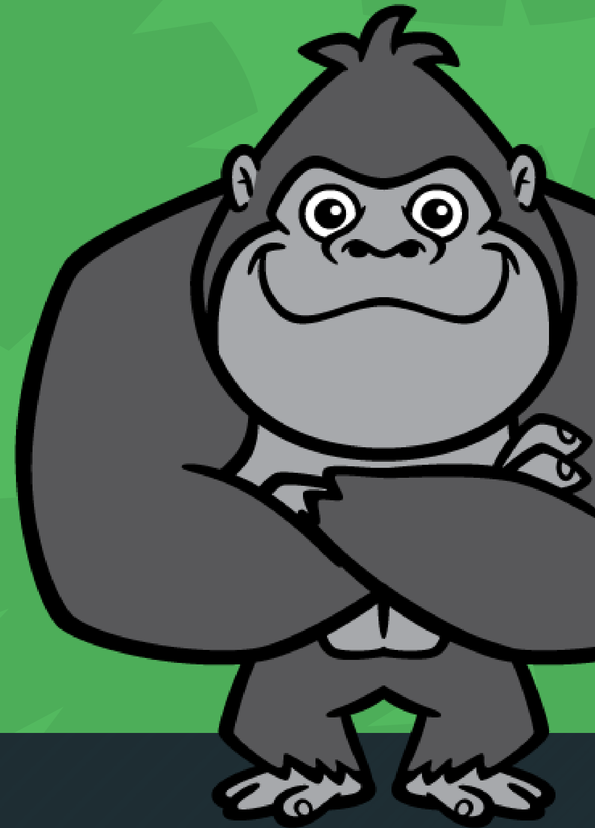
GORILLA GUIDE®
TECH BRIEF

ActualTech Media

DevOps Foundations: A Primer

Adam Bertram



In This Paper

DevOps is a key enabler for organizations looking to integrate their software delivery pipeline into their IT infrastructure as smoothly and quickly as possible. When done properly, it delivers great business value. The problem is, it often isn't done properly. Here's what you need to know.

Highlights include:

- Start with a plan, rather than just "buying tools"
- DevOps is a team effort, and trust is at the core
- What a proper "continuous integration/continuous delivery" process looks like

Contents

Easy Does It: 'Continuous' Is Key....2

Getting Started with Continuous Integration....4

The Last Mile: Continuous Deployment and Delivery....7

"DevOps" is certainly a catchy industry buzzword. Like many such buzzwords, though, the phrase may be common, but understanding what it really means, and how it could affect—and even disrupt—your data center and larger infrastructure is less well known.

This Gorilla Guide Tech Brief aims to help fill in that knowledge gap. It goes over the fundamentals of DevOps, including what it *actually* is, how to properly prepare your environment for it, and some of the core concepts surrounding it.

> The first, most important thing to understand is that DevOps streamlines IT operations via automation.

When you're done, you'll have a much better grasp of DevOps, and likely want to find a way to integrate at least some parts of it into your processes, as it has numerous advantages.

Easy Does It: 'Continuous' Is Key

The first, most important thing to understand is that DevOps streamlines IT operations via automation. Continuously delivering value to end users, the business, and the rest of the teams inside an organization has always been crucial. But why are we hearing about it so much now? What is this whole *DevOps* thing, and does it live up to the hype?

DevOps, and its set of continuous processes, known as *continuous integration*, *continuous delivery*, and *continuous deployment*, are hot. Why? Because it delivers continuous value to your customer.

But to get there, organizations must start slow to move fast. It may sound contradictory, but it isn't. It means that if you're going to do this "continuous" thing right, you need to go about it in a methodical, highly planned way. "Seat of the pants" IT won't get it done, in other words—that spaghetti won't stick to the wall.

If it's done right, the "continuous" process of DevOps will provide your customers value much, much faster than you could possibly do it manually. Value ASAP. And that value doesn't just end, as you'll be delivering fixes, updates, and improvements on a smooth, predictable cadence, rather than huge "service pack"-style updates as in the past.

Organizations understand this, too, so they buy and implement scripting, automation, and testing tools, and cloud infrastructure. But without a solid plan, they often end up spending more time, not less, on the very tasks they're trying to avoid.

It's often not their fault. One of the biggest issues with tooling is the way companies market their products. Many vendors make it seem so easy to implement "the DevOps": "Just buy a tool and your problem is solved!" This sentiment is far from the truth.

Organizations need to learn what DevOps is from a cultural and process perspective *first*, instead of thinking a tool will solve all their problems. Before purchasing a CI/CD tool, for instance, organizations should learn "The Three Ways," for example.

"The Three Ways" are the three pillars that support the DevOps foundation. The blog "The Three Ways: The Principles Underpinning DevOps" describes them as "the values and philosophies that frame the processes, procedures, practices of DevOps, as well as the prescriptive steps."

The pillars, in order, are:

- Systems thinking
- Amplify feedback loops
- A culture of continual experimentation and learning

The concepts for each pillar are more fully defined in the blog.

Once an organization understands how DevOps can transform its software deployment workflows, it's time to ask some hard questions and follow some best practices, including:

- Notice patterns. Why does that application build keep failing?
- Start paying back technical debt. What workarounds and hacks have you put in place just to get the job done? Organizations must first fix the underlying issues before even thinking about taking an application and sprinkling a little DevOps on it.
- Use one new tool at a time. How many shiny services and tools has your organization rolled out recently? You may not be properly taking the time to learn how each works. Start slow and don't get overwhelmed.
- When is the last time you updated documentation? Just because one or two people on your team know how things are done doesn't mean everyone else does. Document everything. Even include it with your builds, and make it a requirement for every new feature.
- Communication is key between both the engineers and management. Engineers have a habit of doing their own thing without regard to the business, but DevOps is all about business value. It's about bringing the business and the technology together. Bring in managers to your daily standup meeting, and build teams with representatives from all areas of the business to ensure everyone weighs in on changes.

If organizations start slow by learning not only tools but DevOps culture and how DevOps can deliver more business value, implementing one of the most important aspects of DevOps—the continuous pipeline—increases your chances of success.

CONTINUOUS INTEGRATION/ DELIVERY/DEPLOYMENT

One of the most important components of DevOps tooling is the pipeline. Referred to as the "build/release" or just "continuous integration/continuous delivery/deployment (CI/CD)" pipeline (**Figure 1**), this construct is automation on steroids—it's the crux of quickly delivering value to customers.

A CI/CD pipeline typically has two to three different stages or phases:

Continuous Integration

The first stage of a pipeline is CI. This is the practice of storing code in a repository that applies version control—Git, for instance—and automatically triggering a *build*. This build performs whatever necessary steps on the code to create an *artifact*, or a single unit, that can be deployed to an environment.

CI is also a great place to integrate testing. When a developer writes code, that code needs to be tested before allowing a customer to run it. Integrating automated testing in the CI phases allows organizations to build a *gate*. That gate will halt the stage unless all of the tests pass.

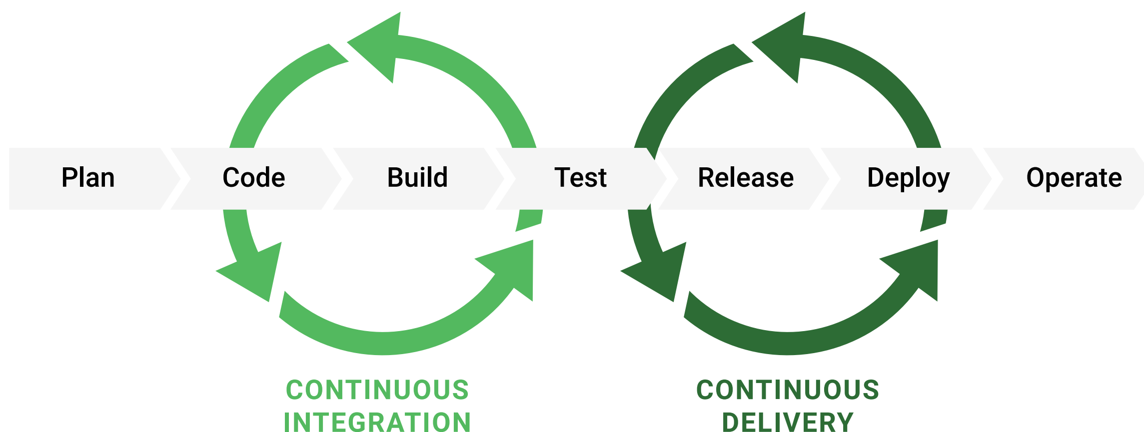


Figure 1: A typical continuous integration/continuous delivery pipeline

Continuous Delivery/Deployment

The second (and optional third) phase is continuous delivery/continuous deployment (CD). The terms are sometimes used interchangeably. Deciding when to use one or the other depends on the extent to which an organization plans on automating the entire software development and deployment process.

Although "continuous delivery" and "continuous deployment" are often considered synonyms, this isn't accurate. See The Last Mile: Continuous Deployment and Delivery for a detailed breakdown of the differences.

The CD stage typically takes the artifact that the CI process builds and automatically deploys that artifact to a testing environment.

CD then takes automation one step further by sending the artifact to infrastructure and installing it. Infrastructure could be a virtual machine, a container, some serverless model, or even bare metal servers.

Once the software is deployed to a testing environment, other teams like QA can perform manual testing or other final validation checks.

When everyone is happy with the final result, organizations can go the final mile and implement a continuous deployment phase. In this optional phase, the pipeline is 100% automated. Software is built, tested, and deployed in a test environment. The pipeline then runs various integration and acceptance tests in the test environment. If all tests pass, the software is immediately deployed to production, where it then goes through another round of automated tests. If all is well, the deployment is successful.

Mature DevOps organizations can deliver value to their customers lightning fast—in fact, some companies are building code with CI and deploying it with CD 50-plus times *per day*. Part of that is how quickly their processes can find and fix bugs. A complete CI/CD pipeline throws all manual steps out the window.

DEVOPS IS A MARATHON, NOT A SPRINT

If your organization develops software or wants a quicker way to deploy infrastructure and isn't using DevOps and CI/CD pipelines, you're missing out. These processes can transform organizations from slow-moving, unwieldy behemoths to nimble, agile gazelles. But first, teams must understand the underpinning of DevOps, instead of simply throwing tools at a problem.

CI/CD, through its many "continuous" layers, creates stepping stones to steadily implement more automation into the software development lifecycle. Start out slow. Document specific areas to improve on. Bring business stakeholders together and understand not just technical problems, but business problems. If you do that, finding and buying the right tool to build your CI/CD pipeline will be the easy part.

> One of the most important components of DevOps tooling is the pipeline.

Now that you understand why DevOps is changing the way companies do IT, it's time to dig into the details. The next section introduces the practical steps necessary to transform your business with DevOps.

Getting Started with Continuous Integration

The first section introduced DevOps and one of its key goals, a smooth CI/CD pipeline. Let's assume that you've studied, planned, and are ready to implement some tools. The first that most developers use is a source control system like GitHub.

That's a great first step. But now you have another problem: You and your team have to remember to kick off some other routines once new code is checked in. You may have to run tests on that code and launch a

build process that compiles the code, creates an artifact, and performs other mandatory tasks to get the code to a deployable state.

Doing that manually is at the very least challenging, and at most impossible to do right, especially in this age of quick development. What you need is to automate the next phase of the software development lifecycle (SDLC) as sown in **Figure 2**.

In other words, you need continuous integration (CI), which you were introduced to in the previous section.

WHAT IS CONTINUOUS INTEGRATION?

CI is an automated workflow that's automatically triggered by a source code "commit." That build process could be anything necessary to get the code into a deployable state, like compiling it, moving it to a specified location, and, typically, packaging it up.

The build process either creates some sort of artifact to be used later, or simply stores the code to be used later by the continuous deployment (CD) process.

CI is just the first step in the automation process of "continuous everything," but some would argue it's the most important. Why? Without CI, there's no code to deploy in the first place! The CI process grabs the code from source control and packages it up so the deployment process knows what to deliver.

A "commit," in software development terms, is when a change is made to a piece of software, and then uploaded to the source code repository, such as GitHub.

For example, let's say you have a C# web application you want to deploy to an Azure App Service for a front-end serverless application. The C# web application needs to get to the Azure App Service somehow. It needs to be deployed. But the deployment can't happen unless the C# web application runs through a build process to get it into a deployable state.

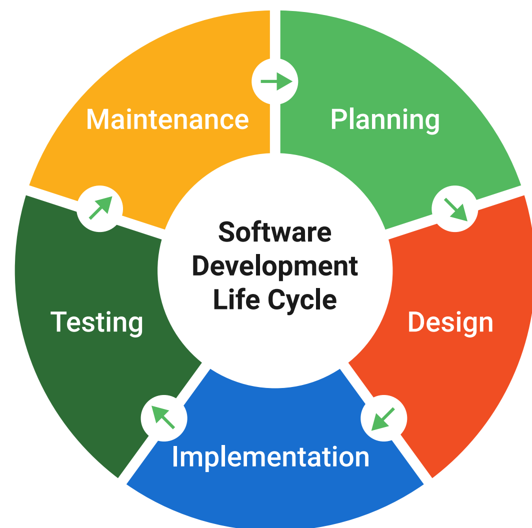


Figure 2: The software development lifecycle

TESTING WITH CONTINUOUS INTEGRATION

No one likes testing code manually, and forgetting to test or just being lazy can introduce bugs, security vulnerabilities, and more into your application.

Perhaps you're working on a web application that your team is continually updating. You should ensure *all* changes to the code have been tested. That's going to get old real quick if you must perform a set of tests for every code commit. Testing manually is not only cumbersome, but also prone to fail. We're humans and we make mistakes. That's where testing with CI comes into play.

Unit tests, which is a testing methodology that ensures code runs how you'd expect, is a common type of test suite to run. Unit tests are great to include in a CI process—once a CI process is set up, unit tests will run automatically, alleviating the task of manually running them.

Testing in CI is called Continuous Testing (CT). CT is crucial to building an overall, automated testing solution. CT eliminates several manual steps, including:

- Manual regression testing
- Manually building code to verify behavior
- Manual code inspections

Take a look at Figure 3, which shows a popular Microsoft CI tool called Azure DevOps. Perhaps you're working on a project written in PowerShell. Pester is a great testing tool for PowerShell. Pester allows you to write many types of unit tests for the PowerShell language.

Figure 3 shows a task in Azure DevOps called Pester powershell unittests, which could be configured to run Pester tests automatically as part of the CI pipeline. You can specify the test file pattern, any testing parameters, and tags.

Although building a CI process into your pipeline involves a lot of upfront planning and work, it saves considerable time and money over the long run.

This task in Azure DevOps is similar in concept to many other CI tools—it allows you to inject different tasks in an overall pipeline to run automatically.

HELPFUL CONTINUOUS INTEGRATION TOOLS

A CI pipeline is important for automated builds and software testing, and there are myriad options. Here are three of the most popular.

Jenkins is an open source CI/CD tool. It's been available for many years, and is one of the oldest CI tools out there. Because it's a plugin-based tool, a user can customize nearly every aspect of their platform using publicly available plugins. This customization ability of Jenkins is one of its key advantages.

The second tool on the list, and one taking the world by storm, is GitHub Actions. GitHub Actions is popular because it's free to use and embedded right into GitHub. GitHub is a hugely popular source control product—due to its popularity, it makes any CI product built directly on it a good CI option.

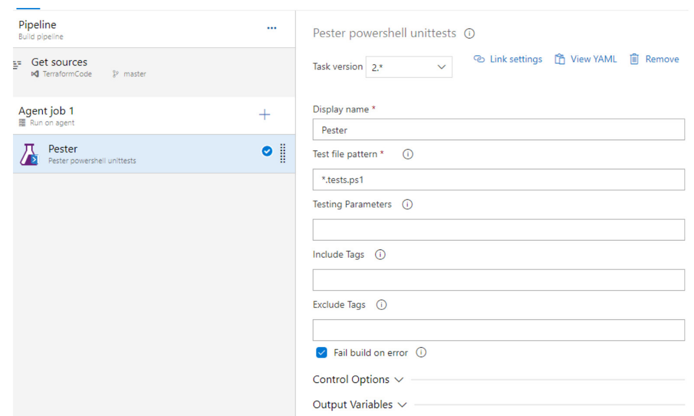


Figure 3: An Azure DevOps continuous integration tool, known as Pester Powershell unittests

GitHub Actions can build code, run tests, and deploy build artifacts if needed. It can do this on any platform right from GitHub. You might be at a disadvantage, though, if you and your team are using a competing source control product.

GitHub Actions is part of GitHub, the most popular code repository and developer collaboration platform in the world. GitHub has free and paid versions, and is used by roughly 50 million developers.

Azure DevOps is another great CI tool, and provides a complete CI solution. Azure DevOps, unlike GitHub Actions, can trigger a CI process from not only GitHub but also Azure's own source control product, Azure Repos, and many other source control products.

Azure DevOps has plenty of other tools built-in, as well, including a Wiki, ticketing system, and testing mechanisms. Azure DevOps is free for up to five projects, and you can create an organization with an Azure account.

START AUTOMATING NOW

Performing code builds manually is not only time consuming but error-prone. Although building a CI process into your pipeline involves a lot of upfront planning and work, it saves considerable time and money over the long run.

Now that you know how to get started with DevOps, it's time to move on to the ultimate goal for most organizations: A completely automated process for deployment and delivery of software.

The Last Mile: Continuous Deployment and Delivery

As you've learned, CI has huge benefits for an organization. The ability to commit code and have it built in a centralized location that all developers are committing to is the key to getting the code ready to use.

But CI is just the beginning. How do you deploy the software to perform further testing, and finally deploy it to a production environment? That's where *continuous delivery* and *continuous deployment* come into play.

CONTINUOUS DELIVERY

Think about the last time you had to deploy software to an environment, as either a developer or IT admin. It's never as easy as just installing software. There are always different packages to install, configuration items to remember, and so on. Even with a document playbook, we're human and make mistakes. And these mistakes are compounded by the increased frequency of software updates these days.

> The continuous delivery process takes the extra step and deploys the code to a testing environment, making it available for someone else to perform other tests on.

Eliminating those errors involves taking the next step in automation and applying *continuous delivery*.

For example, let's say you're a server admin and a developer shows you the location of a new web application running in Internet Information Services (IIS) on a Windows server. A typical IIS web application requires IIS applications, application pools, binding, SSL certificates, and more. To increase efficiency, you take the next step in automation and create a script for all these things. That's good, but you still have to run that script manually.

> **A foundational book on building great DevOps** is "The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations." It's full of great instruction and advice. It's considered by many to be the definitive guide to DevOps.

Using continuous delivery, you could leverage that automation code you wrote and integrate it directly into the CI pipeline. This ensures the required configuration is applied to a server. The integration saves time for both you and the developer, who doesn't need to request a manual deployment.

The continuous delivery process takes the extra step and deploys the code to a testing environment, making it available for someone else to perform other tests on.

CONTINUOUS DEPLOYMENT

Continuous deployment is similar to continuous delivery, but not exactly the same. Continuous deployment takes the next—and final—step of deploying code to an environment. Continuous delivery, on the other hand, gets the code up to that point and waits for approval. In continuous deployment, the entire process is performed automatically in a continuous deployment pipeline, with no approvals whatsoever.

There might be a schedule in place, or perhaps the deployment is done immediately. The automated deployment process can occur in any environment, including a testing or production environment.

Code testing is one of the most important aspects of continuous deployment apart from automating

CONTINUOUS INTEGRATION

CONTINUOUS DELIVERY

Figure 4: The difference between continuous delivery and continuous deployment is subtle, but vital

application delivery. Automated testing often means the difference between deploying an application *with* or *without* bugs. A bug can easily take down a production system if it's not tested before deployment.

A correct implementation of continuous is achieved when the process is trusted and there are no occurring issues like a system crash during deployment. Trust is a huge factor in a continuous deployment pipeline.

One useful tool for the "Dev" part of "DevOps" is "smoke tests," which are designed to find weaknesses in a new piece of code, like errors or other bugs that will keep your program from running properly. Smoke tests can find flaws that may not show up any other way.

The CircleCI blog "Smoke Tests in CI/CD Pipelines" provides a good overview of what smoke tests are, with real-world examples.

DELIVERY VS. DEPLOYMENT

At this point, you may be wondering what the differences are between continuous deployment and continuous delivery. From an automation perspective, there's one critical difference, shown in **Figure 4**.

Think about the two processes in terms of delivering a package to your door. You can think of continuous *delivery* as when the delivery driver drops off the package. They won't go into your house, of course, but you're not getting value from that package until it enters. And it's not entering until you approve it.

Continuous *deployment*, on the other hand, is equivalent to allowing the driver to open your door, set the package inside your house and leave. The package has been delivered to its final destination (inside your house).

Sounds simple, right? But this difference is rarely simple in the real world. For an organization to go from continuous delivery to continuous deployment requires a well-developed culture of monitoring, on-call support, and having the ability to recover an environment quickly.

> **Trust is a huge factor in a continuous deployment pipeline.**

And the culture doesn't apply to just one team—every team has responsibility for this. Continuous deployment requires everyone—front-end devs, back-end devs, infrastructure, cloud engineering, and every other IT team—to be on top of how the application is integrated into an automated process. You can't just throw some automation code into a pipeline and call it a day.

Continuous deployment is the last step in the CI/CD journey. If an organization trusts the process, it can eventually reach 100% automation.

In other cases it may not, however, and by design. Because of the thorough requirements to ensure an application's resiliency and the planning that must go into the process ahead of time, many organizations choose to stay at the continuous delivery step.

> **For an organization to go from continuous delivery to continuous deployment requires a well-developed culture of monitoring, on-call support, and having the ability to recover an environment quickly.**

ENJOY THE DEVOPS JOURNEY

That wraps up this introduction to DevOps. Both continuous delivery and continuous deployment help complete the journey to automating an app's entire deployment process, seamlessly combining both Dev and Ops into something resembling a single team with a single goal.

It doesn't happen, though, without careful planning, buy-in from all stakeholders, and the right automation tools.

Which process an organization follows depends not only on the tools, but also trust. To achieve full automation of an important software application, organizations must trust the teams and the process, and continually learn. They must decide how far they're willing to go to deliver value to their customers.

More automation is always a good thing, but it has an upfront cost in terms of planning, team collaboration, and trust. If you're ready to pay that price, you'll find the benefits more than worth it.

Happy automating!