9 Create the Foundations of Our Deployment Pipeline

In order to create fast and reliable flow from Dev to Ops, we must ensure that we always use production-like environments at every stage of the value stream. Furthermore, these environments must be created in an automated manner, ideally on demand from scripts and configuration information stored in version control, and entirely self-serviced, without any manual work required from Operations. Our goal is to ensure that we can re-create the entire production environment based on what's in version control.

All too often, the only time we discover how our applications perform in anything resembling a production-like environment is during production deployment—far too late to correct problems without the customer being adversely impacted. An illustrative example of the spectrum of problems that can be caused by inconsistently built applications and environments is the Enterprise Data Warehouse program led by Em Campbell-Pretty at a large Australian telecommunications company in 2009. Campbell-Pretty became the general manager and business sponsor for this $200 million program, inheriting responsibility for all the strategic objectives that relied upon this platform.

In her presentation at the 2014 DevOps Enterprise Summit, Campbell-Pretty explained, "At the time, there were ten streams of work in progress, all using waterfall processes, and all ten streams were significantly behind schedule. Only one of the ten streams had successfully reached User Acceptance Testing [UAT] on schedule, and it took another six months for that stream to complete UAT, with the resulting capability falling well short of business expectations. This under-performance was the main catalyst for the department's Agile transformation."

However, after using Agile for nearly a year, they experienced only small improvements, still falling short of their needed business outcomes. Campbell-Pretty held a program-wide retrospective and asked, "After reflecting

on all our experiences over the last release, what are things we could do that would double our productivity?"

Throughout the project, there was grumbling about the "lack of business engagement." However, during the retrospective, "improve availability of environments" was at the top of the list. In hindsight, it was obvious—Development teams needed provisioned environments in order to begin work, and were often waiting up to eight weeks.

They created a new integration and build team that was responsible for "building quality into our processes, instead of trying to inspect quality after the fact." It was initially comprised of database administrators (DBAs) and automation specialists tasked with automating their environment creation process. The team quickly made a surprising discovery: only 50% of the source code in their development and test environments matched what was running in production.

Campbell-Pretty observed, "Suddenly, we understood why we encountered so many defects each time we deployed our code into new environments. In each environment, we kept fixing forward, but the changes we made were not being put back into version control."

The team carefully reverse-engineered all the changes that had been made to the different environments and put them all into version control. They also automated their environment creation process so they could repeatedly and correctly spin up environments.

Campbell-Pretty described the results, noting that "the time it took to get a correct environment went from eight weeks to one day. This was one of the key adjustments that allowed us to hit our objectives concerning our lead time, the cost to deliver, and the number of escaped defects that made it into production."

Campbell-Pretty's story shows the variety of problems that can be traced back to inconsistently constructed environments and changes not being systematically put back into version control.

Throughout the remainder of this chapter, we will discuss how to build the mechanisms that will enable us to create environments on demand, expand the use of version control to everyone in the value stream, make infrastructure easier to rebuild than to repair, and ensure that developers run their code in

production-like environments along every stage of the software development life cycle.

ENABLE ON DEMAND CREATION OF DEV, TEST, AND PRODUCTION ENVIRONMENTS

As seen in the enterprise data warehouse example above, one of the major contributing causes of chaotic, disruptive, and sometimes even catastrophic software releases, is the first time we ever get to see how our application behaves in a production-like environment with realistic load and production data sets is during the release.[†] In many cases, development teams may have requested test environments in the early stages of the project.

However, when there are long lead times required for Operations to deliver test environments, teams may not receive them soon enough to perform adequate testing. Worse, test environments are often mis-configured or are so different from our production environments that we still end up with large production problems despite having performed pre-deployment testing.

In this step, we want developers to run production-like environments on their own workstations, created on demand and self-serviced. By doing this, developers can run and test their code in production-like environments as part of their daily work, providing early and constant feedback on the quality their work.

Instead of merely documenting the specifications of the production environment in a document or on a wiki page, we create a common build mechanism that creates all of our environments, such as for development, test, and production. By doing this, anyone can get production-like environments in minutes, without opening up a ticket, let alone having to wait weeks.[‡]

† In this context, environment is defined as everything in the application stack except for the application, including the databases, operating systems, networking, virtualization, and all associated configurations.

‡ Most developers want to test their code, and they have often gone to extreme lengths to obtain test environments to do so. Developers have been known to reuse old test environments from previous projects (often years old) or ask someone who has a reputation of being able to find one—they just won't ask where it came from, because, invariably, someone somewhere is now missing a server.

To do this requires defining and automating the creation of our known, good environments, which are stable, secure, and in a risk-reduced state, embodying the collective knowledge of the organization. All our requirements are embedded, not in documents or as knowledge in someone's head, but codified in our automated environment build process.

Instead of Operations manually building and configuring the environment, we can use automation for any or all of the following:

- Copying a virtualized environment (e.g., a VMware image, running a Vagrant script, booting an Amazon Machine Image file in EC2)
- Building an automated environment creation process that starts from "bare metal" (e.g., PXE install from a baseline image)
- Using "infrastructure as code" configuration management tools (e.g., Puppet, Chef, Ansible, Salt, CFEngine, etc.)
- Using automated operating system configuration tools (e.g., Solaris Jumpstart, Red Hat Kickstart, Debian preseed)
- Assembling an environment from a set of virtual images or containers (e.g., Vagrant, Docker)
- Spinning up a new environment in a public cloud (e.g., Amazon Web Services, Google App Engine, Microsoft Azure), private cloud, or other PaaS (platform as a service, such as OpenStack or Cloud Foundry, etc.).

Because we've carefully defined all aspects of the environment ahead of time, we are not only able to create new environments quickly, but also ensure that these environments will be stable, reliable, consistent, and secure. This benefits everyone.

Operations benefits from this capability, to create new environments quickly, because automation of the environment creation process enforces consistency and reduces tedious, error-prone manual work. Furthermore, Development benefits by being able to reproduce all the necessary parts of the production environment to build, run, and test their code on their workstations. By doing this, we enable developers to find and fix many problems, even at the earliest stages of the project, as opposed to during integration testing or worse, in production.

By providing developers an environment they fully control, we enable them to quickly reproduce, diagnose, and fix defects, safely isolated from production services and other shared resources. They can also experiment with changes to the environments, as well as to the infrastructure code that creates it (e.g., configuration management scripts), further creating shared knowledge between Development and Operations.[†]

CREATE OUR SINGLE REPOSITORY OF TRUTH FOR THE ENTIRE SYSTEM

In the previous step, we enabled the on demand creation of the development, test, and production environments. Now we must ensure that all parts of our software system.

For decades, comprehensive use of version control has increasingly become a mandatory practice of individual developers and development teams.[‡] A version control system records changes to files or sets of files stored within the system. This can be source code, assets, or other documents that may be part of a software development project. We make changes in groups called commits or revisions. Each revision, along with metadata such as who made the change and when, is stored within the system in one way or another, allowing us to commit, compare, merge, and restore past revisions to objects to the repository. It also minimizes risks by establishing a way to revert objects in production to previous versions. (In this book, the following terms will be used interchangeably: checked in to version control, committed into version control, code commit, change commit, commit.)

When developers put all their application source files and configurations in version control, it becomes the single repository of truth that contains the precise intended state of the system. However, because delivering value to the customer requires both our code and the environments they run in, we need our environments in version control as well. In other words, version control is for everyone in our value stream, including QA, Operations, Infosec,

† Ideally, we should be finding errors before integration testing when is too late in the testing cycle to create fast feedback for developers. If we are unable to do so, we likely have an architectural issue that needs to be addressed. Designing our systems for testability, to include the ability to discover most defects using a non-integrated virtual environment on a development workstation, is a key part of creating an architecture that supports fast flow and feedback.

‡ The first version control system was likely UPDATE on the CDC6600 (1969). Later came SCCS (1972), CMS on VMS (1978), RCS (1982), and so forth.

as well as developers. By putting all production artifacts into version control, our version control repository enables us to repeatedly and reliably reproduce all components of our working software system—this includes our applications and production environment, as well as all of our pre-production environments.

To ensure that we can restore production service repeatedly and predictably (and, ideally, quickly) even when catastrophic events occur, we must check in the following assets to our shared version control repository:

- All application code and dependencies (e.g., libraries, static content, etc.)
- Any script used to create database schemas, application reference data, etc.
- All the environment creation tools and artifacts described in the previous step (e.g., VMware or AMI images, Puppet or Chef recipes, etc.)
- Any file used to create containers (e.g., Docker or Rocket definition or composition files)
- All supporting automated tests and any manual test scripts
- Any script that supports code packaging, deployment, database migration, and environment provisioning
- All project artifacts (e.g., requirements documentation, deployment procedures, release notes, etc.)
- All cloud configuration files (e.g., AWS Cloudformation templates, Microsoft Azure Stack DSC files, OpenStack HEAT)
- Any other script or configuration information required to create infrastructure that supports multiple services (e.g., enterprise service buses, database management systems, DNS zone files, configuration rules for firewalls, and other networking devices).[†]

† One may observe that version control fulfills some of the ITIL constructs of the Definitive Media Library (DML) and Configuration Management Database (CMDB), inventorying everything required to re-create the production environment.

We may have multiple repositories for different types of objects and services, where they are labelled and tagged alongside our source code. For instance, we may store large virtual machine images, ISO files, compiled binaries, and so forth in artifact repositories (e.g., Nexus, Artifactory). Alternatively, we may put them in blob stores (e.g., Amazon S3 buckets) or put Docker images into Docker registries, and so forth.

It is not sufficient to merely be able to re-create any previous state of the production environment; we must also be able to re-create the entire pre-production and build processes as well. Consequently, we need to put into version control everything relied upon by our build processes, including our tools (e.g., compilers, testing tools) and the environments they depend upon.[‡]

In Puppet Labs' *2014 State of DevOps Report*, the use of version control by Ops was the highest predictor of both IT performance and organizational performance. In fact, whether Ops used version control was a higher predictor for both IT performance and organizational performance than whether Dev used version control.

The findings from Puppet Labs' *2014 State of DevOps Report* underscores the critical role version control plays in the software development process. We now know when all application and environment changes are recorded in version control, it enables us to not only quickly see all changes that might have contributed to a problem, but also provides the means to roll back to a previous known, running state, allowing us to more quickly recover from failures.

But why does using version control for our environments predict IT and organizational performance better than using version control for our code?

Because in almost all cases, there are orders of magnitude more configurable settings in our environment than in our code. Consequently, it is the environment that needs to be in version control the most.[§]

‡ In future steps, we will also check in to version control all the supporting infrastructure we build, such as the automated test suites and our continuous integration and deployment pipeline infrastructure.

§ Anyone who has done a code migration for an ERP system (e.g., SAP, Oracle Financials, etc.) may recognize the following situation: When a code migration fails, it is rarely due to a coding error. Instead, it's far more likely that the migration failed due to some difference in the environments, such as between Development and QA or QA and Production.

Version control also provides a means of communication for everyone working in the value stream—having Development, QA, Infosec, and Operations able to see each other's changes helps reduce surprises, creates visibility into each other's work, and helps build and reinforce trust. (See Appendix 7.)

MAKE INFRASTRUCTURE EASIER TO REBUILD THAN TO REPAIR

When we can quickly rebuild and re-create our applications and environments on demand, we can also quickly rebuild them instead of repairing them when things go wrong. Although this is something that almost all large-scale web operations do (i.e., more than one thousand servers), we should also adopt this practice even if we have only one server in production.

Bill Baker, a distinguished engineer at Microsoft, quipped that we used to treat servers like pets: "You name them and when they get sick, you nurse them back to health. [Now] servers are [treated] like cattle. You number them and when they get sick, you shoot them."

By having repeatable environment creation systems, we are able to easily increase capacity by adding more servers into rotation (i.e., horizontal scaling). We also avoid the disaster that inevitably results when we must restore service after a catastrophic failure of irreproducible infrastructure, created through years of undocumented and manual production changes.

To ensure consistency of our environments, whenever we make production changes (configuration changes, patching, upgrading, etc.), those changes need to be replicated everywhere in our production and pre-production environments, as well as in any newly created environments.

Instead of manually logging into servers and making changes, we must make changes in a way that ensures all changes are replicated everywhere automatically and that all our changes are put into version control.

We can rely on our automated configuration systems to ensure consistency (e.g., Puppet, Chef, Ansible, Salt, Bosh, etc.), or we can create new virtual machines or containers from our automated build mechanism and deploy them into production, destroying the old ones or taking them out of rotation.[†]

† At Netflix, the average age of Netflix AWS instance is twenty-four days, with 60% being less than one week old.

The latter pattern is what has become known as *immutable infrastructure*, where manual changes to the production environment are no longer allowed—the only way production changes can be made is to put the changes into version control and re-create the code and environments from scratch. By doing this, no variance is able to creep into production.

To prevent uncontrolled configuration variances, we may disable remote logins to production servers[‡] or routinely kill and replace production instances, ensuring that manually-applied production changes are removed. This action motivates everyone to put their changes in the correct way through version control. By applying such measures, we are systematically reducing the ways our infrastructure can drift from our known, good states (e.g., configuration drift, fragile artifacts, works of art, snowflakes, and so forth).

Also, we must keep our pre-production environments up to date—specifically, we need developers to stay running on our most current environment. Developers will often want to keep running on older environments because they fear environment updates may break existing functionality. However, we want to update them frequently so we can find problems at the earliest part of the life cycle.[§]

MODIFY OUR DEFINITION OF DEVELOPMENT "DONE" TO INCLUDE RUNNING IN PRODUCTION-LIKE ENVIRONMENTS

Now that our environments can be created on demand and everything is checked in to version control, our goal is to ensure that these environments are being used in the daily work of Development. We need to verify that our application runs as expected in a production-like environment long before the end of the project or before our first production deployment.

Most modern software development methodologies prescribe short and iterative development intervals, as opposed to the big bang approach (e.g., the waterfall `model). In general, the longer the interval between deployment, the worse the outcomes. For example, in the Scrum methodology a *sprint* is a time-boxed development interval (typically one month or less) within which

‡ Or allow it only in emergencies, ensuring that a copy of the console log is automatically emailed to the operations team.

§ The entire application stack and environment can be bundled into containers, which can enable unprecedented simplicity and speed across the entire deployment pipeline.

we are required to be done, widely defined as when we have "working and potentially shippable code."

Our goal is to ensure that Development and QA are routinely integrating the code with production-like environments at increasingly frequent intervals throughout the project.[†] We do this by expanding the definition of "done" beyond just correct code functionality (addition in bold text): at the end of each development interval, we have integrated, tested, working and potentially shippable code, **demonstrated in a production-like environment.**

In other words, we will only accept development work as done when it can be successfully built, deployed, and confirmed that it runs as expected in a production-like environment, instead of merely when a developer believes it to be done—ideally, it runs under a production-like load with a production-like dataset, long before the end of a sprint. This prevents situations where a feature is called done merely because a developer can run it successfully on their laptop but nowhere else.

By having developers write, test, and run their own code in a production-like environment, the majority of the work to successfully integrate our code and environments happens during our daily work, instead of at the end of the release. By the end of our first interval, our application can be demonstrated to run correctly in a production-like environment, with the code and environment having been integrated together many times over, ideally with all the steps automated (no manual tinkering required).

Better yet, by the end of the project, we will have successfully deployed and run our code in production-like environments hundreds or even thousands of times, giving us confidence that most of our production deployment problems have been found and fixed.

Ideally, we use the same tools, such as monitoring, logging, and deployment, in our pre-production environments as we do in production. By doing this, we have familiarity and experience that will help us smoothly deploy and run, as well as diagnose and fix, our service when it is in production.

† The term *integration* has many slightly different usages in Development and Operations. In Development, integration typically refers to code integration, which is the integration of multiple code branches into trunk in version control. In continuous delivery and DevOps, *integration testing* refers to the testing of the application in a production-like environment or integrated test environment.

By enabling Development and Operations to gain a shared mastery of how the code and environment interact, and practicing deployments early and often, we significantly reduce the deployment risks that are associated with production code releases. This also allows us to eliminate an entire class of operational and security defects and architectural problems that are usually caught too late in the project to fix.

CONCLUSION

The fast flow of work from Development to Operations requires that anyone can get production-like environments on demand. By allowing developers to use production-like environments even at the earliest stages of a software project, we significantly reduce the risk of production problems later. This is one of many practices that demonstrate how Operations can make developers far more productive. We enforce the practice of developers running their code in production-like environments by incorporating it into the definition of "done."

Furthermore, by putting all production artifacts into version control, we have a "single source of truth" that allows us to re-create the entire production environment in a quick, repeatable, and documented way, using the same development practices for Operations work as we do for Development work. And by making production infrastructure easier to rebuild than to repair, we make resolving problems easier and faster, as well as making it easier to expand capacity.

Having these practices in place sets the stage for enabling comprehensive test automation, which is explored in the next chapter.