O'REILLY®

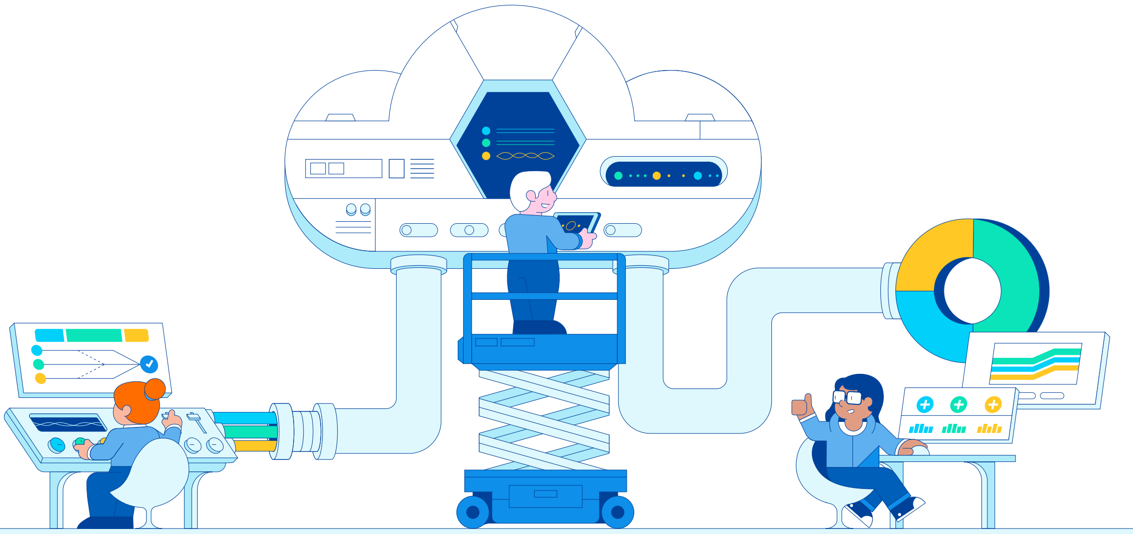
Compliments of split

Continuous Delivery in the Wild

Pete Hodgson

REPORT

Skip the hotfixes and rollbacks with Split's Feature Delivery Platform.



Manage
feature flags

Monitor
release errors

Experiment
with A/B tests

Confidently release features as fast as you develop them. Keeping your customers (and engineering teams) happy.

Try it for free at Split.io/signup

split

Continuous Delivery in the Wild

Pete Hodgson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Continuous Delivery in the Wild
by Pete Hodgson

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: John Devins
Developmental Editor: Sarah Grey
Production Editor: Nan Barber
Copyeditor: Christina Edwards
Proofreader: Nan Barber
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

February 2020: First Edition

Revision History for the First Edition
2020-02-11: First Release

978-1-492-07767-1

[LSI]

Table of Contents

CHAPTER 1

Introduction

Software companies are under constant pressure to deliver features to their users faster, while simultaneously maintaining (or improving) the quality of those features. This may seem like an impossible task, but many organizations have discovered that it is in fact achievable, using the practices of Continuous Delivery.

What is Continuous Delivery?

Continuous Delivery is a set of technical practices that allow delivery teams to radically accelerate the pace at which they deliver value to their users. The core tenet of Continuous Delivery is keeping your codebase in a state where it can be shipped to production at any time. By working in this way, you can quicken the tempo of production changes, going from infrequent, big, and risky deployments to deployments that are frequent, small, and safe.

NOTE

The book *Accelerate*, by Nicole Forsgren, PhD, Jez Humble, and Gene Kim, uses rigorous scientific methods to confirm many of the benefits of Continuous Delivery, based on several years of the *State Of Devops Survey*. The book shows that organizations practicing Continuous Delivery have better software delivery performance, which in turn drives greater organizational performance. In other words, Continuous Delivery leads to better organizational outcomes.

Continuous Delivery in the Real World

A lot of the discussion around Continuous Delivery focuses on the cutting-edge practices advancing the state of the art. In this report, we will instead look at how a variety of organizations have implemented Continuous Delivery in the real world.

We'll look at some of the common approaches these organizations have found helpful. We'll also see how there are multiple ways to achieve the same goal, depending on the organizational context. My hope is that after reading this report you'll come away with some actionable ideas for how to implement, or improve, Continuous Delivery within your own organization.

Research Methodology

To understand how people have implemented Continuous Delivery in the real world, I conducted in-depth interviews with a variety of organizations with a rapid software delivery tempo, deploying code into production at least daily. Some of these organizations have practiced something akin to Continuous Delivery from the start, while others have migrated to Continuous Delivery practices over the last few years.

Throughout the rest of the report, I'll refer to the organizations interviewed as "participants." The more interesting participants are described as follows:

Payment Processor
: Founded in 2009, with around 1000 engineers and *Service-Oriented Architecture* (SOA) consisting of around 500 services.

Automotive Marketplace
: Has had an online presence since the mid-1990s, with around 200 engineers and an architecture that's migrating from monolithic web apps toward SOA, with currently around 300 services.

Online Retailer
: Founded in the 1950s, with around 20 engineers working on a monolithic web application, which is just starting to move toward SOA.

Food Delivery Service
: Founded in 2012, with around 30 engineers working on an SOA of 15 to 20 services.

Healthcare Provider
: Founded in 2007, with around 100 engineers working on a monolithic web application.

Print and Design Service
: Founded in 2004, with around 50 engineers who are partway through a migration from a monolithic web application to SOA, with currently around 50 services.

Online Realtor
: Founded in 1995, with around 450 engineers working on an SOA made of up of both large monolithic systems and smaller services.

Financial Services Startup
: Founded in 2007, with around 800 engineers working on an SOA with a large number of small services.

We'll learn more about how each of these organizations has approached the challenges of Continuous Delivery throughout the report.

The Path to Production

To understand how each participant does software delivery, I asked them to describe the "path to production" for a small user-facing feature. While the organizations vary broadly in terms of architecture, industry, and organization size, there is a striking consistency in the mechanics of how each organization has implemented Continuous Delivery.

Across all the organizations that I surveyed, the path to production looks something like this (Figure 1-1):

- Engineer implements feature
- Change is reviewed and merged to master
- Change is validated via automated tests
- Change is automatically deployed to a shared integration environment

- Brief exploratory testing of change is done (if warranted)
- Change is deployed to production
- Controlled rollout of the change to users happens (if warranted)

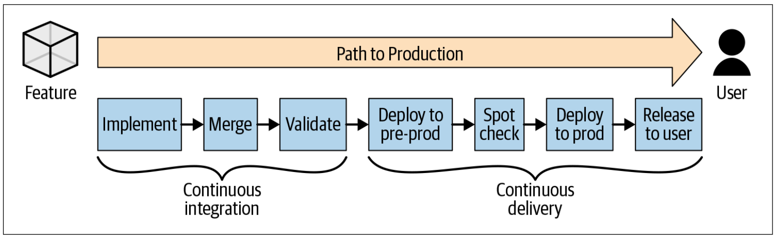


Figure 1-1. The path to production: how a feature moves from an engineer's keyboard into a user's hands

The first half of this path—building the feature, merging it, and performing automated validation of the merged code—constitutes the practice of *Continuous Integration*. The second half—flowing the changes that make up the new feature through to production in a safe, consistent way—constitutes *Continuous Delivery*.

We'll explore how different organizations implement this path to production over the course of this report.

Continuous Delivery Versus Continuous Deployment

Every organization I surveyed has an automated deployment from master[1] to a shared preproduction environment. Any change that lands on master will automatically be deployed to the preproduction environment, as long as it passes Continuous Integration validation.

This approach is taken even further at the Online Realtor and the Automotive Marketplace, where some teams automatically promote changes to their *production* environment, with no human intervention necessary, as long as it passes further automated validation. This is an example of *Continuous Deployment*—every valid change

1 Going forward, I'll use "master" as a shorthand for the main development branch where a team integrates their work, since that's the most common nomenclature in today's git-centric world. This is sometimes referred to as "trunk" (i.e., in Trunk-Based Development).

landing on master will automatically flow all the way through to production.

However, most of the organizations that I talked with avoid full-on Continuous Deployment. Instead, they institute some sort of *manual gate*, requiring an engineer to explicitly promote their changes into production from a preproduction environment. This wouldn't be considered Continuous *Deployment*, but it is still a form of Continuous *Delivery*.

CHAPTER 2

Branch Management and Code Review

Continuous Delivery builds upon the practice of Continuous Integration, which is defined by the frequent integration of work into a shared branch (with "frequent" often interpreted as "at least daily."[1]) All participants I interviewed adhere to these principles, avoiding long-lived feature branches almost entirely.

However, many participants are *not* meeting the strict definition of Continuous Integration. Several reported that feature branches have a typical lifetime of a few days to a week before being integrated into their shared master branch.[2]

GitHub Flow

The majority of participants use a variant of the *GitHub flow* branching model.[3] Engineers create a short-lived *feature branch* for each change they are implementing, and create a pull request (or merge request) once their work is ready to be integrated into master.

1 *Accelerate*, Chapter 4.

2 All participants were using Git for version control.

3 GitHub flow is defined here: *https://oreil.ly/ocvBZ*. While this definition specifies that changes in a feature branch are deployed to production *before* being merged to master, I have yet to encounter an organization that actually does this—besides GitHub themselves, presumably.

That pull request typically also serves as a mechanism for coordinating code review. Once a change has been approved it is merged into master.

Participants using pull requests also typically leverage their Continuous Integration infrastructure to run premerge validation, running the same types of automated checks against the feature branch as they would run against master once the branch is merged. Feedback from these automated checks is then available for reviewers of the change within the pull request UI.

Trunk-Based Development

Some participants forgo the use of branches entirely and work directly on master, a practice known as *Trunk-Based Development*. Engineers at the Online Retailer explained that they simply make their changes directly to their local master branch, and push to the shared remote repository once their changes are ready.

Participants that primarily use Trunk-Based Development do still use short-lived feature branches on occasion. The Online Retailer described using them when a change was risky, or being made by a junior engineer. At the Automotive Marketplace, they are used when an engineer from one team is making changes to a codebase owned by another team. In that case the engineer would create a feature branch and use a pull request to solicit feedback from the owning team before landing the change into their master branch.

Minimal Branches

Across all participants there was universal agreement that long-lived branches are detrimental to Continuous Delivery practices. Some participants that had recently moved to Continuous Delivery were still in the process of moving away from relying on long-lived branches.

What were these legacy long-lived branches used for? At the Healthcare Provider and the Print and Design Service, there were some lingering instances of team integration branches, a shared branch used by engineers on a team in order to collaborate on a hairy code change that would have destabilized their master branch. The Online Retailer had another use case, where a long-lived release branch was still part of their release engineering process. In all cases, participants were experiencing pain from these practices. Team integration branches are inevitably (and ironically) hard to integrate with the shared master branch. Release branches have a variety of drawbacks, as we'll discuss next.

Cutting a Release Versus Promoting a Build

Traditionally, teams have used branches to orchestrate a software release. The first step in getting a set of changes into production might involve "cutting a release branch" off of the master branch. That release branch freezes the version of the codebase that will be deployed to production (often referred to as a release candidate). This release candidate is isolated from potentially destabilizing changes, which will continue to land on the master branch while the various phases of a production release take place against the release branch.

Modern CI/CD systems provide a better alternative to release branches: the *Delivery Pipeline*. This moves the orchestration of a release out of source control and into the CI/CD system itself. A pipeline defines the various stages required to take a version of our source code, gain confidence in it, and eventually deploy it into production. These stages all operate on a static snapshot of the codebase, which provides the same type of isolation as a release branch. No matter what changes happen in version control after a pipeline starts, each stage of the pipeline always works with the exact same version of the codebase.

Because it's working against a static snapshot of our code, a delivery pipeline allows us to gain more and more confidence in that particular version of our code. As it moves through the various stages of our pipeline it is subjected to a series of automated checks as it is deployed into preproduction environments for further validation. When the change is eventually deployed into production we can be confident that what's being deployed is the same code that has successfully surmounted the obstacle course of quality checks put before it earlier in the pipeline. This is in contrast to a release branch, which often receives additional small changes (configuration updates, bug fixes, and so on) as a release candidate moves through the release process.

Another advantage of delivery pipelines is that they force a team to automate the various operations involved in a release. With a traditional release branch approach, there is often a series of manual steps involved in a production release—cut a branch, update configuration files to indicate that this is a release build, and merge any hot-fix changes back into master. These manual steps add additional friction to each release, as well as introduce a high risk of human error—configuration changes inconsistently applied, bug fixes lost after a release, and so on.

For these reasons, most participants use delivery pipelines, rather than release branches. The few still using release branches, such as the Online Retailer and the Healthcare Provider, are actively moving toward the use of delivery pipelines.

Most CI/CD systems also provide a manual gating feature, which prevents a pipeline from moving on to the next stage until it receives approval to do so from a human operator. This feature is often used to pause a pipeline right before a release candidate is deployed to production. For any successful run of the pipeline, an engineer can opt to "push the button" and deploy a change to production, typically after a quick spot-check of the change in a pre-production environment. This act of approving a release candidate to move to the next environment is often referred to as "promoting" the build. The presence of a manual gate is what distinguishes Continuous Delivery from Continuous Deployment, as discussed in Chapter 1.

Code Review

All participants using short-lived feature branches also use pull requests to orchestrate their code review process. Several require a change to be reviewed before it can be merged to master—a regulatory requirement for some—although the majority of participants don't systematically enforce this policy. Several teams reported challenges with code review turnaround time. A delay in a change being reviewed often leads to an increase in feature branch lifetime, but also tends to decrease engineer productivity as they attempt to juggle multiple branches, working on a new change while waiting for an existing change to be approved for merge.

Participants that work directly on master are more varied in their code review practices. Teams at the Automotive Marketplace tend to conduct code review in person, prior to pushing changes to a shared master, either via pair programming or "over-the-shoulder" in-person walkthrough. However this type of premerge code review is typically only reserved for changes considered large or risky. Teams at the Online Retailer practice postmerge code review, managed via their project management tool, with review feedback captured via commit annotations in GitHub. Whether practicing pre- or post-merge review, teams working directly on master don't have a strict policy that all changes should be reviewed.

Reducing Batch Size

Engineers at the Financial Services Startup describe its delivery pipeline as being like a moving assembly line in a factory. If small changes are constantly showing up on the conveyor belt (i.e., small changes landing on master) then there is enough time to inspect each change, and the team feels comfortable with those changes flowing out to production (Figure 2-1). It is clear that teams working directly on master find it much easier to achieve this flow of small changes.

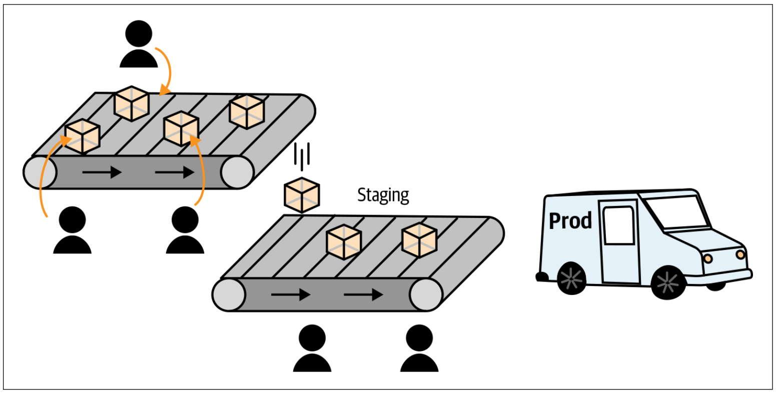


Figure 2-1. A steady flow of small changes moving along the line toward production

However, if a feature branch is allowed to live too long before merging then a large batch of changes accumulates (as shown in Figure 2-2), making it hard to inspect each change once it lands on the assembly line.

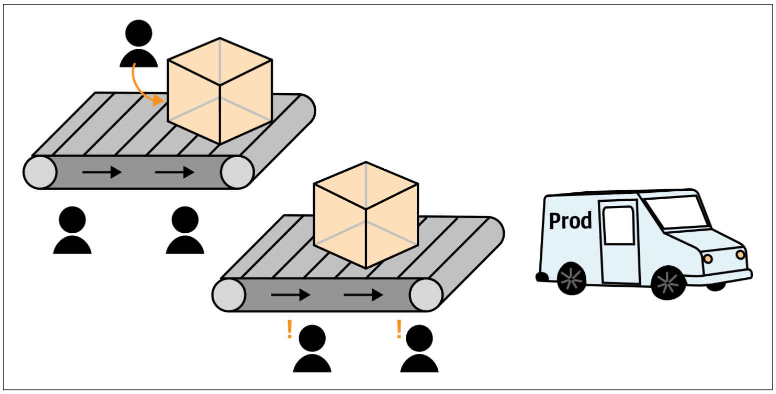


Figure 2-2. A big change landing on the line is harder to inspect

Similarly, if production deployments are held up for any reason—a production issue, or bugs found on master—then again a large set of changes will accumulate (Figure 2-3).

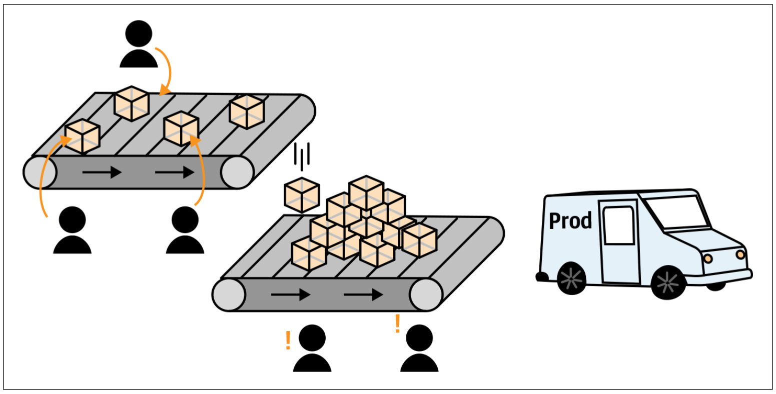


Figure 2-3. Lots of pending changes backed up in staging are also harder to inspect

Several participants explicitly identified small batch size as a key to making Continuous Delivery possible. Their software delivery processes were contingent on a steady flow of small changes into production. Given this, I asked participants what techniques they used to reduce the size of each change going out to production, while also avoiding exposing half-finished changes to end users.[4]

4 Paul Hammant has assembled an exhaustive collection of "Trunk-Correlated Practices": *https://oreil.ly/63jQp*.

Rather than releasing a feature as one large code change, teams spend time breaking a feature down into a set of smaller changes. These changes are also sequenced so that they can be built and deployed into production one by one as *latent code*—code that is tested and in production, but not exposed to users.

Incremental Feature Deployment

Let's look at a simplified example of how you might safely deploy a half-finished feature into production.

You're a product engineer for an online store, and you're working on adding a "request gift wrap" feature. This will require adding a new checkbox in the checkout UI, along with adding a corresponding new field in the backend API that that checkout UI uses, as well as further changes deep in the bowels of the order fulfillment system.

You slice the engineering work up into a set of smaller changes that will be deployed independently. You work on the backend changes first, and deploy them to production. After deployment, the backend API supports gift wrapping requests, but no users can make that request since the checkbox has not been added to the UI. This allows you to safely verify that the core functionality works in production. Once you are comfortable, you make the final change, adding the checkbox to the UI, exposing the new feature to users. If you want extra safety, you might wrap that UI change in a feature flag, a technique we'll discuss further in Chapter 4.

The Food Delivery Service uses *branch by abstraction* techniques as a way to avoid long-lived feature branches. A large internal change is implemented as latent code alongside the existing implementation, along with some internal plumbing that allows switching between the old and new implementation at runtime, typically controlled by a feature flag. Using this approach, large changes can be made incrementally on master, tested along the way, but only "turned on" once they're complete.

Interestingly, several participants shared stories of tight-knit teams working on a smaller codebase who would, at times, opt to simply declare master temporarily unstable and put production deployments on hold while working on a large feature that was tricky to break apart. The Food Delivery Service noted that in these cases a team with mature Continuous Delivery practices was opting to “know the rules well enough to break them.” While a key tenet of Continuous Delivery is that master should always be in a releasable state, these teams decided that in some cases the trade-off was worth it, as opposed to taking on the additional overhead of a full-blown branch by abstraction process.

CHAPTER 3

Running an Integrated System

In Chapter 2 we saw that participants strive for a continuous flow of small changes into production. This leads to two outcomes. First, preproduction environments become less useful. Second, engineers have to test their changes against an integrated system *before* merging those changes to master.

Continuous Delivery Demands Fewer Environments

Participants that had recently moved to Continuous Delivery, such as the Online Retailer, described a pre-CD world where engineers and testers relied on multiple fully integrated preproduction environments—environments running the full stack of software constituting the product, in a similar physical architecture to production (although often at a smaller scale). These preproduction environments are used for different use cases: developer sandboxes, integration testing, exploratory testing, showcasing, and so on.

Multiple environments were necessary because these different activities required different versions of various codebases to be integrated for inspection. For example, product manager might want to preview an upcoming release in an environment running the current *release candidate* for every codebase (a release candidate being the version that has been identified as ready for release to production, pending quality checks). An engineer might also want to work on a new integration between two services by pointing a locally running service to some feature branch version of a dependent service, running in a shared development environment. A tester might want to validate a production *hotfix*—a minor release made outside of regular release cadence in order to apply an urgent production change—by running the hotfix change against the production versions of other systems.

These myriad different versions of different systems become less important when practicing Continuous Delivery. Because the production system is changing so frequently, it's only really interesting to look at what's currently in production, or what's about to be in production. To that end, many participants report operating just two fully integrated environments: production itself and a shared preproduction environment, which I'll refer to as "staging."[1] As described in Chapter 1, the Continuous Delivery infrastructure ensures that staging always contains the latest good version from each codebase's master branch, with the versions deployed in the production environment typically lagging behind staging by a day or two at most.

Testing Changes Prior to Merge

A central tenet of Continuous Delivery is that master should always be releasable. This poses a paradox for an engineer: you want to test how the changes that you're making work when integrated with the rest of the system, but you don't want to merge those changes to master before they've been validated.

The participants resolved this paradox by:

- Running a full environment locally
- Running a partial environment locally, with stubbed out dependencies
- Running a partial environment locally, integrated against a shared environment
- Issuing a personal development environment to each engineer

1 The nomenclature for environments is rather inconsistent across organizations. I've typically seen the type of environment I'm referring to here as "stage," "staging," or "preprod."

- Allowing engineers to stand up transient development environments on-demand
- Allowing engineers to inject custom versions of a service into a shared environment

Running a Full Environment Locally

When working with a monolith (or smaller SOA systems) it can be possible to run the entire product locally on a developer workstation (Figure 3-1). Doing this allows you to assemble whatever set of versions is appropriate for the work at hand. However, as the number of services in a product architecture grows beyond a certain size this approach becomes infeasible.

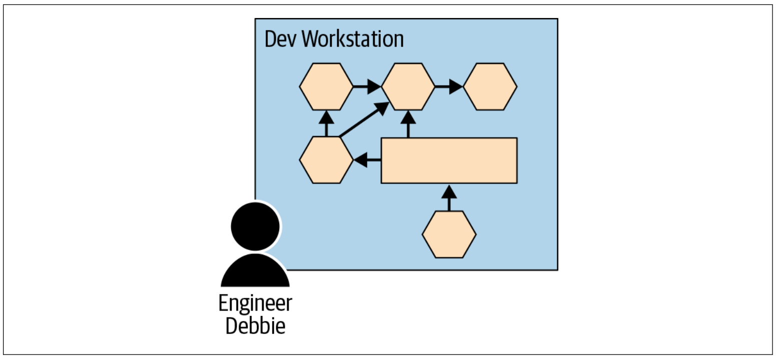


Figure 3-1. Debbie Dev running the full product stack locally

Running a Partial Environment Locally

Some participants, such as the Food Delivery Service and the Payment Processor, invested a fair amount of engineering effort in ensuring that individual services can stand up in isolation. This meant that an engineer working on a service could stand up just that service locally, or if necessary they could stand up that service plus the services it depended upon (Figure 3-2). Those depended-upon services would run in an isolated manner, preventing the entire graph of transitive dependencies from being pulled in.

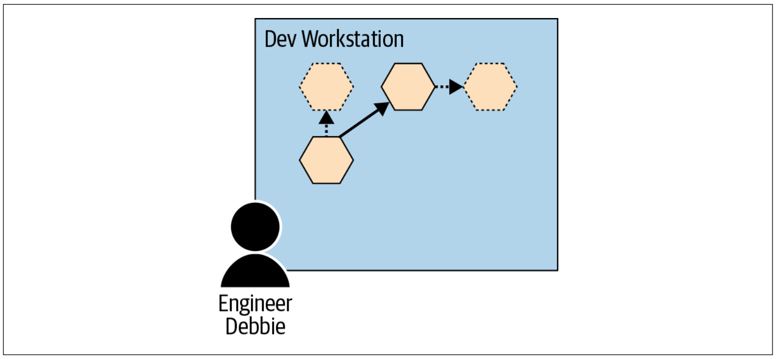


Figure 3-2. Debbie Dev running a partial stack locally

Issue Personal Dev Environments

When the Healthcare Provider's architecture became too large to run locally, they opted to instead stand up a full-stack remote development environment for each engineer (Figure 3-3). An engineer is responsible for maintaining her environment, and can deploy different versions of services in that environment using custom tooling (command-line scripts and/or a web interface). This approach involves significant management overhead, as well as a nontrivial infrastructure cost.

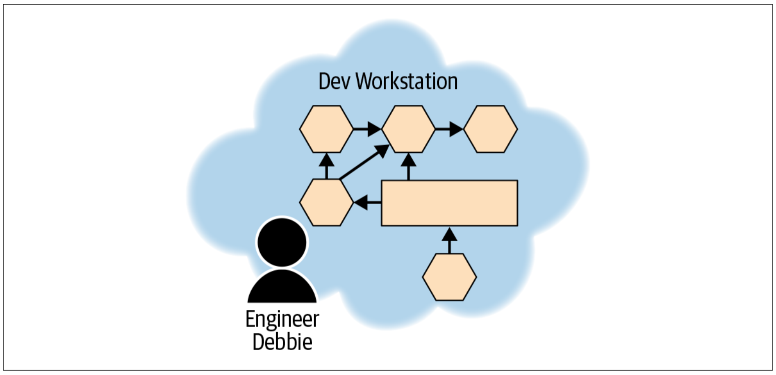


Figure 3-3. Debbie Dev's personal dev environment

Allow Transient, On-Demand Dev Environments

The Print and Design Service and the Financial Services Startup opted for an alternative approach, where engineers can do self-service provisioning of short-lived, full-stack environments, and then manage them similarly to the personal dev environments described above.

With this approach, environments are automatically torn down every night. This reduces infrastructure cost, and also reduces the amount of ongoing configuration and version drift. However, these provisioning systems also allow engineers to request a "stay of execution," which in some cases leads to the establishment of long-lived environments serving as a sort of shared team integration environment.

Allow Connecting Development Workstations to Staging

At the Automotive Marketplace, engineers can integrate a service running on their local development workstation directly into the shared staging environment (Figure 3-4). This can work well when the locally running service depends on one or more other service, but doesn't allow you to test the inverse integration, where other services depend on your locally running service.

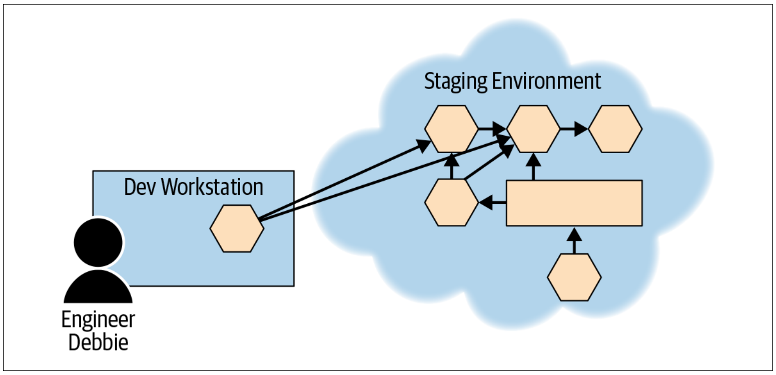


Figure 3-4. Debbie Dev connecting a locally running service into staging

Overriding Service Versions in Staging

The Food Delivery Service and the Payment Processor also provide engineers with the ability to override the version of a service running in the shared staging environment (Figure 3-5). An engineer can take a feature branch build of a service (that has not yet landed on master) and temporarily deploy that build into a staging environment. This capability is used sparingly—typically when an engineer has a particularly risky or complex change—but is very valuable when needed.

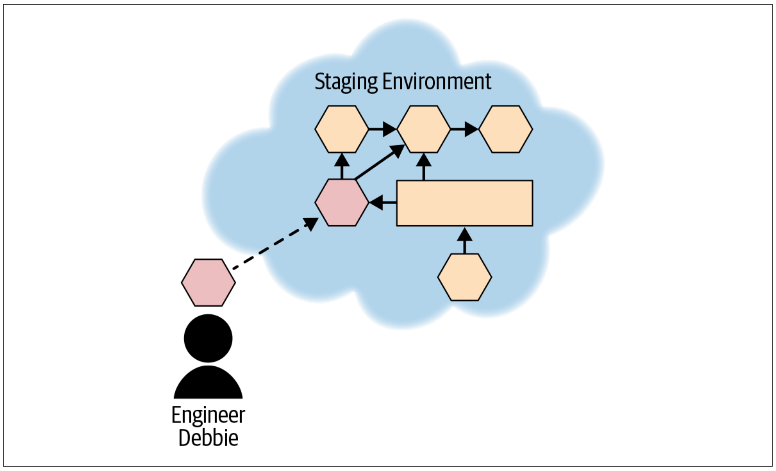


Figure 3-5. Debbie Dev temporarily overriding the version of her service in staging

CHAPTER 4

Deployment and Release

All the participants I surveyed are making production deployments at least daily. In this chapter we'll look at the techniques they use to achieve this release tempo.

In every single organization, the engineer who makes a change takes ownership of moving that change into production. They are also accountable for ensuring that the change does not cause production defects.

Single-Piece Flow

For smaller codebases owned by a single team, such as microservices, each change landing on master preferably only sits in staging briefly—just long enough for an engineer to make any last spot checks—before being promoted to production by the same engineer.

Several participants shared a strong preference for *single-piece flow*, a concept from Lean Manufacturing where batch sizes are reduced down to the single item that's actively being worked on. Teams apply this concept in software by avoiding multiple changes batching up in staging.

Release Buses

A larger, monolithic codebase make it much harder to achieve single piece flow. It has such a broad scope that different teams own different areas (this diffused ownership is, in my mind, a good working definition of a monolith). At any one time, changes will be landing from multiple teams, and they'll be arriving at a rapid pace, since a large number of engineers are all targeting their changes at the same monolithic codebase.

Organizations handle this scenario by batching production changes up into a release candidate. One engineer referred to a *Release Bus* approach, and describes it as follows: every hour, an automated system identifies changes that have landed in staging but have not yet been promoted to production.[1] These changes constitute the "passengers" on the next release bus, which is getting ready to head off to production. The system identifies the engineers who own these changes, and asks them all to confirm that their respective changes are good to go to production by performing whatever spot checks are necessary in the preproduction environment where that bus has already been deployed. If any engineer spots a problem the entire release is abandoned, and the bus is sent back to the depot. If all engineers give the thumbs up the bus is deployed into production, and engineers are notified so that they can ensure there are no production issues.

The organization that described the Release Bus system to me has made a large investment in automation. Other participants reported a similar approach, but orchestrated by an engineer, rather than automation, as part of a rotating *Release Raccoon* role. Once a day, this engineer would identify the batch of changes for the next release bus, coordinate with engineers and testers to validate that the bus is good to go, and then orchestrate the bus's journey into production. The delightful Release Raccoon nomenclature comes from the Amplify team in this blog post, although the etymology is murky.

Coordinating Production Changes

Regardless of their investment in automation, every participant reported manual coordination and orchestration from time to time around production deployments.

1 I assume that the Release Bus naming is a play on the traditional Release Train approach, where an extremely large batch of changes accumulates over a multiweek period, with a cut-off date at which the "train leaves the station" and no further changes are allowed into that batch.

An engineer might want to request a temporary pause on deployments while they investigate a production issue. As stated in Chapter 2, some teams will on occasion want to declare master as unstable (and thus not deployable). There are also situations where a change in one service depends on another change being deployed first, even though engineers agree that this sort of release coupling should be avoided as much as possible.

Participants have various mechanisms to manage this coordination, with the most common being communication over shared chat channels, often augmented with bots that contribute context such as deployments and alerts, along with low-friction remediation, an approach sometimes referred to as Chat Ops.

Participants with a large number of engineers invest significantly in custom release tooling, which includes coordination capabilities. For example, at the Food Delivery Service, engineers have the ability to "thumbs up" a specific build within a release dashboard, as well as to request a hold on production deploys for a service (with a note explaining why).

Custom Delivery Platforms

A common theme among participants was an investment in custom tooling to automate deployment processes. This appears to be an expensive but necessary investment to empower engineers to manage their own releases, which is widely regarded as extremely valuable.

This tooling provide a variety of capabilities, such as:

- Tracking which version of each service is deployed into an environment
- Reporting which new versions of a service are available for deployment
- Signing off on a version as being ready for production
- Requesting a hold on production deployments
- Deploying a new version of a service into an environment, including in some cases capabilities for things like incremental rollout or blue/green deployment
- Rolling back to a previous deployment
- Showing a history of previous deployments

- Performing data management tasks in an environment (such as reseeding test data or importing scrubbed production data)
- Reporting overall service health in an environment
- Providing a Service Registry—a way to view metadata about the service in an environment, such as team ownership, service dependencies, and quick links to production dashboards

Controlled Rollout

A faster release tempo means less time to test changes before they are put in front of users. You might think this means a higher likelihood of production defects, but research has in fact shown the opposite—deploying more frequently has a positive relationship with both a lower change-failure rate and a lower *mean time to recovery* (MTTR).[2]

Nevertheless, all participants do have mechanisms in place to reduce or mitigate the risk of a change causing a production defect, by allowing fine-grained control over how a change is rolled out to users in production. I collectively refer to these mechanisms as *Controlled Rollout*.

In Continuous Delivery there is a distinction between the technical act of *deploying* a build artifact and the user-facing act of *releasing* a feature to users. There are techniques to control rollout at both levels.

Incremental Deployment

At a low level, the deployment of a specific version of an artifact can be performed incrementally, using techniques like blue/green deployment (sometimes called red/black deployment, because naming things is hard), rolling deployment, and canary deployment.

You need some form of incremental deployment in order to perform a deployment without downtime. All participants are deploying to production very frequently, and incurring downtime as part of each deployment is not an option. Therefore, they all use some form of

2 *Accelerate*, Chapter 2.

incremental deployment. Engineers at the Financial Services Startup can directly control that incremental deployment, as a way to manage the impact of a risky change. However, this is fairly unusual. For most participants the actual act of deploying a new build is an all-or-nothing operation as far as the engineer deploying is concerned, with no fine-grained control over the rollout.

Decoupling Deployment from Release

It's possible to deploy the implementation of a feature without exposing that feature to users. *Feature flagging* is the technique that enables this decoupling of deployment from release. An engineer can deploy a half-finished feature into production, but hide it from users behind a feature flag, a mechanism that decides at runtime whether a given feature should be enabled for a user, based on some configuration.

Once the feature is complete, they can use that same feature flag to manage a controlled rollout of that feature. They might decide to initially expose it to 5% of users (a canary release), or they can opt to expose it to a specific cohort of users (an A/B test).[3]

All participants report that feature flagging is an important part of their Continuous Delivery practice, for two reasons. First, feature flags allow engineers to develop larger features incrementally—an engineer can integrate half-finished work to master, allowing one big, risky change to be sliced into multiple small, safer changes. Second, feature flags provide the safety net of controlled rollout, allowing risky changes to flow quickly into production with less risk of users being exposed to defects.

Correlating Cause and Effect

Engineers are responsible for rolling out production changes—and checking for any negative impacts from those changes—at all participating companies. This means they keep an eye on dashboards showing production metrics for some time after deploying a build or rolling out a feature.

3 Feature flagging enables a bunch of additional controlled release patterns. The Managing Feature Flags report from O'Reilly is a good resource for more details.

In order to figure out whether a change has a negative impact an engineer needs to be able to correlate the observed impact (say, an increase in error rates) with a change (rolling out a feature). In other words, they need to be able to connect cause and effect. The most obvious way to do this is with temporal correlation—I see that error rates increased at 10:24 am, and I know that I rolled out a code change at 10:23 am. Environments with a rapid deployment tempo make this correlation more challenging. If I see a production issue and there's been one deployment in the last few hours then I have a place to start looking. If there's been 10 deployments in the last hour my job is a little harder.

Incremental rollouts bring further challenges when it comes to correlating cause and effect. After rolling out a risky change to a canary population (5% of users, let's say), an engineer needs some way to compare and contrast metrics for that canary population versus the general population. Rather than solving this correlation problem in a general way—which would require a large technical investment—most participants achieve this correlation via proxy attributes. For example, the Healthcare Provider and the Food Delivery Service both roll out risky changes to a canary market, rather than a random sample of their user base. An engineer would roll out a change to all users in Denver, let's say, and then keep an eye on whether metrics for users in Denver are changing relative to the metrics in other cities.

Moving Fast with Safety

We've seen that participants achieve the most rapid release tempo by maintaining a continuous flow of small, independent changes into production. This requires a set of practices and techniques, as well as discipline, but the outcomes are worthwhile. The same tools that allow a team to make small, incremental changes also reduce the risk associated with a feature release, and greatly improve the team's ability to react to a bad change when it does occur.

CHAPTER 5

Summary

We've looked at how a variety of organizations achieve a rapid tempo of production changes using the principles of Continuous Delivery. We've seen some common themes across that wide range of organizations—fundamental values of Continuous Delivery that seem to be universal. We've also seen some interesting variations in practices.

Shared Values

As we saw in Chapter 2, all organizations have found value in reducing the size of each production change. Ideas like Trunk-Based Development and decoupling deployment from release allow engineers to get closer to their ideal of single-piece flow.

Organizations that excel at Continuous Delivery all empower product engineers with autonomy, as well as accountability for their changes. The engineer who authors a change is the person responsible for shepherding that change into production and watching for any potential defects. Product delivery teams also have a lot of autonomy in terms of how they work—I repeatedly heard from participants that it was hard to describe the delivery process since different teams within the organization work in different ways.

In order to achieve this level of autonomy, there is a noticeably heavy investment in custom delivery platforms that provide self-service capabilities to product engineers (there is a summary of the capabilities of these platforms in Chapter 4). Many organizations make a distinction between their Continuous Integration system and their Continuous Delivery infrastructure, where their Continuous Integration system is responsible for initially validating a change and building a deployable artifact, while the Continuous Delivery infrastructure is responsible for moving a build artifact through various environments and monitoring those environments for potential issues. The Automotive Marketplace is the only organization I spoke with that has a unified system providing both Continuous Integration and Continuous Delivery.

Two Modes of Continuous Delivery

While participants shared a lot of software delivery practices, I did notice that all fell into one of two distinct modes of Continuous Delivery.

One mode is *Branch-Based Continuous Delivery*, where a team uses short-lived feature branches as their unit of change and manages code review with pull requests. The other mode is *Trunk-Based Continuous Delivery*, where teams practice Trunk-Based delivery, work directly on master, and do ad hoc code review.

Trunk-Based teams have a more rapid deployment tempo than Branch-Based teams with multiple new deployments per hour for Trunk-Based versus once or twice a day for Branch-Based.

There is an even starker distinction in the typical cycle times for a production change between these two modes, with changes being in progress for much longer for Branch-Based teams. This is partly because the unit of change for Branch-Based teams is much larger—often an entire feature, rather than an individual commit. In addition, Trunk-Based teams are typically doing code review out-of-band, rather than blocking a change from going out.

Trunk-Based teams also tend to rely much less on preproduction environments for quality checks, instead of testing in production. I suspect this is because faster change makes the sort of manual validation that is done in preproduction environments a lot less feasible or valuable.

Most of the larger engineering organizations I talked to had teams in both of these modes. Generally the teams working with larger, older systems use Branch-Based Continuous Delivery while teams working on new, smaller systems operate using Trunk-Based Continuous Delivery.

Monoliths Versus SOA

The high-level architecture of a system has a marked impact on how Continuous Delivery is implemented. Most notably, as discussed in Chapter 4, teams working on large, monolithic systems are forced to batch production changes up using mechanisms like Release Buses. This is due to the rate of changes from multiple teams, a relatively slow deployment, and cross-team coordination challenges. In contrast, in SOAs a team typically has full ownership of a small codebase, and can often achieve single-piece flow, where each unit of change rolls out to production independently. Several participants highlighted this as a big advantage of working in a service-oriented system.

Succeeding with Continuous Delivery

After studying how teams are succeeding with Continuous Delivery in the wild, themes emerged which inform your strategy.

Reducing batch size should be a guiding principle. Trunk-Based Development practices like Feature Flagging and Branch by Abstraction are also key. Keep the size of each unit of change as small as possible (e.g., by aggressively focusing on short feature branches), and avoid batching up production changes whenever possible. The ideal is to get all the way to pure Trunk-Based Development, eschewing feature branches entirely and reaching single-piece flow.

In general, branches should be viewed with suspicion. Long-lived feature branches should certainly be avoided, but so should the use of branches for release management. Instead, prefer delivery pipelines that flow a build artifact through environments, and eventually into production, as discussed in Chapter 2.

Organizations who want to move toward Continuous Delivery should plan to invest quite a lot in a delivery platform with self-service tooling for product engineers. Rather than having infrastructure or operations engineers manage deployment and monitor production themselves, they should focus on building the tools that enable product engineers to do this work.

The most consistent thing I heard from the participants was that Continuous Delivery has resulted in overwhelmingly positive outcomes. I hope that the experiences of those organizations will help you achieve the same.

About the Author

Pete Hodgson is an independent software delivery consultant, based near San Francisco. He teaches teams to deliver awesome products at a sustainable pace, by leveling up their engineering practices and technical architecture. Before going independent, Pete spent several years as a consultant with ThoughtWorks, leading technical practices for their West Coast business, in addition to several stints as a tech lead and architect at various San Francisco startups.