Coinbase exploded with new users, new engineers, and new projects. When you get hundreds of projects all building and deploying at the same time, you quickly find out which of your paved roads don't have enough lanes. Bob being the only one creating AWS resources became a big blocker. Even though we had all of our resources codified using Terraform, adding to them or changing them became a huge pain.

To remove Bob from this workflow, we created a project called Terraform Earth. Terraform Earth allows developers to submit a change to our codified resources. These resources are in a format we created called GPS. GPS compiles to Terraform but simplifies it to encourage developers to manage their own resources. When a developer submits a change, Terraform Earth will comment with a detailed plan of what will happen if merged. Once the plan and the code have been reviewed, the change can be merged. Then, Terraform Earth will apply that change to the cloud.

This removes Bob as a blocker, and has enabled us to scale to hundreds of changes from hundreds of contributors per week. Our Auto Scaling group deployer also became a big blocker. It started out as nothing more than an infinite while loop listening to a queue. We found this design difficult to scale reliably and safely, and deploy times were getting painful during peak work hours.

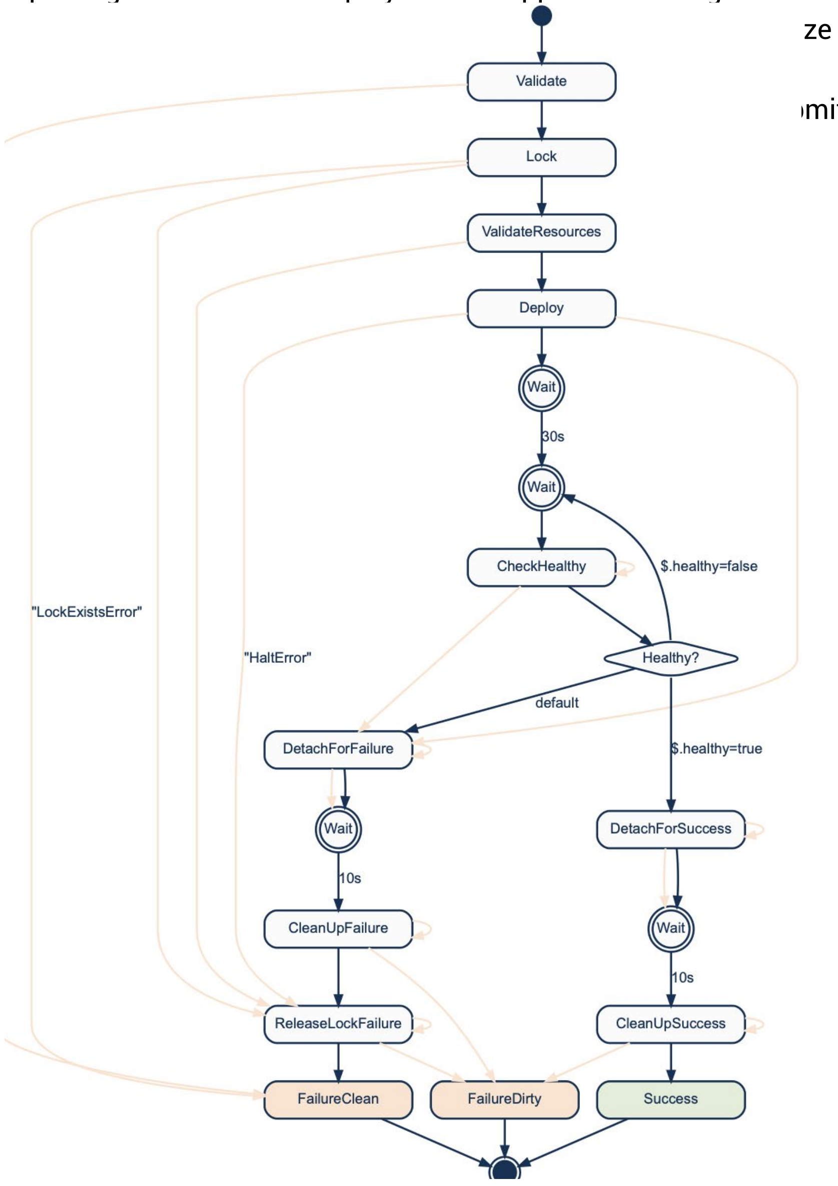
Teams were also wanting to deploy to other locations, like serverless with Lambdas and API gateways. What we saw was an explosion, not only in deployments but different types of deployers. In response to that, we created bifrost, our paved road for building deploy pipelines.

Bifrost uses AWS step functions, which are basically state machines that orchestrate AWS lambdas and come with strong guarantees, error and retry handling, and built-in scale. This lets us build deploys quickly, replacing our old ad hoc deploys

Around this point, we started building an internal framework called Coinbase Service Framework. This is a multi-language library that contains much of the boilerplate needed to build a service at Coinbase. The paved road with Terraform Earth and CSF is about creating a Git repository for your project. Create the application using CSF. Write

ze

mit a



code change to create the AWS resources that you need. Add the project to Codeflow with all of its configurations. Go through security and code review, and click to deploy.

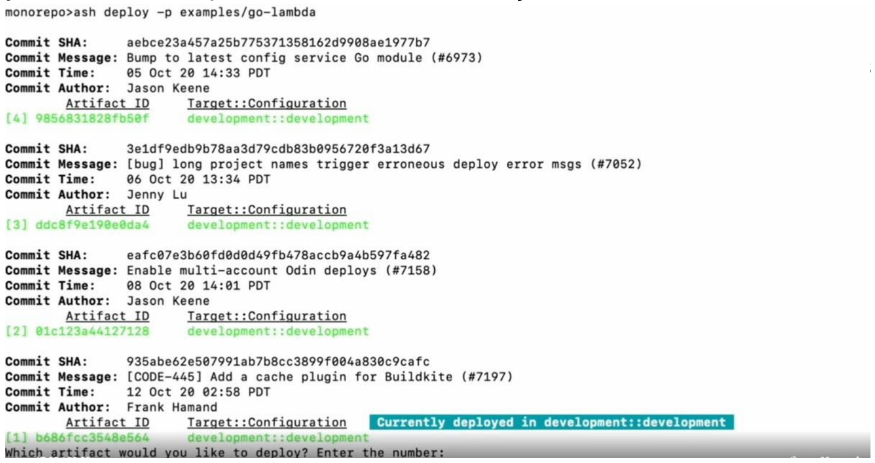
The Highway

From 2018 to the present, we saw Coinbase grow enormously. The biggest problem when you get thousands of projects and hundreds of engineers is that any ad hoc process, any tribal or siloed knowledge, becomes a massive pain point. The only way to scale is to document everything. We needed to build maps of all of our existing roads and where they went.

Another issue is the increased amount of work that goes into upgrading shared components, like CSF, and all the projects that depend on them. Before, if a component was used by a dozen projects, it might take an hour to upgrade all of them to a new version. When there are 500 projects, that can be weeks' worth of work. Components become a victim of their own success. The more projects that use them, the more difficult they become to maintain. This is one of the reasons why in late 2019 we started a monorepo at Coinbase. A monorepo is a single large repository that contains code for many projects. Having all components and their dependencies in a single location, with examples and tests, makes broad multi-project updates possible with a single commit.

With a monorepo we had to throw out some of our old assumptions. Firstly, and most obviously, a project is no longer one-to-one with the repository. Second, configurations like Docker Compose will now live with the code. Also, although we still use Docker, we don't allow each project to define its own Docker file. Instead, we have a shared base container for each different language.

For our monorepo, we use Bazel, an open source build tool based on Google's internal monorepo tool, Blaze. Bazel has some pretty strict constraints, like explicitly listing all the inputs and outputs for every build. This creates a queryable dependency graph from every file to every output. In the monorepo, when we see a new commit, we can calculate the exact tests to run and the exact artifacts we need to build based on the change files in that commit. This together with really good caching, and will allow us to expand our monorepo to thousands of projects. To deploy a project in the monorepo, we don't have a UI yet. Instead, we use a CLI tool called Artifact-Shipper, or Ash. To deploy from the monorepo, you just run, monorepo> ash deploy, the project. This will give you a list of artifacts to deploy and a list of where they can be deployed to. Once selected, Ash will send that artifact to one of our bifrost deployers. You can also use Ash to get deploy history, follow a current deploy, or perform a bunch of other related tasks. To create a new project in the monorepo, we built a bootstrapping tool, where you run, monorepo>bazel run



```
monorepo>ash deploy -p examples/go-lambda

Commit SHA:     aebce23a457a25b775371358162d9908ae1977b7
Commit Message: Bump to latest config service Go module (#6973)
Commit Time:    05 Oct 20 14:33 PDT
Commit Author:  Jason Keene
        Artifact ID      Target::Configuration
[4] 9856831828fb50f      development::development

Commit SHA:     3e1df9edb9b78aa3d79cdb83b0956720f3a13d67
Commit Message: [bug] long project names trigger erroneous deploy error msgs (#7052)
Commit Time:    06 Oct 20 13:34 PDT
Commit Author:  Jenny Lu
        Artifact ID      Target::Configuration
[3] ddc8f9e190e0da4      development::development

Commit SHA:     eafc07e3b60fd0d0d49fb478accb9a4b597fa482
Commit Message: Enable multi-account Odin deploys (#7158)
Commit Time:    08 Oct 20 14:01 PDT
Commit Author:  Jason Keene
        Artifact ID      Target::Configuration
[2] 01c123a44127128      development::development

Commit SHA:     935abe62e507991ab7b8cc3899f004a830c9cafc
Commit Message: [CODE-445] Add a cache plugin for Buildkite (#7197)
Commit Time:    12 Oct 20 02:58 PDT
Commit Author:  Frank Hamand
        Artifact ID      Target::Configuration   Currently deployed in development::development
[1] b686fcc3548e564      development::development
Which artifact would you like to deploy? Enter the number:
```

Our paved road and the monorepo now is about executing "bazel run new" to create a new project with CSF, and submitting a code change to create your AWS resources; going through security and code review. Ash deploy. This has removed and simplified so many steps compared to where we were just a few years ago.

Our monorepo is still in its early stages. It contains dozens of projects, and supports four languages: Python, Ruby, Go, and Node. It continues to grow daily. The next big project though, will be moving coinbase.com into the monorepo. At the moment, we're just laying the foundations for that big move.

Fork in the Road

The United States Interstate Highway System is a massive grid of roads that crisscrosses every state. It would cost around

driven in a year in the states is on those highways, it's probably worth it. Imagine the way in which interstate roads were built before this was in place; each state would have to negotiate with its neighbors about how and where to build the roads. Sure, Hawaii and Alaska would have it easy, but it would be a nightmare for states like Tennessee and Missouri, which each have eight neighbors. For drivers, the safety, the rules and the quality of the roads would change from state to state, making it a horrible experience. Having a single entity set the standards by which these roads operate removes the need for in-squared negotiations.

This is the same at any organization; if you have to negotiate with every team and project about how they're going to build and deploy, it's going to be very difficult to build a shared paved road. This means less quality for developers and infrastructure engineers. If you want to build a shared paved road, the very first step has to be getting buy-in from your organization.

End of the Road

Building a shared paved road lets you focus on the specific problems developers have in getting their code into production. The easiest place to start is to write a list of all of the steps a developer needs to go through to get a new application into production. Once you have that list, work on reducing the number of variations, eliminating steps, simplifying decisions made by developers, and removing any blockers like Bob.

LaunchDarkly

SPONSORED ARTICLE

So You Took Down Production - Now What?

by **Jessica Cregg**

We've all been there. You're pushing to main, something you've done hundreds of times before, and by the time you go to refresh your production environment, you notice something's wrong. When you take down prod, the point at which you see is already too late. Your name is on the deployment, the issue is live, and the clock is ticking.

So what happens now?

The first thing to do is understand where you've gone wrong. Use the diff command to highlight the code that's changed without relying on your own eyes to spot the change. We can sometimes become so accustomed to the way we write our functions that a breaking change will be practically invisible. Remember those recent additions like git switch and restore are there to make your life easier and help you triage while looking for a fix.

Next, you need to accept that it might not just be the code. It might, in fact, be a combination of your environment, your infrastructure, and the combination of all three being stress-tested by particularly unusual behavior. Don't assume that you can handle this on your own. Simply rolling back your change may not work. Contact DevOps.

It's always better to ensure that the people who need to know about a worst-case scenario are alerted if there's even a remote chance it could be in effect.

The most important thing to remember is that this happens to nearly every engineer—no matter their status or tenure, no one is immune to taking down production. Whether you're joining a new engineering team or embarking on a new stage of a product's life cycle, the likelihood of you breaking production is rarely a case of if. More often than not, bringing down production is closer to a matter of when.

Case in point, earlier this year, HBO Max surprised its 44 million-strong user base with an integration test email that went to everyone. While this example is one of the more achingly public displays of what happens when your production build goes awry, given enough honest discussions and code reviews, a forensic investigation would undoubtedly uncover countless instances of these occurrences happening.

One of the most reassuring principles of software development is that you can usually count on the task you're trying to accomplish having been performed by someone else. This goes for both the good and the bad. Whatever the dilemma, you can almost guarantee someone out there is experiencing a scar-tingling Harry Potter-esque flashback on your behalf. Here's how not to do it again.

Set the stage for rehearsals

When you're running a race, you want the conditions to be right: not too hot, a manageable incline, and most importantly, an easy-to-follow route. The same goes for your training runs. You ideally want to be stress-testing yourself under conditions that prepare you for the final event. This analogy couldn't ring truer for your development environments. If your change is going to break in production, it needs to break in staging first.

Read the full-length article here

Safe and Fast Deploys at Planet Scale

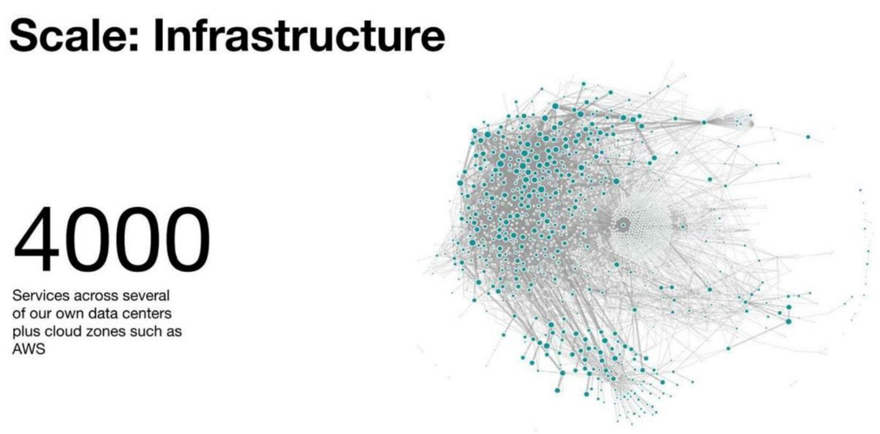
with **Mathias Schwartz**, Infrastructure Engineer at Uber

At QCon Plus, Mathias Schwarz, a software engineer at Uber, presented safe and fast deploys at planet scale. Uber is a big business and has several different products. They are, in most cases, deployed to dozens or hundreds of markets all over the world. The biggest of our products is the Uber Rides product that will take you from somewhere in town to somewhere else with a click of a button. Daily, Uber makes 18 million trips every day – and those are numbers from Q1, 2020. In addition to the trips on the Uber Rides platform, they have Uber Eats for meal delivery in the list of other products.

Scale - Infrastructure

To handle all these products, Uber has a large set of backend services. In their case, it's about 4000 different microservices deployed across machines in several data centers.

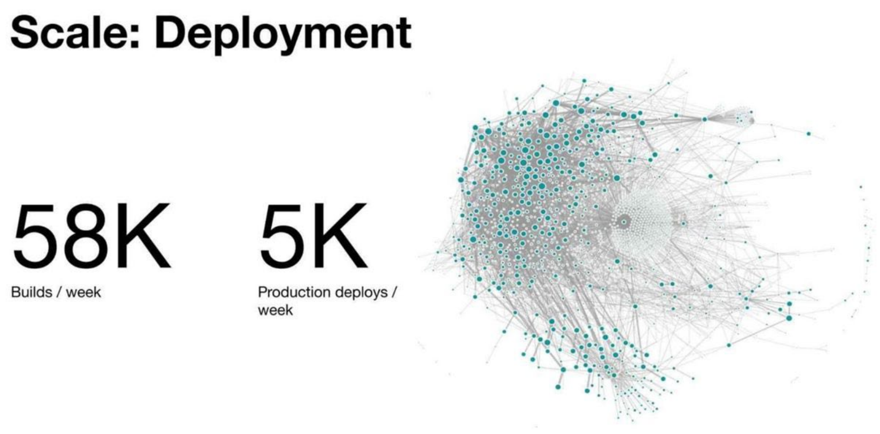
Since it takes a while to perform an upgrade of a service, it also means that there's never a point in time where the system isn't undergoing some upgrade of at least one of our backend services.



Scale: Infrastructure

4000

Services across several of our own data centers plus cloud zones such as AWS



Scale: Deployment

58K

Builds / week

5K

Production deploys / week

Scale - Deployment

At Uber we do 58,000 builds per week and roll out 5,000 changes to production every week. So if you look at those numbers, it means that every one of Uber's backend services, on average, is deployed more than once in production every week.

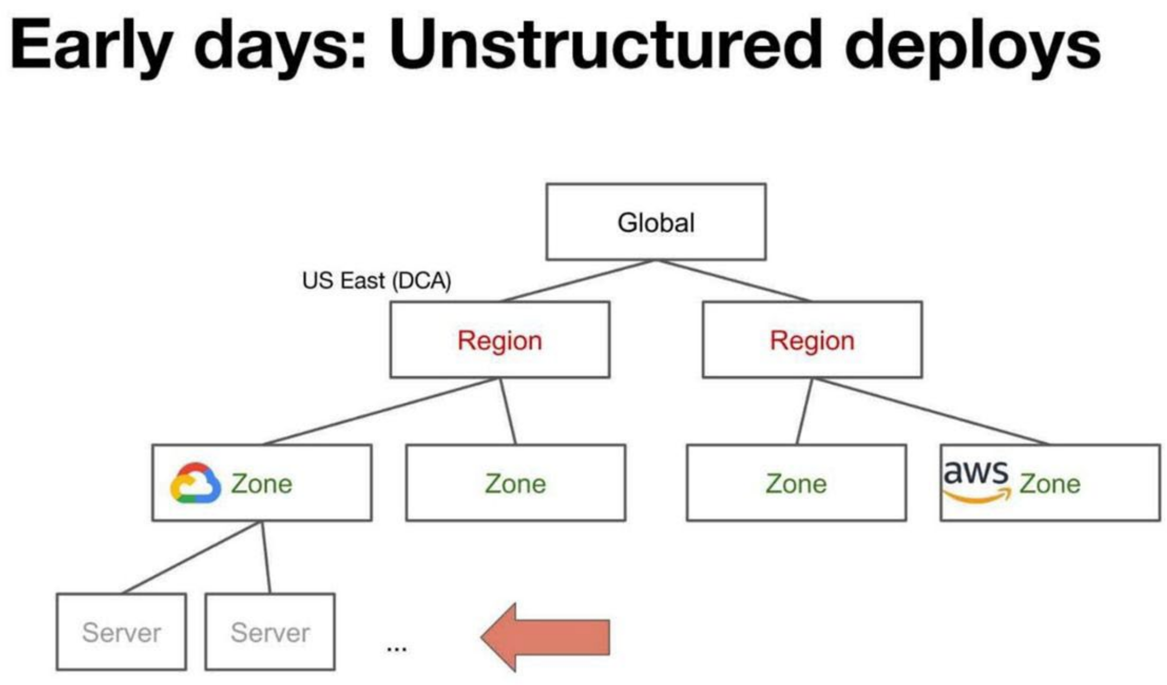
Regions and Zones

When we think of our infrastructure at Uber, we think of it in terms of these layers. At the lowest layer are the individual servers. The servers are the individual machines, the hardware that runs each of the processes. Servers are physically placed within some zone. A zone can either be something we own ourselves, a data center at Uber, or a cloud zone where the machines are part of the GCP public cloud or AWS.

A zone never spans multiple providers - it's always only one provider, and a set of zones makes up a region. A region is essentially zones that are physically close to each other so that there's low latency on calls between processes within these zones, which means that you can expect a request to have low latency from one zone to the other. Combined, these regions make up our global infrastructure. So when you want to deploy a new build into production, it is basically the process of globally deploying these new builds, to all the relevant servers in all zones of the Uber infrastructure.

Early Days: Unstructured Deploys

When Uber started building their deploy strategies and deploy systems, it started out the same way as most other companies. Each of Uber's service teams had a specific set of hosts where they would deploy their new builds. Then, whenever they wanted to release a change, they would go to these servers, either manually or use a Jenkins script to deploy the build to the servers and make sure that they had upgraded all of their processes to roll out that new build. However, this approach had several drawbacks. For instance, it would be a



manual process for the team to clean up when a server failed. Even worse, if there were a bug in the change that was being rolled out, it would mean that the team would have to clean that up and get the system back to a good state after getting their bad change out of the production system.

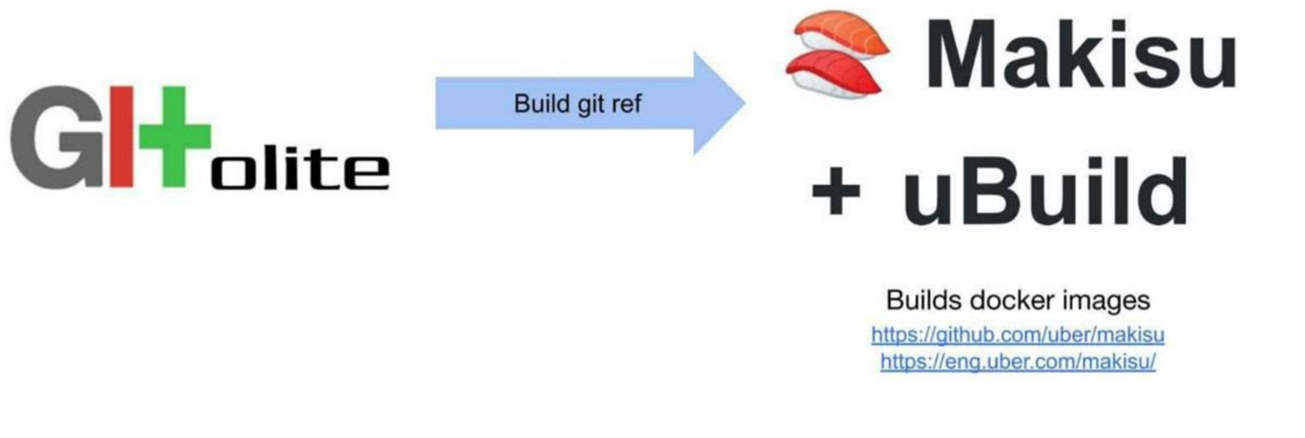
Important Deploy System Features

In 2014, we took a step back and began thinking about what it would take to create a deploy system that will automate all these operations and make it easier for our engineers to keep deploying at a high frequency, but also, at the same time, make it safe. We at Uber came up with a list of requirements of things we wanted the system to be able to do. We wanted our builds to be consistent; moreover, we also:

- Wanted the builds to look the same, regardless of what language was used, what framework was used, and what team was building the service. The build should look the same to the deploy system to make it easier to manage them.
- In addition, we wanted all deploys to have zero downtime, which means that when you want to roll out your build, you want the system to manage the rollout order to the servers automatically. We wanted the system to make sure not to stop more processes than it can without interfering with the traffic that goes into the service.
- Wanted to make outage prevention a first-class citizen of this system. Essentially, we wanted the system to be able to discover and respond to issues if there were any issues when we rolled out a new build to production.
- Finally, we wanted the system to be able to get our backend back to a good state. Overall, the idea was that this would let our engineers simply push out new changes and trust the system to take care of the safety of those deploys.

Structured Deploys With uDeploy

Based on these requirements, we at Uber started building the Micro Deploy system. Micro Deploy went live in 2014. Over that year, we moved all our backend services to that new platform.



In Micro Deploy, we made all our builds be Docker images. We did that using a combination of a build system called Makisu that we built internally. Essentially, these two systems combined meant that all our Docker images looked the same and would behave the same to the deployed system, simplifying management of deploys quite significantly.

Deploy Into Cluster in Zones

At Uber, we also changed the level of abstraction for our engineers. Instead of worrying about the individual servers to deploy to, we told them to tell us which zones and what capacity they wanted in each of those zones. So instead of asking the engineer to find specific servers, we had capacity in these zones. We would then deploy into that zone. Whenever there was a server failure, we would replace that, and the service would be moved to these new servers without any human involvement. We did that in uDeploy, using a combination of the open-source cluster management system called Mesos, plus a stateless workload scheduler called Peloton that we built internally at Uber, and made it open source. Today you can achieve something similar using Kubernetes.

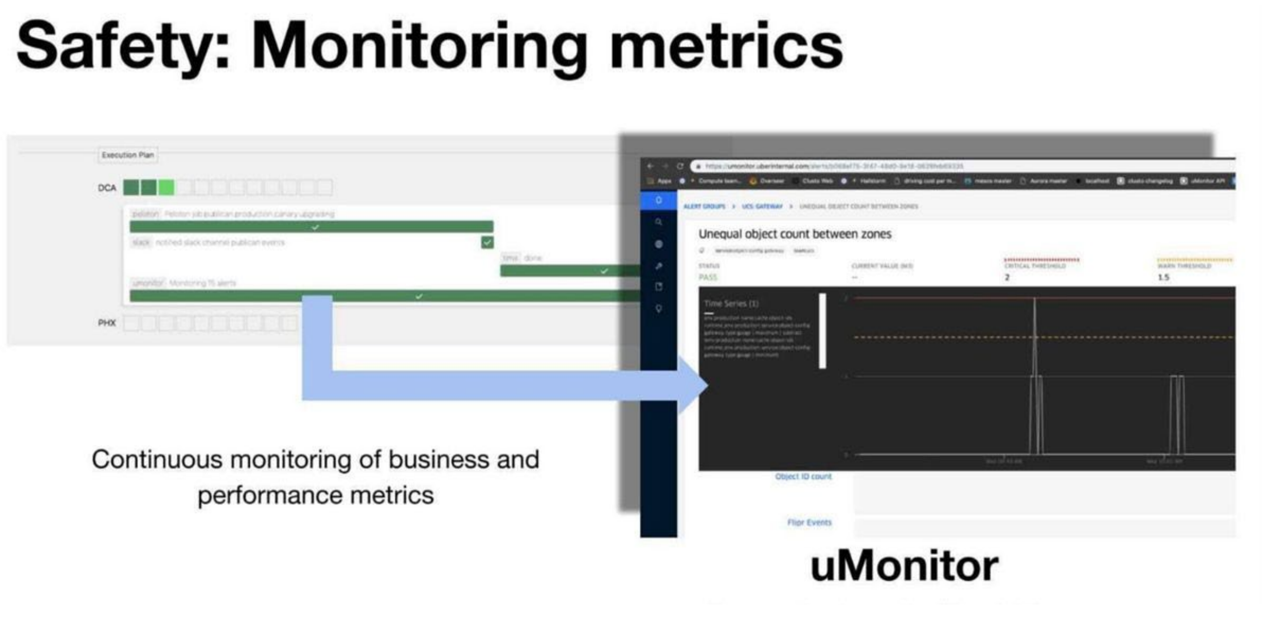
Safety - Monitoring Metrics

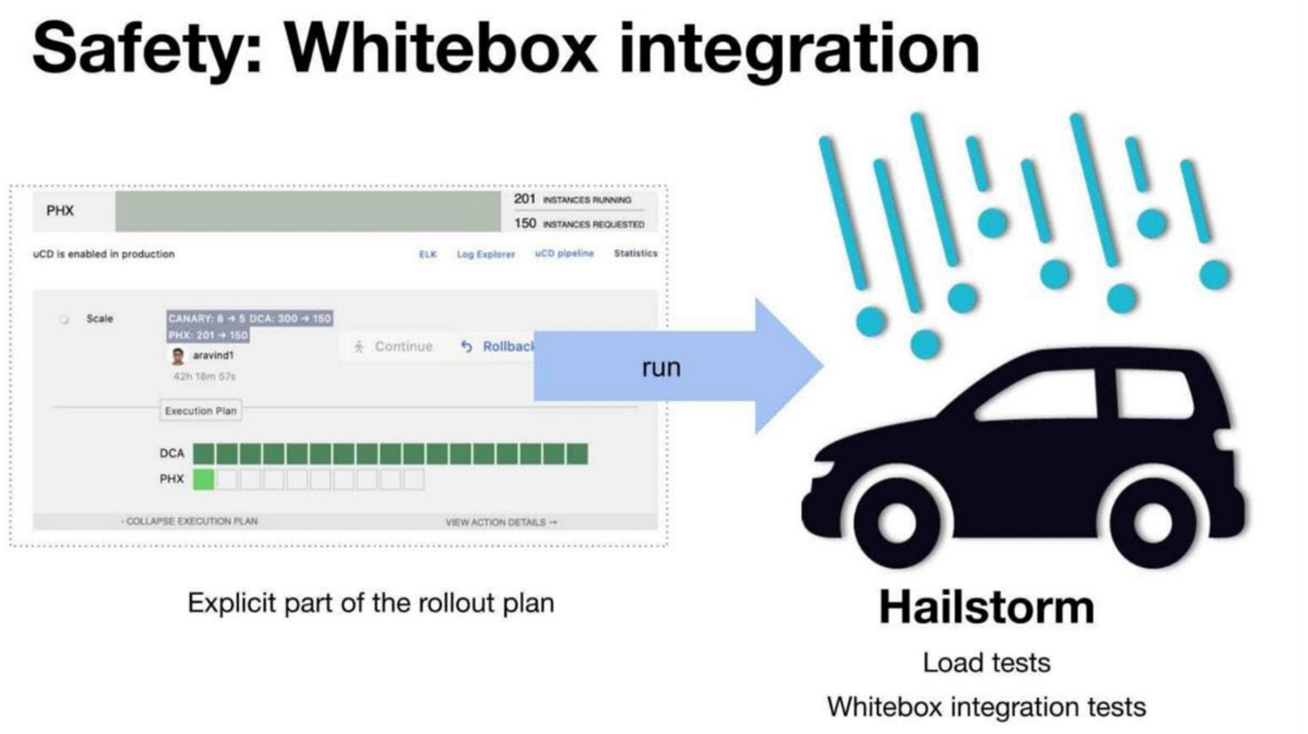
We also decided to build safety mechanisms directly into the deployed platform to make our deploys as safe as possible. One thing that we built into the deployed platform is our monitoring system, uMonitor. All our services emit metrics that are ingested by uMonitor. uMonitor continuously monitors these metrics in time series and makes sure that the metrics do not go outside some predefined threshold. If we see the database metrics break these predefined thresholds, we will initiate a rollback to a safe state, which will automatically happen in the Micro Deploy system. Micro Deploy captures the system's previous state, and then when the rollback is initiated, Micro Deploy automatically gets the service back to its old state.

Safety - Whitebox Integration

Also, for our most important services at Uber, we have Whitebox integration testing. We use a system that we developed internally called Hailstorm. When you roll out the first instances to a new zone, it will run load tests for these specific instances in production and run Whitebox integration and load tests. This happens in addition to large sets of integration tests that are run prior to landing the code.

These integration tests hit the API endpoints of the deployed service and make sure that the API still behaves as we expect.


Safety: Monitoring metrics
Execution Plan
DCA
PHX
Continuous monitoring of business and performance metrics
Unequal object count between zones
uMonitor

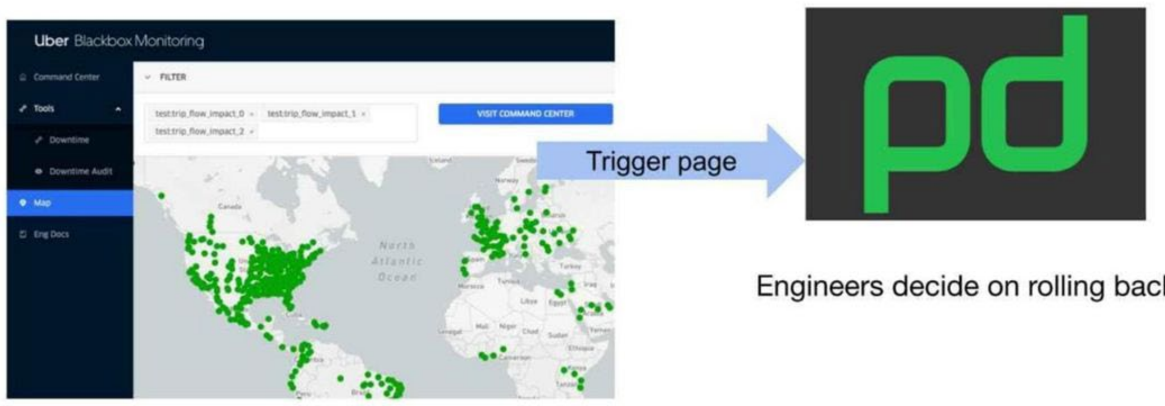


By doing this on the first few instances that roll out to a zone, we can discover issues in production before they hit more than a few of our hosts. We can also roll back to the previously known safe state for the service due to some of these tests failing.

Safety - Continuous Blackbox

Finally, we have built what we call BlackBox testing. Blackbox testing is essentially virtual trips happening continuously in all the cities where Uber products are live. Blackbox takes these virtual trips, and if we see that you cannot take a trip in one of those cities, then there'll be a page to an engineer. This engineer will then manually have to decide whether to roll back or whether to continue the rollout. They'll also have to determine which services could have caused the trips suddenly on the platform to start seeing issues. So BlackBox

Safety: Continuous blackbox



Engineers decide on rolling back

Blackbox testing/monitoring

Virtual trips in production

testing is the last resort issue detection mechanism that we run.

Micro Deploy gave us safety at scale. It also gave us the availability of services, despite individual servers failing. A couple of years ago, we discovered that we were spending an increasingly large amount of our engineering time managing services. Engineers still had to figure out in which zones to place a service. Would they for example want to put a service on AWS or on our own data centers? How many instances would they need, and so on? Service management was still a very manual task two years ago.

Efficiency at Scale

Hence, we took a step back again and thought, what would it take to build that system that could automate all of these daily tasks for our engineers and ensure that the platform could manage itself?

We came up with three principles that we wanted to build into our system:

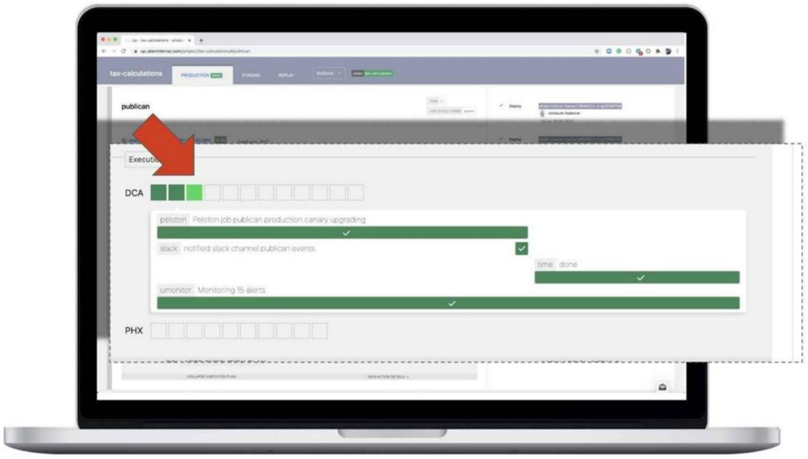
1. First, we wanted it to be genuinely multi-cloud, meaning that we didn't want any difference for our engineers, whether the service ran on a host or server that we owned ourselves in one of our own data centers or on one of the public clouds. It shouldn't matter. We should be able to deploy anywhere without any hassle.
2. Secondly, we also wanted it to be fully managed, meaning that we want the engineers to only worry about making their changes, making sure that these changes work, and rolling them out to production. We no longer wanted them to handle placement in zones, scaling the services, or other manual management tasks. At the same time, we still wanted the deploy system behavior to be predictable.
3. And finally, we still wanted the engineers to understand and predict what would happen to their service. So even if we decided to change the scaling or move them to a cloud zone, we wanted to tell our engineers what was happening and why.

Efficient Infrastructure with Up

Based on these three principles, we started building our current deploy platform at Uber, called Up. In Up, we took another step up in terms of the level of abstraction for our engineers to care about when they managed their services and rolled out their changes. So, for example, instead of asking them to care about individual zones, we asked them about their physical placement requirements in terms of regions. So as an engineer using Up, I'd ask for my service to be deployed into a specific region, and Up would then take care of the rest. That looks something like it is shown below to our engineers today.

We can see that this service is deployed into a canary and deployed into two different regions, which are, in this case, called “DCA” and “PHX”. We're not telling our engineers whether the physical servers run in cloud zones or whether they run in our own data centers. We're just telling them that there are these two regions, and this is how many instances they have in these two regions.

When the engineer does a rollout to production, they see a plan like this when the system decides on making changes by services. The plan lists the steps that have already been performed, so you can see what has happened so far for this service. Second, it shows what is currently happening. For example, which zone are we currently upgrading for this service, and how far are we in that upgrade? Then finally, there's a list of changes that will be applied later, after the current change has been applied - which means that it's entirely predictable for the engineer what's going to happen throughout this rollout.

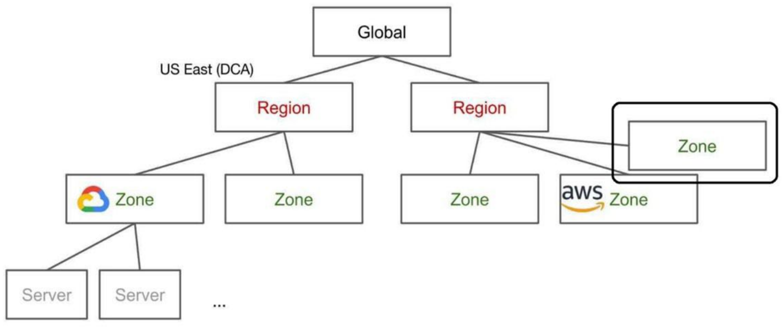


peloton Peloton job publican production canary upgrading
slack notified slack channel publican events
time done
umonitor Monitoring 15 alerts

Let's Add a New Zone

The one thing that we wanted the Up system to solve for us was to make our backend engineers not care about the infrastructure, and specifically the topology of the underlying infrastructure. As a specific goal, we wanted it to not matter to engineers if we add or remove a zone. If I have my regions here, and I'm adding a new zone to an existing region, it looks like the drawing below.

The infrastructure team will set up the physical servers, set up storage capacity, and connect the zone physically to the existing infrastructure. The challenge is now to get 4000 service owners and 4000 services, or at least a fraction of them moved to the new zone to use this new capacity that we have in that zone. Before Up, it would be a highly manual process involving dozens of engineers to complete



wanted Up to automate that for us.

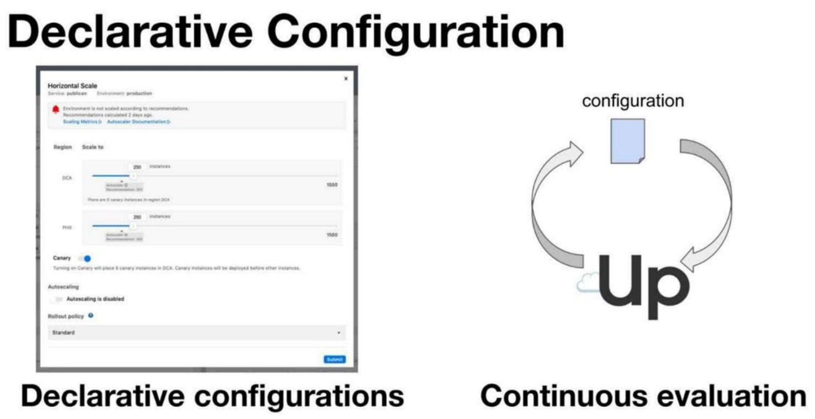
Declarative Configuration

Let's say we had a new zone as described earlier; then, the engineers will only configure their capacity in terms of regions and physical placement in the world. They will tell us that they want 250 instances in the DCA region and 250 instances in the PHX region. Moreover, they can tell the deploy system some basic stuff about their dependencies to other services and whether they want to use canary for their service. It then becomes Up's responsibility to look at this configuration and continuously evaluate whether the current placement and the current configuration of the service is the right one for each. Up will constantly compare the current topology of the infrastructure to these declarative service configurations and figure out how to place this service optimally.

With this configuration and continuous evaluation loop, what happens in our system when we add a new zone? First, the Up system will automatically discover a better placement; there is a new zone available with much more capacity than the existing zones for some specific service. Then after evaluating these constraints and deciding that there's a better placement, we have Balancer that will kick off migration from one zone within that region to another zone. Engineers no longer have to spend their time manually moving these services since our Balancer does that for them automatically.

Summary

This article told you about our journey from a small-scale company where engineers managed individual servers to the zone-based abstraction of Micro Deploy, where we could automatically manage servers. Service management overall was still a task for our engineers to maintain daily. Then finally, to our new Up system, where we have automated everything at the regional level. You can safely roll out 5,000 times to production per week, and you can manage a system of such a crazy scale as the Uber backend has become. The key to getting this to work in practice is automation. Its abstractions at a level that allows you to perform the management tasks that the engineers would otherwise have had to perform manually. This means that we can leave service management entirely up to machines, both in terms of placements, deciding on host providers, and scaling the service.



Horizontal Scale
Service: publican Environment: production

Environment is not scaled according to recommendations.
Recommendations calculated 2 days ago.
Scaling Metrics Autoscaler Documentation

Region Scale to
DCA 250 instances 1500
There are 3 canary instances in region DCA
PHX 250 instances 1500

Canary
Turning on Canary will place 5 canary instances in DCA. Canary instances will be deployed before other instances.

Autoscaling
Autoscaling is disabled

Rollout policy
Standard

Submit

Improving Speed and Stability of Software Delivery Simultaneously at Siemens Healthineers

by **Vladyslav Ukis**, Software Development Lead

Driving a software delivery transformation in the healthcare domain

In this article, we focus on the software delivery process at Siemens Healthineers Digital Health. The process is subject to strict regulations valid in the medical industry. We show our journey of transforming the process towards speed and stability. Both measures improved at the same time during the transformation, confirming research from the “Accelerate” book.

Domain

Siemens Healthineers is a medical technology company with the purpose of driving innovation to help humans live healthier and longer. Within Siemens Healthineers, the teamplay digital health platform is the enabler of digital transformation for medical institutions with the goal of turning data into cost savings and better care. The platform provides easy access to solutions for operational, clinical and shared decision support. It provides a secured and regulatory compliant environment for integrating digital solutions into clinical routines fostering cross-departmental and cross-institutional interoperability. Moreover, the platform provides access to transformative and AI-powered applications for data-driven decision support - from Siemens Healthineers and curated partners.

To date, there are more than 6.500 institutions and 32.000

systems from 75 countries connected to the platform. This makes more than 30 millions patient records accessible across institutions. The platform is open for SaaS and PaaS partners alike. SaaS partners make their existing applications available through the teamplay digital marketplace. PaaS partners develop new applications and services leveraging teamplay APIs.

The teamplay platform is cloud-based. It is built on top of Microsoft Azure, with privacy and security by design and default. The speed and stability of software delivery are central to teamplay. In 2015, the speed and stability were insufficient. With this insight, the transformation of the software delivery process at teamplay began the same year. The goal of the transformation was to make the software delivery faster and more stable. In order to achieve the goal, a large number of people, process, technology and regulatory changes were implemented over the years.

Transformation roadmap

As part of the transformation process, a whole host of new methodologies were introduced: HDD, BDD, TDD, user story mapping, pairing, independent deployment pipelines, Test DSL, SRE and Kanban. These are described in detail in a previous InfoQ article, "Adopting Continuous Delivery at teamplay, Siemens Healthineers". The adoption and "stickiness" of the methodologies differed by team. The following picture maps the major milestones of the transformation over time.

In 2015 the need for transformation became apparent. As a nascent platform in the enterprise, delivered based on the enterprise-wide regulatory quality management system (QMS) for both hardware and software products, we were challenged by the product speed and stability demand we could not meet. The product owners were entering the digital services market totally new to the company at the time. There was no knowledge available as to which services would resonate with the users, which ones the users would be willing to pay for and which feature sets would be most valuable.

Thus, the need for fast experimentation with ideas turned into software was high. Fortnightly or monthly software releases, and immediate hotfixes on-demand, would be welcome by the product owners. This was far from the software delivery the organization was set up for doing. It was obvious that the changes to the QMS would require significant expertise from the regulatory department. We started a long-term initiative towards making the QMS more lean. Internally in R&D, we increased our emphasis on automated testing.

In 2016, we started the BDD movement. This was done as part of the automated testing improvements. It had a broad impact on requirement specification, automated testing, test implementation, test reporting and understandability of test results by all roles. Whereas in the past requirements were big, the introduction of BDD forced the product owners to break them down into rather small user stories. Each user story started being broken down even further by the entire team into a set of small BDD scenarios (specification by example using the Given / When / Then statements).

The teams welcomed these changes as they addressed a long-term developer concern that the requirements were too big and bulky to implement in a short time frame. Smaller requirements led to smaller automated tests. Smaller automated tests led to more stable automation. Despite these great and necessary improvements, the overall speed of transformation was rather slow. In terms of QMS changes, we performed an analysis of how the reduction of the number of roles, deliverables, activities and workflow breaks could be done while still maintaining the required regulatory compliance.

In 2017, we brought in Continuous Delivery consultants to speed up the transformation. Dave Farley from Continuous Delivery Ltd. provided strategic consulting as well as training for managers, product owners, architects and developers. Many consultants from Equal Experts Ltd. worked alongside our product owners, architects and developers at all locations in order to jointly deliver features using many methods and techniques new to our teams. Specifically, the application of BDD, TDD, user story mapping and pair programming was the focus during the consulting activities.

By co-working with our teams, the consultants showed our developers, architects and product owners firsthand how to work in new ways, implement independent deployment pipelines, put in place initial observability, etc. In addition, we brought in medical QMS consultants from Johner Institute GmbH to discuss our analysis of QMS changes and confirm that the changes could be done while preserving regulatory compliance.

In 2018, we continued working with consultants adopting the Continuous Delivery ways of working in a deeper manner. This time it was not about introducing new methods, but rather ingraining the methods introduced before into the daily lives of teams and team members on a sustainable basis. In the spirit of the Japanese martial art concept Shu-Ha-Ri that describes three stages of learning on the path to mastery (Shu - follow the master, Ha - learn from other masters and refine your practice, Ri - come up with your own techniques), we transitioned from the Shu to Ha stage of learning. The goal was to embed the new ways of working to the point where the involvement of consultants would no longer be necessary to sustain the new practice. We reached a stage where Continuous Delivery ways of working became the standard for all new digital health products. On the regulatory side of the transformation, we brought in the BDD-based requirement engineering officially into the regulatory QMS.

In 2019, we made the first QMS release that enabled Continuous Delivery ways of working in the teams. The tools for QMS were released alongside. For requirement engineering, we validated the product "Modern Requirements for Azure DevOps" in a formal way using a validation plan and associated tests. It streamlined the requirement baselining, requirement review process and traceability of requirements.

For regulatory reporting purposes, we implemented our own tool dubbed "QTracer". Also, this tool got validated in a formal way using a validation plan and associated tests. The combination of the new QMS and associated tooling enabled the teams to make regulatory-compliant releases more efficiently with reduced regulatory overhead.

The first signs of the overall impact of the transformation were observed in the stability of delivery. The production deployment failure rate for all deployments done in the year fell by a factor of 2 compared to the previous year.

In 2020, the breakthrough of the transformation became possible. The production deployment lead time for all deployments done in the year was reduced by a factor of 2,4 compared to the previous year. At the same time, the production deployment failure rate for all deployments done in the year fell by a factor of 1,2 compared to the previous year. More details and corresponding graphs are available later in the section **Speed and stability improve together**.

In 2021, joint improvements of speed and stability of software delivery continued. The production deployment lead time for all deployments done in the year so far got reduced by the factor of 2,1 compared to the previous year. At the same time, the production deployment failure rate for all deployments

Easy transformation wins

Although the transformation has been a long and difficult process, there were some easy wins along the way. These are listed in the table below.

Change	**Explanation**
From Scrum to Kanban process	Before the transformation, our teams were required to work according to the Scrum process. At some point during the transformation, the teams were given the choice of selecting the Scrum or Kanban process. Within a few weeks, the majority of our teams voluntarily switched to Kanban. The teams enjoyed the freedom provided by the Kanban process: just-in-time backlog item grooming, just-in-time functionality demonstration when the work on a backlog item got finished, and any time prioritization of the backlog by the product owner except for the backlog items currently in work. Kanban is used by most of our teams to this day.
From big requirements to user story identification using user story mapping	A long-standing concern of developers was that requirements coming to the teams were too big. They took a long time to implement and were difficult to test. The introduction of user story mapping addressed this concern. It provided the teams with a structured way to break down big requirements into small user stories. Additionally, it enabled all the team members to be part of the user journey, as well as release planning, and discussions from the beginning. The teams welcomed the methodology and mastered it over time. Today, the user story mapping is the default method for breaking down requirements at teamplay.
From big requirements to BDD scenario specifications by product owners	Breaking down requirements into user stories is done using the user story mapping. Further breakdown of user stories can be done using BDD scenarios. This change was welcomed by the product owners as it allowed them to convey to the developers what they wanted to get implemented using examples. This is the standard practice in teamplay teams to this day. What remains challenging is the involvement of the entire team in the definition of BDD scenarios. This is very important to get a set of scenarios from different angles: functional, operational, security, performance, data protection, regulatory, etc. The richer the set of scenarios, the deeper the understanding of the user story by the team and the greater the test coverage, resulting in a better quality for the users.
From a giant pipeline deploying several products to the idea of having an independent deployment pipeline per product	When strategizing for Continuous Delivery at teamplay, we envisioned each product to be independently releasable. Therefore, an independent deployment pipeline per product was necessary to be implemented. This idea caught on very fast because the teams were suffering a lot from a giant pipeline deploying all products together only once a day. Being able to deploy independently became a movement in the organization. However, the implementation of the idea was challenging as the teams lacked the knowledge and experience of implementing independent deployment pipelines. This needed to be built over time. Today all new products are equipped with an independent deployment pipeline from the beginning.
From knowledge sharing sessions to using pairing as a means to share knowledge	At some point during the transformation the teams were given the freedom, and coaching, to try out pair programming. The practice caught on gradually. Today, pairing is a primary way of sharing developer knowledge, onboarding new developers and implementing challenging parts of the system. The practice did not catch on as a general way of doing programming.

Major transformation challenges

During the transformation, some major challenges were encountered, mitigated or addressed. These are presented in the table below.

Challenge	Explanation	Mitigation
Changing the regulatory relevant Quality Management System (QMS)	A regulatory QMS in an enterprise grows over the yearsbased on the changes in statutory laws, regulations and audit findings from all relevant geographies. This leads to the QMS containing requirements to be fulfilled by the teams whose origins are difficult to track back. Additionally, any QMS change can only be done with audit-proof explanations of the reasons and proof that the resulting process is fulfilling the laws and regulations in an equivalent way. Taken together, these aspects lead to the QMS managers being reluctant to make QMS changes due to the fear of exposing the organization to new audit findings.	• We tried the following to mitigate the challenge: • Taking the QMS managers for visits to companies that implemented Continuous Delivery despite operating in regulated industries • Hiring consultants experienced in medical device regulations to co-design QMS changes with our QMS managers • Educating the QMS managers in Continuous Delivery principles and methods
Changing people's mindsets	The transformation required a thorough rethink of all software delivery aspects from all roles in the organization. As many people, teams and groups were changing many technical, organizational and process aspects at the same time, it proved challenging toavoid intransparency, ambiguity and obscurity of changes.	We tried the following to mitigate the challenge: • Providing a high level roadmap for changes • Explaining the benefits of changes and the required investments • Having one-on-ones with selected people • Providing coaching to selected people
	Outcomes of transformation recognizable by the people not actively participating in the ongoing changes (e.g. increasing the speed of delivery) take a long time to materialise and be seen	We tried the following to mitigate the challenge • Making regular reporting presentations to leadership explaining the changes done and the outcomes achieved • Making presentations at company-wide events about the ongoing transformation and the benefits realised:
Transform while you perform	The business demand for new features has been high during the transformation years. Splitting development capacity between newfeature implementation, transformation activities and operations of existing features turned out to be very challenging. The resulting pace of transformation was rather slow.	We tried the following to mitigate the challenge: • Providing the teams with the guidelines for capacity split • Providing the teams with an understanding of gains and benefits to be reaped when investing in transformation

Architectural decoupling of products	When the transformation started, all teamplay products were tightly coupled architecturally. It took a very long time to perform the necessary architectural decouplings enabling independent product releases. The prioritisation of the architectural decouplings against new features has been a major challenge.	We tried the following to mitigate the challenge: • Bringing architectural changes into the portfolio management, increasing visibility at the organizational level • Explaining to product owners the benefits of independent product releases enabled by the architectural decouplings to be prioritised • Educating the architects about the necessary architectural decouplings, reasons for them, benefits of performing them and the outcomes that could be achieved with them • Celebrating successes when a major architectural decoupling was done • Fostering knowledge sharing between the teams putting them in a position to better estimate the time it would take to re-architect services
Making TDD a default software development method	Although TDD is a fundamental Continuous Delivery method, making TDD a default software development method turned out to be a major challenge. The introduction of TDD was promising. Working alongsideconsultants experienced in TDD was fruitful. However, as time went by, the number of teams and individuals applying TDD dwindled. We did not manage to get TDD practiced in the ongoing programmingwork of the teams.	We tried the following to mitigate the challenge: Encouraging teams to apply TDD to newly developed code Encouraging teams to refactor code being worked on to prepare it for future TDD work Offering hands-on code craftsmanship classes on demand

done in the year so far fell by a factor of 1,7 compared to the previous year.

More details and corresponding graphs are available later in the section **Speed and stability improve together.**

Overall, a long-term process change involving people, process, technical, organizational and regulatory changes will be challenging by definition. A strong vision and courageous leadership is required to grow and sustain the momentum over time. Showcasing and celebrating successes on the journey goes a long way in keeping the people highly motivated in order to weather the emerging challenges.

Due to a large number of options different aspects of the transformation can take, it is nearly impossible to plan the process. A viable alternative to planning is to state high level business goals for the

Major transformation opportunities

Although the transformation has been a lengthy and demanding process, it held the key to major opportunities. These are listed in the table below.

Change	Explanation
Enable fast experimentation with business ideas	Operating in the rather new digital health market, there is little prior knowledge of product categories and specific products that would resonate with the market. The same holds true for the willingness to pay and competitive price points for products and services. In this environment, experimentation with business ideas is the central point of business activity. The ability to explore the markets using fast business experiments bears a great competitive advantage. In the healthcare domain, this translates to the product demand of roughly fortnightly to monthly releases. For medical device software products, monthly releases are feasible after initial regulatory submission as long as the intended use of the product remains unchanged. For non-medical device software products, releases every 2-4 weeks are feasible also for companies governed by heavy non-medical regulations such as ISO 9001 for example.
Change the culture of the organization towards generative	There is a high degree of collaboration between different groups and teams in the product delivery organization required to make frequent releases. This naturally leads to positive changes in organizational culture towards generative. A generative culture is performance-oriented. It is characterised by high cooperation, risk sharing, bridging encouragement, inquiry into failure, etc. These characteristics can be supported organically when a product delivery organization is transforming towards faster software releases. The generative culture was found to drive higher software delivery and organizational performance by the research from the book "Accelerate".
Change the software delivery towards speed, stability and reliability at scale	Fast and stable software delivery necessitates efficiency in processes such as requirement engineering, development, testing, deployment, regulatory documentation generation, release and operations. Not only is it possible at the team level; rather, the efficiency can be scaled up at the organizational level. Operating efficiently maximizes the time available for the actual iterations on the products.

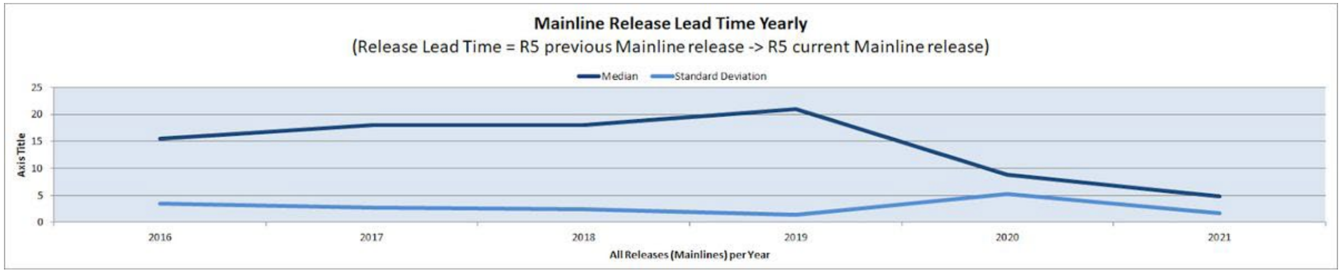
Speed and stability improve together

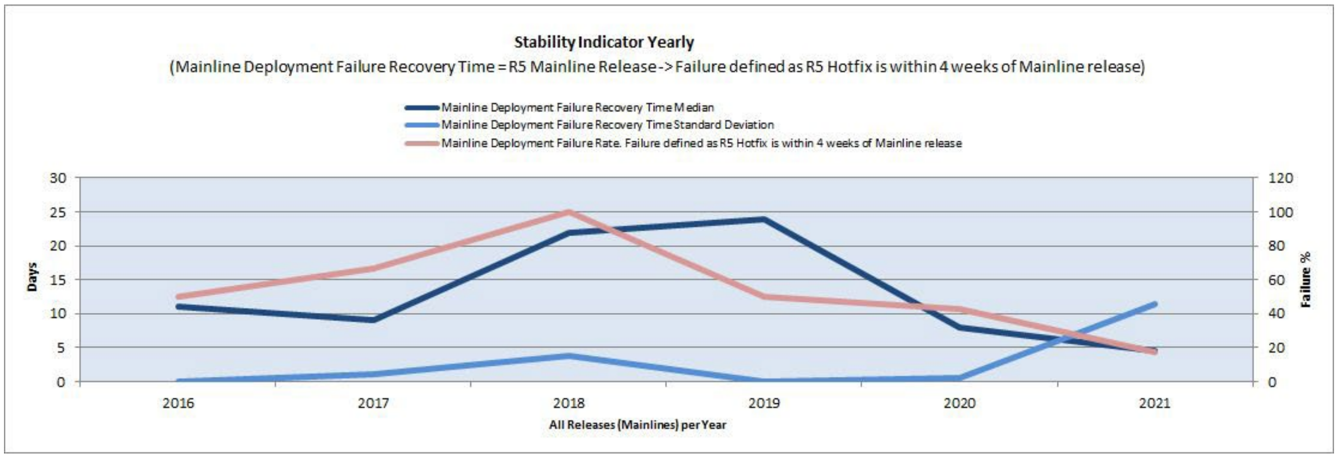
As shown in the previous chapter, speed and stability improved at the same time during the transformation. This can be seen in detail based on the speed and stability indicators in the pictures below.

The production release lead time trend over the years of transformation shows that for a long time, between 2016 and 2019, the lead time did not decrease. On the contrary, in 2019 the lead time increased despite all the transformation efforts. However, since 2020 there has been a significant drop in the lead time: 2.4x from 2019 to 2020 and, on top of that, 2.1x from 2020 to 2021.

That is, there has been a compound 5x production lead time decrease since 2019!

The production deployment failure rate trend over the years of transformation is shown in the picture below. It shows that for a long time, between 2016 and 2018, the failure rate did not decrease. On the contrary, in 2018 it peaked at 100% despite all the transformation efforts. However, since 2019 there has been a steady drop in production deployment failure rate: 2x from 2018 to 2019, then 1.2x from





transformation and quantify them appropriately. Then let the teams define intermediate milestones to explore their paths towards the goals.

2019 to 2020 and then 1.7x from 2020 to 2021. That is, there has been a compound 4x production deployment failure decrease since 2018!
The two pictures above demonstrate that since 2020, the speed and stability of software delivery has been improving simultaneously. The production deployment lead time and failure rate trend downwards at the same time. That is, the software delivery is getting faster and more stable at the same time.

The following reasons led to speed and stability improving only after several years into the transformation:

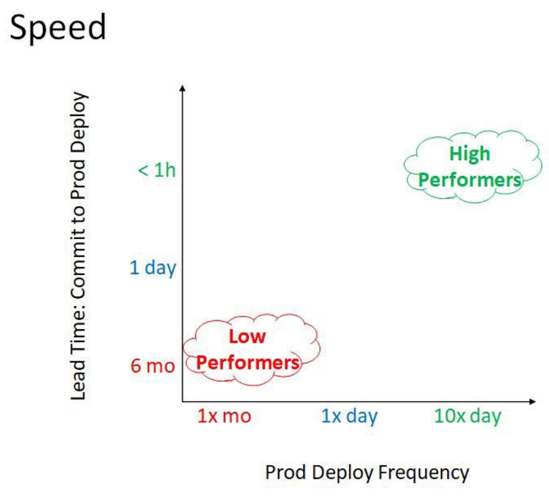
The knowledge of Continuous Delivery methods was not there in the organization when the transformation began. It took a significant knowledge ramp-up and experimentation effort in order to build up the knowledge and arrive at a set of methods that are suitable for and accepted by the organization.

“Transform while you perform”: the transformation took place in parallel to a significant implementation of new customer-facing features using new technologies. This was a business necessity. The feature implementation took a significant share of development bandwidth leaving only the rest for transformation activities.

Technical changes such as architectural decoupling, creation of independent deployment pipelines per team and test refactorings took a long time to be put in place due to development bandwidth being shared between transformation and feature development activities.

Confirming research from the “Accelerate” book

The book “Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations” by Nicole Forsgren, Jez Humble and Gene Kim presents research on software delivery based

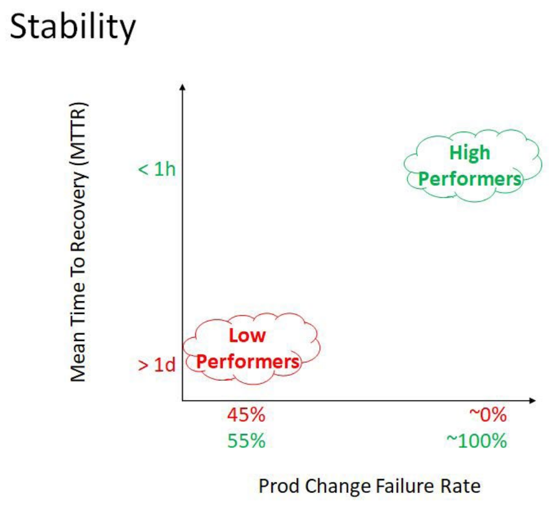


Attribution: image by Randy Shoup in the talk "Moving Fast at Scale"

on a survey of more than 400 companies around the world. The research from the book was succinctly summarized by Randy Shoup, in his talk, "Moving Fast at Scale" using the next two pictures.

As part of the research, two clusters of companies were uncovered: high and low performers. In terms of speed of software delivery, high performers deploy about 10 times a day with a lead time of less than an hour. The low performers deploy about once a month with a lead time of about six months.

Interestingly, there are not a lot of companies in-between the two clusters. That is, the companies surveyed tend to be either high or low performers. Even more interesting to see is that the clusters of high and low performers based on the speed of delivery, shown in the picture above, are the same for the stability of delivery, illustrated in the picture below!



Attribution: image by Randy Shoup in the talk "Moving Fast at Scale"

High performers nearly never fail production deployments.

If they do, the recovery is brought about in less than an hour. Low performers, on the other hand, fail nearly half of production deployments. The recovery from a production deployment failure occurs in more than a day.

Due to the clusters of high and low performers being the same for speed and stability, it can be concluded that in software delivery, speed and stability go together. In fact, the faster the software delivery, the more stable it becomes.

This might sound counterintuitive. However, the rigorous research from the book proves it. During our software delivery transformation, we could see firsthand how speed and stability of delivery improved at the same time. That is, our transformation could, indeed, confirm the research findings from "Accelerate".

Transformation retrospective

Looking retrospectively at the transformation, the following lessons can be highlighted.

- The transformation was not driven in a data-driven way from the beginning. Only later in the process the Continuous Delivery indicators of speed and stability were established at the organizational and team level. We can recommend establishing the indicators from the start of transformation. It can be done based on the book "Measuring Continuous Delivery" by Steve Smith detailing the definition and implementation of the speed and stability indicators for all stages of a deployment pipeline. The speed indicator is structured as production deployment lead time and production deployment frequency. The stability indicator is structured as production deployment failure and production deployment recovery time.

Having the indicators at the organizational and team level caters for a faster and deeper alignment of people in the organization regarding the goals to be achieved through the transformation. Moreover, it enables the teams to set their own intermediate speed and stability goals in a data-driven way. Most importantly, the indicators allow a proper application of the improvement kata throughout the organization formalizing continuous improvement as a general way of driving the transformation:

- The introduction of SRE and operability in general was done in later years of transformation. We can recommend weaving these aspects earlier in the transformation process. This caters for growing the "you build, you run it" attitude, processes and tools more organically during the transformation, rather than being perceived as yet another big transformation step once Continuous Delivery has been established. Moreover, the SRE indicators of reliability can be used in the improvement kata on a regular basis.

For instance, an availability indicator can be established as service availability rate and time to re-establish availability on loss. Long-term and intermediate goals can be defined and iterated towards under the guidance of the availability indicator.

- In terms of working in new ways, we learned that providing people with both knowledge of new methods and coaching by the people proficient in using the methods is a powerful combination. That is, when allocating time for people to learn, just providing access to learning materials or training is not sufficient to change habits. By the same token, just providing access to coaches is not sufficient to build enough understanding for reasons to be coached.

Step	**Description**	**Application of speed and stability indicators**
1	Get the direction	Expressed as long-term speed and stability goals
2	Grasp the current condition	By looking at the current speed and stability indicator values
3	Establish your next target condition	Expressed as the next set of speed and stability goals
4	Conduct experiments to get there	Make technical, process, organizational etc. changes and measure the impact of changes using the speed and stability indicators

Consequently, coaching alone does not change habits either.

By contrast, a combination of initial learning and a prolonged period of application of what was learned under the guidance of an experienced coach tends to change habits at the individual and team level. The initial learning establishes an understanding of the underlying reasons for, benefits of and contexts for the application of new methods. The subsequent application of the new methods paired with an experienced coach ingrains the new methods into daily work with a frequent and powerful feedback loop adjusting the process in-the-small.

- The reductions of production deployment lead time are going to be made by decoupling services in terms of architecture, test, deployment, regulatory compliance, release and operations. With the growing number of independent services, a lightweight governance is required in order to centrally assert organization-wide best practices, agreements and regulations while leaving as much independence as possible to the teams.

The establishment of the governance can beneficially be done for example using an opinionated platform that codifies the most important control points. These activities should be part of the transformation rather than something to be established later once the transformation has shown signs of success. This way the necessary governance can also be grown organically, beginning with very small steps such as the naming of services and environments.

We started the governance activities late in the transformation and can recommend growing the practice as the first independent services are being built up.

Summary

Accelerating software delivery is a long-term transformation process. The process can be steered effectively using speed and stability goals, and measured using respective indicators. The research from the popular book "Accelerate" says that speed and stability go together. Following this, both measures should improve at the same time. The software delivery transformation at the Siemens Healthineers teamplay digital health platform confirmed the research from the book. During the transformation, speed and stability improved at the same time.

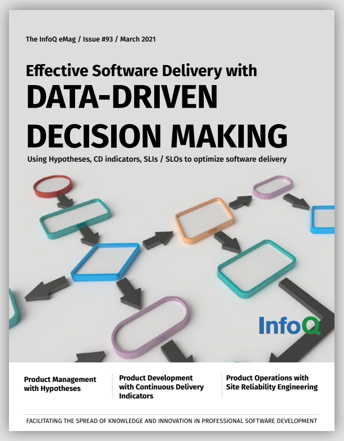
Acknowledgments

We would like to acknowledge the following people who were instrumental in driving the software delivery transformation at the Siemens Healthineers teamplay digital health platform: Thomas Friese, Carsten Spies, David Schottlander, Fabio Giorgi, Frank Schneider, Frank Stanischewski, Philipp Guendisch, and many others.

Furthermore, we would like to acknowledge the following Continuous Delivery consultants whose help was invaluable during the transformation: Dave Farley from Continuous Delivery Ltd as well as Neha Datt, Ryan Bayly, Marcel Britsch, Keerthana Jayaram, Louis Abel, and many others from Equal Experts Ltd.

Finally, thanks go to Akshith Rai for maintaining the Continuous Delivery indicators at teamplay. Major transformation challenges.

Read recent issues


The InfoQ eMag / Issue #93 / March 2021

Effective Software Delivery with
DATA-DRIVEN DECISION MAKING

Using Hypotheses, CD indicators, SLIs / SLOs to optimize software delivery

InfoQ

Product Management with Hypotheses | Product Development with Continuous Delivery Indicators | Product Operations with Site Reliability Engineering

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

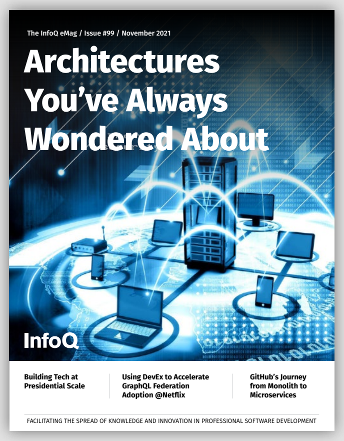
The InfoQ eMag / Issue #100 / November 2021

DevSecOps: Shifting Left in Practice

InfoQ

Virtual Panel: DevSecOps and Shifting Security Left | Failing Fast: the Impact of Bias When Speeding up Application Security | Lessons Learned from Reviewing 250 Infrastructures

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT


The InfoQ eMag / Issue #99 / November 2021

Architectures You've Always Wondered About

InfoQ

Building Tech at Presidential Scale | Using DevEx to Accelerate GraphQL Federation Adoption @Netflix | GitHub's Journey from Monolith to Microservices

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

Effective Software Delivery with Data-Driven Decision Making

This eMag on Data-Driven Decision Making provides an overview of how the three main activities in software delivery can be supported by data-driven decision making to increase the effectiveness, efficiency and service reliability of a software delivery organization.

DevSecOps: Shifting Left in Practice

We have prepared this eMag for you with content created by professional software developers who have been working with microservices for quite some time. If you are considering migrating to a microservices approach, be ready to take some notes about the lessons learned, mistakes made, and recommendations from those experts.

Architectures You've Always Wondered About

The "Architectures You've Always Wondered About" track at QCon is always filled with stories of innovative engineering solutions. Bringing those stories to our readers is at the core of InfoQ, and this eMag is a curated collection of some of the highlights over the past year.

InfoQ

InfoQ @InfoQ InfoQ InfoQ