WHITE PAPER

Speed Thrills: How to Harness the Power of CI/CD for Your Development Team

By Ben Kamysz and Jared Ruckle

Pivotal

Table of Contents

Introduction

Your code is worthless until it's in the hands of users. Risk and uncertainty accumulate the longer you hold on to code.

These uncomfortable truths spawned the Lean Software Development movement. Now, there's a coalition of developers, operators, product owners, and executives rallying around another truth:

Time-to-value is the most important metric in business today.

Continuous Integration (CI) and Continuous Delivery (CD) are foundational elements of realizing rapid time-to-value. These processes and related tools help you automate the checks and balances required to ensure a high level of code quality. At the same time, they deliver new code to customers quickly.

Many enterprises committed to a software-led digital transformation are counting on five key benefits:

- Speed - how can I release software faster than I am now? How do go from a few releases a year to many releases in a week?
- Stability - how do I deliver enterprise-grade uptime for mission-critical apps and websites?
- Scalability - how do I handle unpredictable amounts of traffic to my APIs?
- Savings - how can I save money with more efficient infrastructure spending? How can I trim legacy software costs?
- Security - how do I boost my security posture while making this transformation?

In this paper, we'll explore how CI and CD can contribute to all of these factors. We'll also provide pragmatic strategies and tactics to help you along the way.

Why You Need CI and CD

Continuous Integration (CI) and Continuous Delivery (CD) are essential for any organization that seeks to rapidly deliver software. These methods are the natural evolution of agile development. At the core of these practices: your software should always be in a deployable state.

Continuous Integration is the process of automatically building software after new bits of code are integrated into a shared repository. This yields "builds" of the code base that are in a working state at all times. Unit tests are included as part of the build generation process, thereby validating the functionality of the software. This identifies bugs up-front, and prevents wasted cycles further down the feedback loop.

Continuous Delivery is the process of automatically deploying the artifacts created with CI. Scripts are used to provide a consistent method of deploying software and configuration changes into an environment. The consistency CD provides means production deployments become a non-event.

The fundamental idea of a DevOps culture: optimize your teams and processes around the shared goal of delivering value to the customer. This is a useful way to look at the practice of CI and CD. The most successful teams:

- Write unit tests to verify that requirements are met. Unit tests improve quality, while ensuring that changes to the software do not break existing functionality.
- Create end-to-end tests to simulate user flows throughout the application. This is an essential QA practice. Note: Many organizations have a separate QA team before starting their DevOps journey. Over time, the QA function is performed by development and DevOps teams as processes mature.
- Create deployment scripts to ensure consistency between the environments.

CI and CD are transformational changes. As such, they are the responsibility of the entire IT organization.

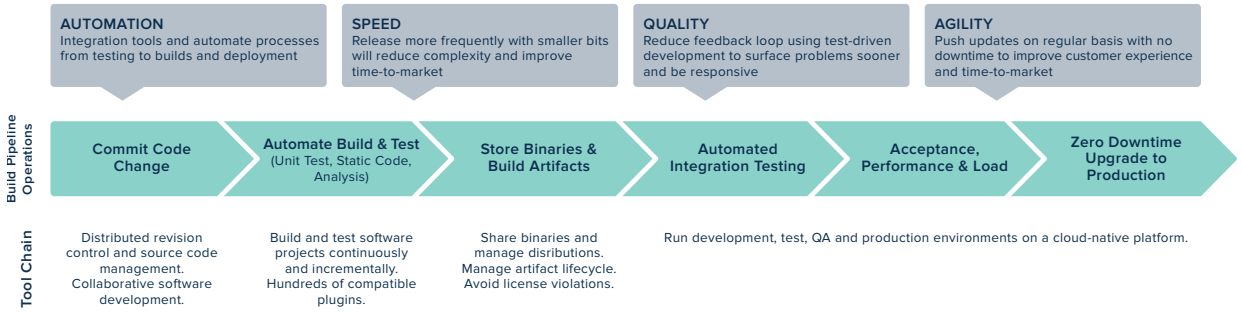


Figure 1: An overview of continuous integration & delivery phases.

If you want to get better at software, you should adopt CI/CD. So how should you get started? Glad you asked!

Get Started with CI/CD in 7 Steps

There's a lot of excitement around Continuous Integration and Continuous Delivery. And with it, comes a deluge of advice, best practices, and e-books. For our part at Pivotal, we've helped hundreds of organizations implement CI and CD. We've collected the essential best practices here. Read on for practical guidance that's proven to boost development velocity!

1. Get code into source control

The first of the 12-factors for modern app development? "One codebase tracked in revision control, many deploys."

There's a reason why this is the first factor. CI and CD require that all parties work in sync. A shared code repository is the foundation for boosting development velocity. Version control allows changes to be mapped to their respective requirement and to their contributor. Common source control features like branching enable multiple versions of the code to be built at any given time. This enables simultaneous build streams and parallel work. For example, a team can fix production bugs in one branch, while a second team works on the next release in a separate, second branch.

Many agile enterprises choose Git or Team Foundation Server (TFS) as their source control repository. Why? Both offer distributed version control; every client pulls down the entire repository. Local commands are faster, since communication with the server is required **only** when changes are shared back to others. These tools aid in a disaster recovery scenario, should central servers ever fail.

2. Use a build automation tool

If you want to **release** code often, you have to **build** your code often. Automating this process eliminates unique "snowflake" configurations and harmful "tribal knowledge" silos that prevent faster cycles. Successful CI and CD depend on consistent builds. With tools like Gradle, Maven and Nuget, compiling and packaging are managed in simple configuration files. Dependencies - and their host repositories - are defined in these files. This ensures stable versions of upstream software. Configuration files also define where to store a compiled artifact. Versioning information is included; consumers of the compiled artifact know which build of the software they are using. Use semantic versioning to further simplify dependency management. This approach provides a consistent specification for compatible software versions.

This important practice is codified in the 12 factor manifesto as "store config in the environment".

3. Write unit tests

A crucial component continuous integration: unit testing. Unit tests are written against code to verify that functional requirements are met. When teams adhere to unit tests, they produce high quality software, faster. Over time, unit tests become more and more valuable. Developers use unit tests to confirm that newer versions of the code don't break current features. Java projects typically use a framework such as JUnit for testing. Pivotal created Jasmine, a framework, for testing Javascript.

Pivotal encourages teams to adopt Test Driven Development (TDD). Here, teams write tests first. Of course, tests fail at the start, since no code has been written. Developers then write the minimum amount of code required to pass the test. The final step in TDD is to refactor the code to the coding standards established for the project. This practice makes the code base more manageable, eliminating duplication and needless complexity. We call this process "red, green, refactor". "Red" indicates a failing unit test, "green" denotes a passing unit test result (with the minimal amount of code), while "refactor" means following the aforementioned project standards.

This method has two benefits: requirements are established up front, and nothing is missed in the implementation of the software.

4. Store builds in an artifact repository

When a build completes successfully, it produces an artifact: a package of code that can be tested. Most teams have many successful builds before a given release moves to production - hence the need for a unified, central place to access these outputs. Using an artifact repository makes an artifact easily accessible to downstream projects. This, in turn, accelerates testing and subsequent promotion of production-ready software. Teams often use simple, HA blob storage, or more full-featured products like JFrog Artifactory and Sonatype Nexus.

5. Create deployment scripts

After an artifact is created, it needs to be deployed to a given environment in a consistent way. Use deployment scripts! These scripts are usually written in a shell language. They execute common tasks like code deployment code and environment configuration. Often, automation tools like Concourse and Jenkins have plugins to handle most commands. Refer to Automate builds and deployments for more details.

6. Write end-to-end tests and smoke tests

Once code is deployed to an environment, additional testing verifies the software. Smoke tests examine the most important functionality (i.e. ensuring the application is accessible). This "light touch" helps expose integration issues quickly. After the release has passed this cursory testing, end-to-end testing kicks off.

End-to-end tests are used to simulate user experiences through the application. These tests execute automated workflows that interact with an application just exactly as a user would. End-to-end tests can use mobile device emulators, scripted API calls, and browsers to approximate the user experience. This automation is achieved with a webdriver such as Selenium.

7. Automate builds and deployment

The final step: automate the processes described above. Automation is the "continuous" part of CI and CD. Concourse and Jenkins repeat the previous steps in a consistent manner after every code commit. The organization can then test the entire code base early and often. Feedback loops are tightened. Issues are identified immediately. This saves time and money downstream, since QA testing and user acceptance testing (UAT) avoid examining trivial defects. This minutia is identified and resolved earlier in the process.

Use **build pipelines** to automate the entire testing and delivery process, from initial code commit to production deployment.

Build pipelines introduce two concepts: “jobs” and “resources.” A job is simply a unit of work in the build process, a step or a stage where code is examined in a specific way. A resource is source code, configuration file, dependency, or other artifact.

Now, let’s examine how build pipelines work in practice.

Once a job has completed successfully, code flows into the next job in the pipeline. “Entrance” and ”exit” criteria are defined for each job, to ensure all requirements are met. Code successfully exits the pipeline in production, typically as a zero-downtime deployment. Advanced techniques such as blue/green deployments can be automated.

Automated build pipelines can be a big change for a development organization. To make the transition a little less jarring, manual checks can be inserted after each job. This often gives a team “peace of mind” while they learn the best way to automate the process from start to finish.

How to Measure CI/CD Success

Implementing Continuous Integration and Continuous Delivery speeds up time to production and reduces defects. But how do you measure success and continually improve?

Based on our experience, successful organizations hone in on these Key Performance Indicators (KPIs).

1. Deployment Frequency: From Months to Days and Hours

Organizations that do not practice CI and CD typically have months between deployments. The release cycle can be shortened substantially once processes are automated. It's common to transition from releases every 6 months to daily or hourly releases. Teams that can iterate quickly have a significant competitive advantage.

Measure the frequency of code commits with your chosen source control tool. Build promotions (and also build failures) in each environment can be tracked through the build automation tool, or aggregated to another tracking tool with plugins or scripts.

2. Testing Duration: How Long Until You Know the Build is Valid?

Manual testing isn't easy. It often takes weeks or months to run a full regression test on software. Permutation testing yields endless variations, inevitably leading to overlooked edge cases. CI/CD requires teams to automate the majority of testing. As a result, manual regression cycles are shortened, even eliminated in some cases. Once testing is automated, testing duration metrics are available through the build automation tool.

3. Production Defects: Track Volume and Severity

Everyone wants flawless software. But defects slip through into production. The task becomes limiting the number of defects, and reducing their severity. Continuous Integration exposes the majority of these bugs early, leading to high-quality code. Track real-time error alerts with a tool like Rollbar. Triage and categorize user issues with tools like Jira or Pivotal Tracker.

4. Production Uptime & the Need for Zero Downtime Deployments

High availability is a "table stakes" requirement. The value of software diminishes substantially if it isn't available to users. On-demand deployments help the organization respond faster when issues occur. Use tools such as New Relic and Dynatrace measure uptime and availability. NOTE: Four nines availability (99.99%) translates to 1 minute of downtime a week. That means you must do zero downtime deployments. A consistent deployment, tested and vetted in "lower" environments boosts availability. This improves your security posture as well, since you can patch critical systems faster. You don't have to wait for nights or weekends to repair your systems.

Deploy Like the Cloud Natives with Concourse

The shift to cloud-native architectures requires rethinking many dimensions of software development. This extends to the world of CI/CD. Traditional tools can't easily build, test, and deliver software in cloud environments at scale. They can't simultaneously support different architectures, different platforms, and different IaaS deployment targets. These shortcomings drove the creation of Concourse, an open-source tool sponsored by Pivotal.

Concourse is a refreshing departure from other CI/CD tools if you need to support:

- Test-driven development with extensive automated testing
- Multiple code versions that must maintain compatibility
- Many code derivatives, such as multiple IaaS targets or configurations
- High-velocity teams that deliver code frequently

Concourse differentiates itself from other automation tools because it makes automated build pipelines a first-class citizen. Teams that adopt Concourse place build pipelines at the center of their day to day work, and even annual performance reviews!

Concourse supports cloud-native patterns a few ways:

1. Pipeline as Code

Configuration is defined in a single, declarative document that can be stored in version control. Build automation servers with "snowflake" wizard-based configurations simply cannot fit this practice.

2. Stateless Builds

Builds require a workspace to run. If the workspace isn't cleaned after every build, it becomes polluted. This is noticed most significantly when dependencies are removed from configuration. In traditional tools, the build will continue to work because said dependencies exist in the workspace. But the very same build will fail spectacularly if it's run anywhere else. Concourse uses containers to deploy "sterile" worker resources for every task. It automatically "cleans" them up later.

3. Environment Parity

Enterprises are adopting multi-cloud strategies. As such, teams need their pipelines to run virtually everywhere and anywhere consistently. Concourse is designed to run on many different environments, and also run the same locally for developers. Consequently, jobs can be run locally and unit tested before committing into source control for mass consumption. Similarly, entire Concourse pipelines and their "worker" resources can be moved to different clouds (to lower cost, for example).

4. Usability

Concourse allows easy visualization of pipelines - real-time information is always shown. Concourse adopters often display their pipeline build status on overhead monitors throughout their office. When a pipeline shows red, it's instantly visible. This motivates the team to rapidly fix broken builds.

5. Scalability

Concourse allows "worker" and "web" nodes to be declared in configuration files. This helps teams scale up, providing more resources to their their pipeline (to increase speed). Similarly, teams can easily scale down, efficiently packing more workers into fewer resources (to decrease cost). Scale according to your needs at any given time!

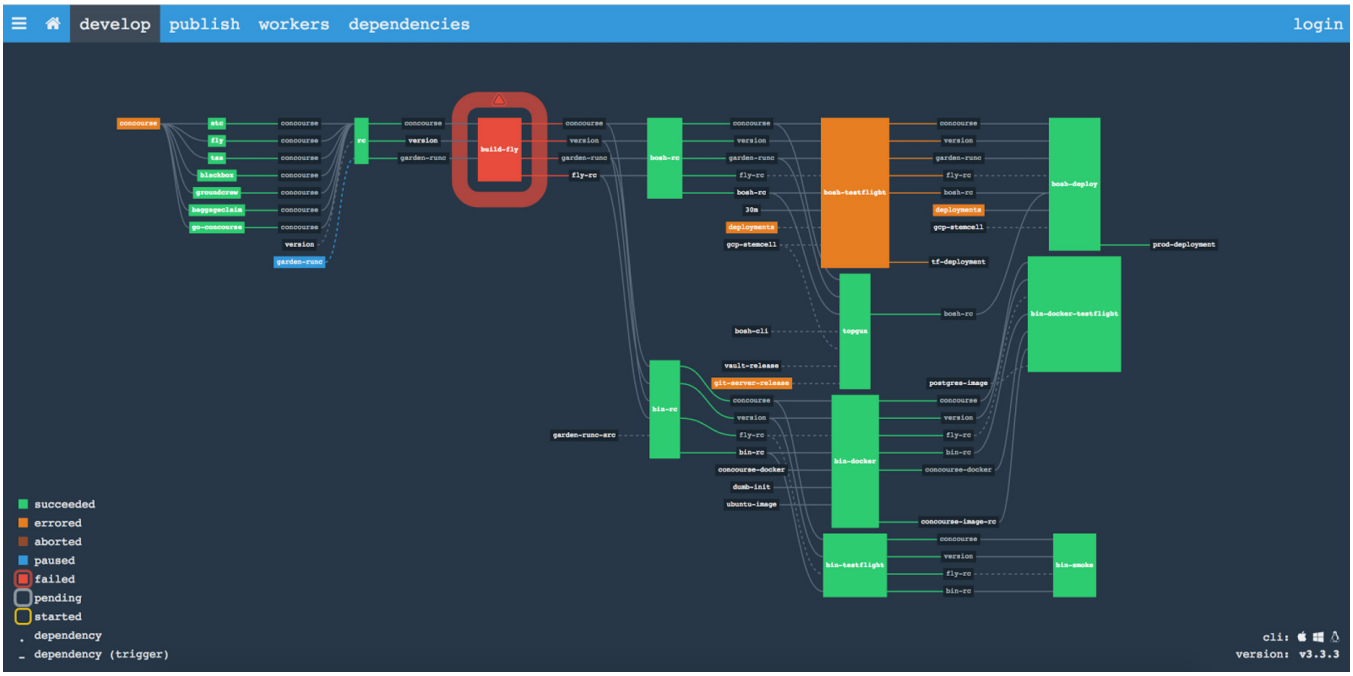


Figure 2: A build pipeline, automated with Concourse, deploying to a cloud-native platform.

Read This Before You Get Started

Just starting your CI/CD journey? Keep these things in mind, and you'll be much more successful.

1. Define version control best practices up front

Most engineers are familiar with the concept of code check-in. But don't assume this! Developers cringe when they hear a colleague yell "someone broke the build again!" Check that committers understand that the latest code needs to be pulled locally, and that all tests need to successfully pass before code is committed. Trust us - it will save lots of headaches and hurt feelings later.

2. Keep communication lines open

CI and CD is a team effort. Teams are successful when everyone is on the same page. Ensure engineers have a full understanding of requirements before writing any code. It saves massive amounts of rework later. When in doubt, just ask!

3. Don't boil the ocean when it comes to testing

Everyone loves to work on green field projects; there's no technical debt to lug around! Unfortunately, the majority of software engineers work on existing systems. Much of this code has minimal test coverage, or none at all. Achieving 100% test coverage "right out of the gate" would bring new development to a complete halt. Obviously, this won't work. So what do you do?

Write tests for any new code you ship. This stops the technical debt from growing. Adding tests when modifying existing code also decreases debt over time. Do this at a method/function level, or even a class/file level. Just think of the Robert Baden-Powell quote: "Leave this world a little better than you found it." Adhering to unit tests over time creates a flywheel effect. New tests become easier to write, because more and more dependent code is already in a known good state.

4. Start small and iterate often

Start with simple pipelines that reduce the number of places where problems can occur. For example: get the versioning working and push to the artifact repository before deploying into a development environment. Verifying the steps individually will make assembling the entire pipeline much easier.

Implement the prescriptive guidance included in this paper gradually. Build on this success incrementally into other areas. Fundamentally, this approach models what you're trying to achieve in your software development. Smaller and more frequent deployments allow you to figure out what's working well, and what can be optimized. The same is true for your cultural and organizational changes. It's all about Lean Software Development, right?

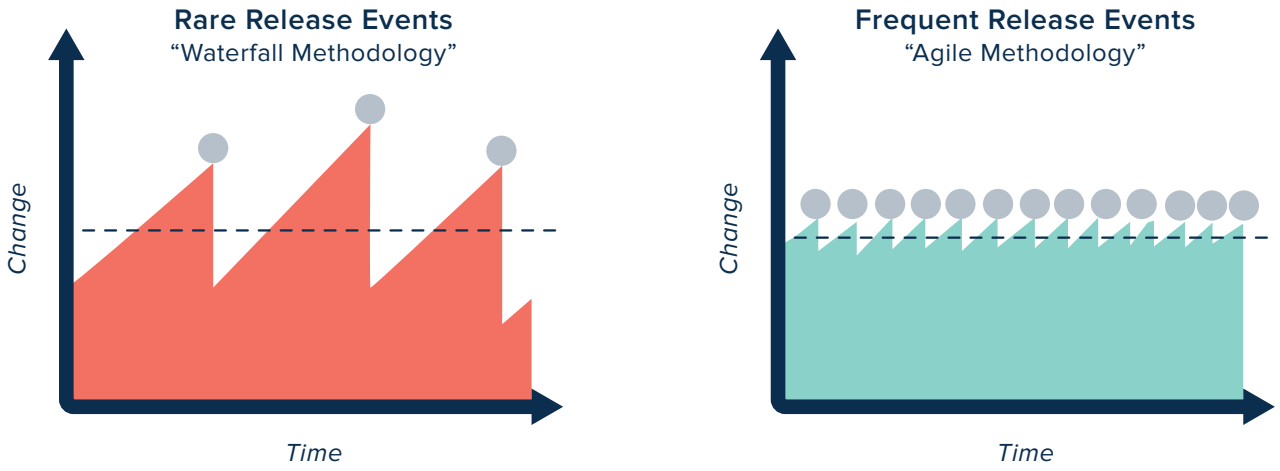
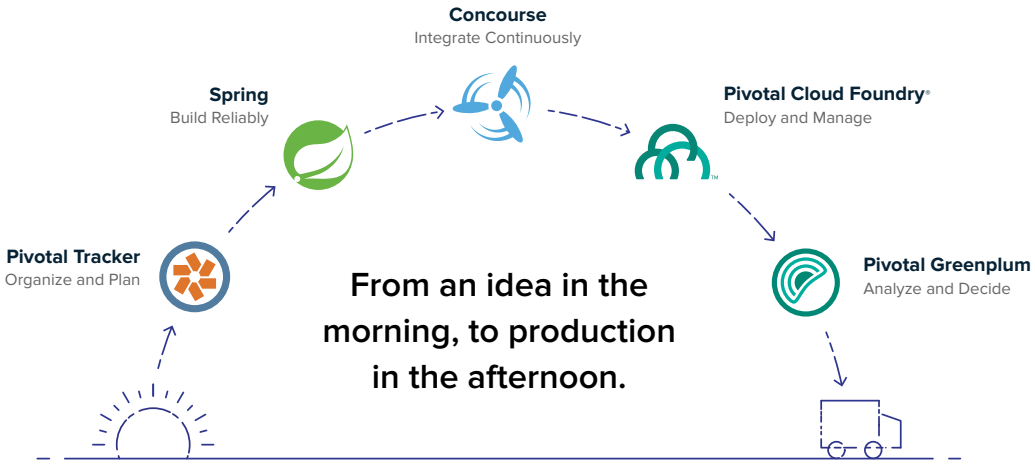


Figure 3: Comparing Waterfall and Agile Methodologies

The Circle of Code

The “Circle of Code” - coined by Pivotal in 2016 - represents something you’ve known for a while: software has a lifecycle. This lifecycle is the time-to-value: how long it takes to get code into the hands of customers. So what’s that lifecycle in most organizations today? It’s likely a series of clumsy handoffs between independent teams and applications. As a result, it’s harder to get software into the hands of your customers. Pivotal believes that your ongoing software lifecycle should complement, not frustrate, your creative process. Continuous Integration and Continuous Delivery help you go **from an idea in the morning, to production in the afternoon.**



Recommended Reading

7 Principles of Lean Software Development

Pivotal Cloud Foundry Overview — Onsi Fakhouri [Video]

Continuous Delivery: Conception To Production In Pivotal Cloud Foundry

Continuously Deploying Pivotal Cloud Foundry

Continuous Deployment From GitHub To PWS Via Concourse

Cloud-Native: Designing Change Tolerant Software

Cloud Native - Topic Page

Unwinding Platform Complexity with Concourse

Install Concourse CI with BOSH

Appendix A: Learn from Your Peers - How the Best Companies Use CI / CD

Companies that embrace best practices for Continuous Integration and Continuous Delivery establish themselves as innovators. They are able to adapt quickly to market changes with minimal risk. Here's a look at how a few enterprises are make it happen.

Liberty Mutual

Liberty Mutual helps customers protect what they earn, build, own, and cherish. The century-old insurer has always known the faster it responds, the better it serves individuals and communities. That's why it engaged Pivotal. Together Liberty Mutual and Pivotal are speeding the insurer's software development processes by combining proven methodologies with in-depth industry experience to build the right software and deliver the right services, at the right time. Through market disruption, Liberty Mutual is leading the way in app development to ensure the company is always there when customers need it most.

The result? A stalwart, Fortune 500 company that's actually leading the way in app development to improve lives and communities. In other words, a total transformation that promises to disrupt the insurance industry from the inside.

Read more

Humana

Humana Inc., headquartered in Louisville, KY, is a leading health and well-being company founded in 1961. With a focus on helping people achieve their best health with clinical excellence through coordinated care, the company has approximately 50,000 associates and serves more than 14 million customers throughout the country.

A core mission at Humana is to help its 14 million customers enjoy better and healthier lives, and one way they accomplish this goal is by offering people interactive mobile applications that provide useful healthcare advice.

Today, Humana software designers use a variety of agile software development methodologies to streamline the software development cycle, including pair programming, test-driven development and a lean focus on products, from start to finish.

With new agile expertise, Humana and Pivotal teams have collaborated on developing the Humana Vitality mobile application, which allows consumers to create and measure particular personal wellness goals. They also worked together to quickly tackle a second application called My Health by Humana, a condition management mobile app that allows consumers to enter their blood pressure and their weight, and then receive tailored content to help them manage their health more effectively. By using Cloud Foundry at the back end, development on My Health by Humana took less than three months.

Read more

The GAP & Other Retailers

Perhaps no industry has faced as much upheaval in the last decade as retail. So how are brick-and-mortar brands responding? By transforming how they deliver software. Here's a look at the results The GAP as achieved with CI/CD:

"[Pivotal Cloud Foundry] improved our ability to deliver quickly... In the old way of doing things, we configured everything through Chef. We built servers. We had this dodgy build system. It really could take weeks to get the feature into production. Now with this tool set that we have along with this CI/CD capability, that really has accelerated that. We could push something in 5 minutes... It's been a dramatic improvement for us."

Philip Glebow
Director of Architecture, The GAP

Read more