O'REILLY®

COMPLIMENTS OF LaunchDarkly

Continuous Integration vs. Continuous Delivery vs. Continuous Deployment

Second Edition

The Processes & Tools of Effective Continuous Delivery Pipelines

Brent Laster

REPORT

SECOND EDITION

Continuous Integration vs. Continuous Delivery vs. Continuous Deployment

The Processes and Tools of Effective Continuous Delivery Pipelines

Brent Laster

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

Continuous Integration vs. Continuous Delivery vs. Continuous Deployment
by Brent Laster

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Mary Preap
Development Editor: Jeff Bleiel
Production Editor: Kristen Brown
Copyeditor: JM Olejarz
Interior Designer: David Futato
Cover Designer: Randy Comer

November 2017: First Edition
July 2020: Second Edition

Revision History for the Second Edition
2020-06-23: First Release

This work is part of a collaboration between O'Reilly and LaunchDarkly. See our statement of editorial independence.

978-1-492-08891-2

[LSI]

Table of Contents

Continuous Integration vs. Continuous Delivery vs. Continuous Deployment

Continuous Integration, Continuous Delivery, and Continuous Deployment—if you do any work in the area of software delivery, it's impossible not to encounter these terms on a regular basis. But what does each of these processes do for your product development and release cycles? I'll explain what they really boil down to, what practices are associated with them, and what kinds of tooling are available to implement them. I'll show what they offer, how they differ, and how they help, both separately and together, companies release software to customers frequently, reliably, and with high quality. And I'll show you how the core practices of DevOps fit in with all of these.

Armed with an understanding of those terms, I'll then explain how these processes fit into the modern environments of containers that run in managed clusters in the clouds.

A software delivery pipeline generates releases from source code in a fast, automated, and reproducible manner. The overall design for how this is done is called Continuous Delivery. The process that kicks off the assembly line by feeding in raw materials is referred to as Continuous Integration. The process that ensures quality is called Continuous Testing, and the process that makes the end product available to users is called Continuous Deployment. Done correctly, such a pipeline minimizes barriers, handoffs, and confusion among development teams and operational teams—goals that are part of the DevOps approach to software delivery.

Defining "Continuous"

Continuous doesn't mean "always running." It does mean "always ready to run." In the context of creating software, that includes several core concepts and best practices. These are:

Frequent releases
: One of the benefits of adopting continuous practices is enabling the delivery of quality software at frequent intervals. Frequency here is variable and can be defined by the team or company. For some products, once a quarter, month, week, or day may be frequent enough. For others, multiple times a day may be desired and doable. Regardless of the frequency, the goal is the same: deliver software updates of high quality to end users in a repeatable, reliable process.

Automated processes
: A key part of enabling this frequency is having automated processes to handle nearly all aspects of software production. This includes building, testing, analysis, versioning, and, in some cases, deployment. It may not be possible to automate everything, and some things, such as user acceptance, may need to remain manual or may seem too difficult to automate. But adopting an "automate everything" mindset will often inspire new ways of thinking about current processes that can surface new approaches for automation.

Repeatability
: If we have processes that always have the same behavior given the same inputs, then we can automate the processes so they are repeatable. This means if we go back and provide a certain version of code as an input, we should get the corresponding set of artifacts. Having the artifacts exactly match may also depend on having the corresponding versions of any external dependencies available to be included. With DevOps principles, this would also mean that the processes in our pipelines can be versioned and re-created.

Fast processing
: "Fast" is a relative term here. Regardless of the frequency of software updates/releases, continuous processes are expected to transform source code changes into updated deliverables in an efficient manner. Automation takes care of much of this. But

automated processes may still be slow. For example, integrated testing that takes most of the day may be too slow for product updates that have a new candidate release multiple times per day.

Fast problem detection and remediation
: One of a pipeline's jobs is to quickly process changes. Another is to monitor the different tasks/jobs that create the release. A third is to notify developers of failures as soon as possible since code that doesn't compile,[1] or fails a test, can keep the pipeline from proceeding. A secondary benefit of notifying developers quickly is that they will still have the context of the change fresh in their minds.

Continuous implies that a change pushed into source control can proceed, with little or no human intervention,[2] through the stages of builds, testing, packaging, and so on. And combined, these stages form a pipeline of processes and applications to take changes and turn them into a releasable product. This pipeline can go by various names, including *Continuous Delivery pipeline*, *deployment pipeline*, and *release pipeline*. Although handoffs between stages are intended to be automatic as the sequence progresses, there are a few cases in which human intervention can be required:

- When a change breaks and should be fixed before proceeding
- When you want a human to validate something before it proceeds (e.g., user acceptance testing or choosing to manually approve deploying to customers)
- When an automated process breaks
- When a process or workflow needs to change or be updated

With this background in mind, let's look closer at what each of the three terms encompasses.

1 "Fail fast" means that the pipeline processing finds problems as soon as possible and quickly notifies developer(s). Those developer(s) should then correct the problem ASAP and submit the updated code for another run through the pipeline.

2 Automated processing should be able to proceed without human intervention. Human intervention may still be desired or needed at certain points, as discussed in this section.

Continuous Integration

In the *Continuous Integration* (CI)[3] phase, changes from a developer are merged and validated. The goal of CI is to quickly validate these pushed code changes. The intended outcome is to identify any problems in the code and automatically notify the developer. This helps ensure that the code base is not broken any longer than necessary. The CI process detects when code changes are made, and runs any associated build processes to prove the code changes are buildable. It can also run targeted testing to prove that the code changes work in isolation (function inputs produce the desired outputs, bad inputs are flagged appropriately, and so on).

Unit Tests

These targeted tests are called *unit tests*, and developers should be responsible for creating them. Developers know the contexts about what the individual blocks of code are supposed to be doing and can best align that with the test cases. In fact, one development approach, known as test-driven development (TDD), requires unit tests to be designed first—as a basis for clearly identifying what the code should do—before the code is written.

In the standard CI workflow, a developer creates or updates source code in their local working environment. They then add tests to ensure that a function or method works. Because code changes can be frequent and numerous, and because they are at the beginning of the pipeline, the unit tests must be fast to execute. They also must not depend on (or make calls to) other code that isn't directly accessible, and should not depend on external data sources or other modules, such as a database component. If such a dependency is required for the code to run, those resources can be mocked by using a stub. The mocked stub looks like the resources and can return values, but doesn't actually implement any functionality.

Typically, these tests take the form of asserting that a given set of inputs to a function or method do, or do not, produce a specific result. They generally also test that error conditions are properly

3 I will use "CI" as an abbreviation for Continuous Integration. But I will not use abbreviations for Continuous Delivery or Continuous Deployment because referring to either or both as CD would be confusing.

flagged and handled. Various unit testing frameworks, such as JUnit for Java development, are available to assist with this.

CI Tools

After completing the code and validating it with local testing, the developer can then push their code and unit tests into the shared source code repository. A variety of source code management systems, such as Git, are available for use today. If the code is pushed and does not merge cleanly with other users' changes, the developer must manually fix the conflicts and try again.

After the code is merged in, a CI tool such as Jenkins can detect or receive a notification that a change has been made in the source code repository. It will then automatically grab the latest set of code from the repository and attempt to build it and execute any associated tests. Jenkins can detect changes by regularly polling the source control system to check for changes. Or, it can be configured to do a scheduled, periodic build of whatever code is current. As another option, most source control systems can be set up to send a notification that a change has occurred to Jenkins or tools like it.

Prechecks

A variation on this simple workflow involves adding in "prechecks" on the code before it makes it all the way into the source code repository. A simple form of this might use hooks or triggers. Hooks and triggers are processes that can be run before or after a particular source control operation to execute a process. Another tool, Gerrit, can intercept pushes to a remote Git repository and wait for a person to sign off on a review of the code before allowing it to go into the repository. It can also initiate early Jenkins builds of the code at that point as another pass/fail check.

Public or enterprise hosting sites for Git source repositories, such as GitHub, GitLab, and Bitbucket, offer a further variation of prechecks. Users wanting to contribute changes to a hosted, shared source repository can "fork" the repository (make a copy of a Git repository in their own user space in the hosting site). They can then push candidate changes to their copy. To get it merged into the original repository, they can submit a "pull request" (also known as a "merge request"). This asks the owner of the original repository to review and merge in their code changes. Pull/merge requests are

connected to automatic checks such as builds or simple testing. Such connections provide some confidence that the change is workable and won't break the existing code base.

Most prechecks also include a way to review and comment on proposed code changes. Once changes are approved and merged, they can kick off the Continuous Delivery processes.

Continuous Delivery

Continuous Delivery refers to the chain of processes (the pipeline) that automatically gets code changes and runs them through build, test, packaging, and related operations to produce a deployable release. Typically, it does this without much or any human intervention.

Continuous Delivery takes the changes pulled in by CI and executes the rest of the pipeline processes to produce the deliverables. Optionally, it may trigger Continuous Deployment processes to make releases from the pipeline automatically available to users. The mechanism of Continuous Delivery is the Continuous Delivery pipeline, although it may be known by other names.

While we may think of the sole deliverable of a pipeline as being a deployable set of code, along the way there are key intermediate outputs. In fact, one of the key things that happens in the pipeline is that new changes (validated and merged during CI) are combined with other code that they need to work with (or that they may be dependent upon) to produce artifacts. The management of these is worth exploring further.

Artifacts

An *artifact* is an item that is either a deliverable (something directly used by the final product) or included in a deliverable. For example, you might have an executable file created by compiling source that links in several other libraries. Or you might have a compressed file such as a WAR or ZIP file that contains another set of files within it.

The delivery pipeline is made up of stages connected in a sequence. Each stage in turn can be made up of smaller jobs with specialized tasks. For example, what's typically referred to as the *commit stage* usually consists of jobs to compile the code, run the unit tests, do integrated testing, gather metrics, and publish artifacts. Then there

is an automated handoff to an *acceptance stage* in which artifacts are retrieved and functional testing and verification are done.

Versioning

Regardless of an artifact's type, as it proceeds throughout the pipeline, it should be versioned via a process like Semantic Versioning. Although deployment technology has made overall product versioning less of a concern to users (think app updates on your phone), for developers, testers, and some automated processes, it isn't always desirable to update to the latest versions right away. Operations such as testing and debugging may require stable versions of artifacts that don't change every time the pipeline runs. And to track down issues completely, the versioning of the artifact should be able to be traced back to the exact set of source code that was used to produce it.

Versioned artifacts can be managed via an artifact repository tool such as Artifactory. This kind of tool works like a source management system for artifacts, allowing multiple versions to be stored and retrieved using different repositories. In some cases, these repositories can serve as a dev/test/prod (development quality/testing quality/production quality) organizational structure to distinguish multiple levels of releases that are in progress.

Promotion between levels and/or progress through the delivery pipeline is gated by testing. If incorrect versions of artifacts are pulled in or functionality is broken, the automated testing in the pipeline should detect this and raise an alert. Testing in the pipeline like this is referred to as Continuous Testing.

Continuous Testing

Continuous Testing refers to the practice of running automated tests or other types of analysis, of broadening scope as code goes through the CD pipeline. There are various forms of testing that can and should occur. These include the following:

- Unit testing is typically integrated with the build processes as part of the CI stage and focused on testing code in isolation from other code interacting with it.
- Integration testing validates that groups of components and services all work together.

- Functional testing validates that the result of executing functions in the product is as expected.
- Acceptance testing measures some characteristic of the system against acceptable criteria. Examples include performance, scalability, stress, and capacity.
- Coding metrics and analysis are not types of validation in the same way as pass/fail testing. But they can fit in the category of Continuous Testing because they assess the quality of code and testing. And they can be used to gate (block/pass) code at various points in the pipeline. Some quick examples in this category for checking metrics and analysis include:
  - Analyzing the amount of code covered by test cases. This metric is called *code coverage* and can be measured by tools (such as JaCoCo for Java code).
  - Counting lines of code, measuring complexity, and comparing coding structure and style against known best patterns can be done by tools such as SonarQube. Tools like this run checks, measure results against desired thresholds, block/allow further pipeline processing, and provide integrated reporting on the results.

All of these types of testing may not be present in an automated pipeline, and the lines between some of the different types can be blurry. But the goal of Continuous Testing in a delivery pipeline is always the same: to prove by successive levels of testing/analysis that the code is of sufficient quality to be used in the release that's in progress.

Continuous Deployment

Continuous Deployment refers to being able to take a release of code that has come out of the delivery pipeline and automatically make it available for end users. (A pipeline that includes this is usually called a deployment pipeline.) Depending on the way the code is intended to be "installed" by users, deployment may mean automatically deploying in a cloud, making an update available, updating a website, or simply updating the list of available releases.

As noted earlier, just because Continuous Deployment can be done doesn't mean that every set of deliverables coming out of a pipeline is always deployed or that new functionality is turned on. It means

that, via the pipeline, every set of deliverables is proven to be deployable through mechanisms such as Continuous Testing.

Having to roll back or undo a deployment to all users can be a costly situation (both technically and in the users' perception). So, whether or not a release from a pipeline run is deployed may be gated by human decisions. These can be based on various methods employed to "try out" a release before fully deploying it.

Numerous techniques have been developed to allow trying out deployments of new functionality and easily undoing them if issues are found. These are discussed in the following sections.

Blue/Green Testing and Deployments

In this approach to deploying software, two identical hosting environments are maintained—a blue one and a green one. (The colors are not significant and only serve as identifiers.) At any given point, one of these is the production deployment (released to customers) and the other is the candidate deployment (being prepped for release).

In front of these hosting environments is a router or other system that serves as the customer "gateway" to the product or application. By pointing the router to the desired blue or green instance, customer traffic can be directed to the desired deployment. In this way, swapping out which deployment instance is pointed to (blue or green) is quick, easy, and transparent to the user.

When a new release is ready for testing, it can be deployed to the candidate environment. After it's been tested and approved, the router can be changed to point the incoming production traffic to it (so it becomes the new production site). Now the hosting environment that was production is available for the next candidate.

Likewise, if a problem is found with the latest deployment and the previous production instance is still in place in the other environment, a simple change can point the customer traffic back to the previous instance. This action effectively takes the instance with the problem "offline" and rolls back to the previous version. The new deployment with the problem can then be fixed in the other area.

Canary Testing/Deployment

In some cases, swapping out the entire deployment via a blue/green environment may not be workable or desired. Another approach is known as canary testing/deployment/release. In this model, a portion of incoming traffic is rerouted to new pieces of the product. For example, a new version of a search service in a product may be deployed alongside the current production one. Then, a small representative sample (say 10%) of search queries may be routed to the new version to test it out in a production environment.

If the new service handles the limited traffic with no problems, then more traffic may be routed to it over time. If no problems arise, then gradually, the amount of traffic routed to the new service can be increased until 100% of the traffic is going to it. This effectively "retires" the previous version of the service and puts the new version into effect for all customers.

Feature Toggles

For new functionality that may need to be easily backed out, in case a problem is found, developers can add a *feature toggle* (aka *feature flag*). A feature toggle allows code to be present in the product but not active until some external process signals it to be active. A similar process could also signal it to go inactive again.

One example is a software if-then switch in the code that only activates the code if a data value is set. This data value can be set in a globally accessible location that the deployed application checks to see whether it should execute the new code. If the data value is set, it executes the code; if not, it doesn't.

If active at release, this gives developers a remote "kill switch" to turn off the new functionality if a problem is found after deployment to production.

Dark Launch

In this approach, code is incrementally tested/deployed into production, but changes are not made visible to users (thus the "dark" name). For example, in the production release, some portion of web queries might be redirected to a service that queries a new data source. Tracking information can be collected by development for

analysis—without exposing to users any information about the interface, transaction, or results.

The idea here is to get real information on how a candidate change would perform under a production load without impacting users or changing their experience. Over time, more load can be redirected until either a problem is found or the new functionality is deemed ready for all to use. Feature toggles can be used to handle the mechanics of dark launches.

Techniques such as the ones in this section are a key aspect of the continuous paradigm. But to get the complete set of benefits, they must be paired with team dynamics that support them across the organization.

DevOps: Bridging the Gap Between Teams

Historically, dev teams created products but did not install/deploy them in a customer-like way. The set of install/deploy tasks (as well as other support tasks) were left to ops teams to sort out late in the cycle. This often resulted in confusion and problems. The ops team was brought into the loop late in the cycle and had a short time frame to make what it was given work for customer environments. Dev teams were often left in a bad position as well, because they had not sufficiently tested the product's install/deploy functionality. So they could be surprised by problems that emerged during that process.

This situation often led to a serious disconnect and lack of cooperation between development and operations teams. DevOps is a set of ideas and recommended practices around making it easier for development (dev) and operations (ops) teams to work together on developing and releasing software, from the start of the cycle through the end.

The Continuous Delivery pipeline is an implementation of several DevOps ideals. The later stages of a product, such as packaging and deployment, can always be done on each run of the pipeline rather than waiting for a specific point in the product development cycle. And both dev and ops staff can clearly see when things work and when they don't, from development to deployment. For a run of the pipeline to be successful, it must execute not only the processes

associated with development, but also the ones associated with operations.

Carried to the next level, DevOps suggests that even the infrastructure that implements the pipeline be treated like code. That is, it should be automatically provisioned, trackable, and easy to change, and should spawn a new run of the pipeline if it changes. This can be done by implementing the approach of infrastructure as code.

Infrastructure as Code

In this approach, configuration and setup of the tools used in the pipeline, as well as the pipeline itself, are automated and described in files that can be stored in source control. A change to any of these files drives a reconfiguration or update of the tools used in the pipeline. The same change also triggers a new run of the pipeline, just as a source code change for the product code would.

In short, this implements CI not just for source code, but also for tools, the pipeline itself, and even configuration and testing data—if carried to its ideal. Tools that can help with this include common workflow tools such as Jenkins.

Today, thanks to advances in virtualizing systems and encapsulating applications and their runtime environments in containers, this ideal can be achieved.

Continuous Integration and Continuous Delivery in the World of Containers

In the not too distant past, individual hardware systems used in pipelines were configured with software (operating systems, applications, development tools, etc.) one at a time. At the extreme, each system could be a custom, handcrafted setup. This meant that when a system had problems or needed to be updated, that was often a custom task. This kind of approach goes against the fundamental Continuous Delivery and DevOps ideals of having easily reproducible and trackable environments.

Over the years, applications have been developed to standardize the provisioning (installing and configuring) of systems. Virtual machines (VMs) were developed as programs emulating computers running on top of other computers. These VMs require a

supervisory program to run them on the underlying host system. And they require their own copy of the operating system (OS) to run.

Next came containers. Containers, while similar in concept to VMs, work differently. Instead of requiring a separate program and a copy of an OS to run, they simply use some existing OS constructs to carve out isolated space in the operating system. Thus, they behave similarly to a VM to provide the isolation but don't require the overhead.

Side note: many products these days are deployed in containers. In such pipelines where containers are the intended delivery mechanisms, the artifacts created, managed, versioned, tested, etc. for eventual delivery/deployment may be containers themselves instead of the traditional binary modules we think of.

Because VMs and containers are created from stored definitions, they can be destroyed and re-created easily with no impact to the host systems where they are running. This allows the use of a re-creatable system to run pipelines on. Also, for containers, we can track changes to the definition file they are built from—just as we would for source code. The most common kind of definition file is a Dockerfile, used by a common container management application called Docker, to create an "image" that can then be launched as a container.

Thus, if we run into a problem (especially in a container), it may be easier and quicker to just destroy and re-create the container instead of trying to debug and make a fix to the existing one. As well, we can store the definition files (such as the Dockerfile) in source control and have them monitored by CI. If they change, a new image can be created based on the file and a new container can be created easily from the new image.

With containers, we can have the entire runtime environment and tooling for a process in our pipeline encapsulated. When we need that part of the pipeline to run, the orchestration process (such as Jenkins) gets the Dockerfile (or an existing image based on it). It then spins up a container to do the processing. When that part of the pipeline processing is done, the container can be deleted to save resources. But like managing multiple machines, managing multiple

containers for a group of processes can be challenging. That's where a tool called Kubernetes and its "cluster management" technology can help.

Kubernetes

Kubernetes is designed to automatically deploy, manage, and keep running large numbers of containers, and their workloads, on sets of machines working together as a cluster. You can think of Kubernetes as a data center for containers. Based on specifications provided to it by users, it ensures that a certain scale of containers are spun up and kept running. If something fails, it can start another instance. If demand increases, it can add more instances. If demand decreases, it can remove instances. It can provide virtual IP connections to pools of containerized workloads to ensure that, if one goes down, users are still connected. In short, it functions like a data center would for actual real computers, making sure there is availability, reliability, scalability, and consistency for large numbers of containers and their workloads. This all lends itself well to cloud environments.

Kubernetes is well suited for pipelines and CI/Continuous Delivery use cases. This kind of automatic, unattended management for applications is exactly what's needed in cloud environments. When applications are running in the cloud, we don't want to have to be manually monitoring, scaling, or restarting them. So it is becoming more common to manage and leverage cloud environments using applications like Kubernetes. In fact, every cloud provider has some implementation of a Kubernetes service that you can ask for.

A key aspect of leveraging cloud environments is cost. For any particular kind of processing in the pipeline, we can spin up and manage the workload in containers and delete them when done, thus limiting the use of resources and saving money. Also, companies do not have to provide and support their own resources for pipelines. They can farm that responsibility out to the cloud, using containers and Kubernetes to manage the processes for them once they are set up.

There can still be nontrivial work to set up pipelines that use containers and Kubernetes with the best practices of CI/Continuous Delivery and DevOps. But there are applications and tools that can help with this. One of the newest ones is Jenkins X, a way to set up and run automated, cloud-ready pipelines with a few basic steps.

Conclusion: Continuous Value

As you can see, Continuous Integration, Continuous Delivery, and Continuous Deployment form a powerful set of disciplines, best practices, and technologies for transforming changes from source to production quickly and reliably. The end result is high quality and is easily reproducible. This process not only benefits the customer, but also greatly simplifies the worlds of development, testing, and deployment teams. It lets the teams work together in a common environment from start to finish, helping to realize one of the core goals of the DevOps movement.

To get these benefits, Continuous Delivery pipelines (and related processes) can be created and run in many different forms. Adding in containers, and managing those containers in Kubernetes clusters, takes the benefits to another level—automating the setup and management of the pipeline and its processes. This reduces the need for custom hardware and support. It also allows for true DevOps infrastructure-as-code implementations.

Leveraging all of these continuous technologies, virtualization and containerization techniques, and DevOps practices can provide fast, disciplined, reproducible, easily managed, and cost-effective pipelines. Effectively implementing these will allow you and your organization to not have to continuously focus on your infrastructure. Instead, you can focus continuously on the most important goal—designing, implementing, and delivering quality software efficiently and reliably to your customers.

About the Author

Brent Laster is a global trainer, author, and speaker on open source technologies, as well as an R&D director at a top technology company. He has been involved in the software industry for more than 25 years, holding various technical and management positions.

Brent has always tried to make time to learn and develop both technical and leadership skills and share them with others. He believes that regardless of the topic or technology, there's no substitute for the excitement and sense of potential that come from providing others with the knowledge they need to accomplish their goals.