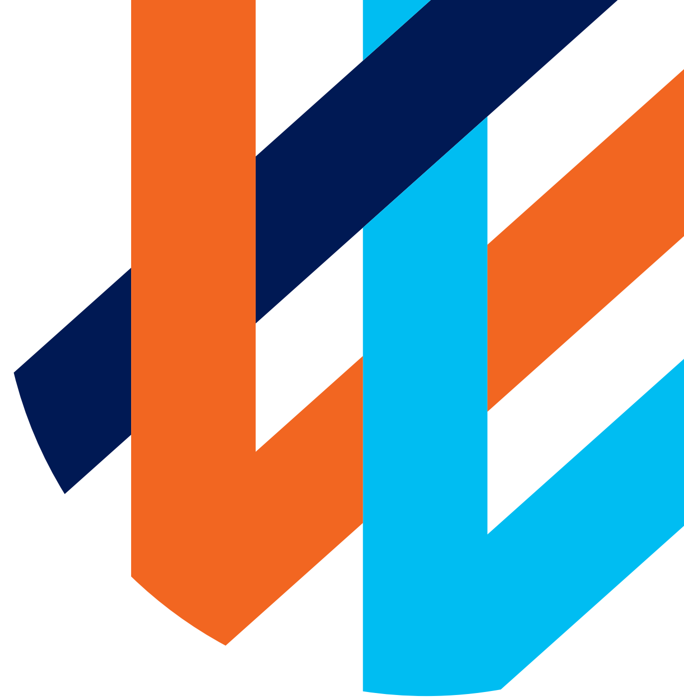
weaveworks



Making the Leap from Continuous Integration to Continuous Delivery

WHITEPAPER

Continuous Integration/Continuous Delivery (or CI/CD) is the holy grail of DevOps. If you achieve CI/CD, you achieve software delivery nirvana—or so the traditional DevOps mantra goes.

In reality, the relationship between CI/CD and software delivery perfection is more complicated than many DevOps conversations suggest. The term CI/CD conflates Continuous Integration with Continuous Delivery, implying that CI automatically leads to CD.

Conflating CI and CD is a mistake that will undercut your ability to achieve efficient software delivery. There is a tremendous gap separating CI from CD. CI is only one of several steps and practices that DevOps teams need to implement in order to achieve CD (which should be the goal of any modern software delivery team).

As a result, despite the frequency with which DevOps practitioners use the term CI/CD, many DevOps teams fall short of CD. They perform only CI, and miss out on the benefits of a complete CD operation.

The challenges that organizations face in making the leap from CI to CD are many. They lack the right tooling. They are hampered by a risk-averse culture that makes CD difficult to implement. They orient their software delivery strategy around outdated concepts, and they overlook the importance of visibility and observability in achieving a complete CD pipeline.

This whitepaper details the hurdles that DevOps teams must clear in order to move from CI to CD and identifies best practices for making the difficult leap. It is designed as a resource for DevOps practitioners who want to take full advantage of the efficiencies and operational advantages that CD enables, yet struggle to overcome the conceptual, cultural and technological challenges that complicate the transition from CI to CD.

THE BENEFITS OF CONTINUOUS DELIVERY

The first step in evolving your software delivery process from CI to CD is recognize the benefits that CD provides. Doing so will help your organization to identify the goals it aims to achieve through its transition to CD.

A well-designed CD operation will enable the following.

✓ Faster (and More Secure) Software Delivery

A primary goal of CD is to enable software updates to be written, tested, built and deployed into production on a rapid, continuous basis. Rather than requiring end users to wait months or years between updates to the applications they use, CD makes it possible to deliver new features and patches on a daily or even hourly basis.

Faster software updates lead to happier users, as well as applications that can evolve rapidly to meet shifting demands in the marketplace.

In addition, rapid software delivery provides security benefits by making it possible to fix security vulnerabilities quickly. That is a key advantage in the current age of **ever-increasing rates of software security breaches**.

✓ Streamlined Software Workflows

Traditionally, software production involved disparate workflows. Developers wrote and integrated code in a process. Software testing was performed as a separate process. So was deployment into production, as well as monitoring.

The DevOps movement is a response to this siloed operational structure. DevOps aims to integrate all of the processes that are required to deliver software into a single workflow in which different teams collaborate seamlessly.

This type of workflow becomes possible only through CD. Relying on CI alone will lead to operations that remain fragmented and inefficient.

✓ Better Collaboration and Communication Across the Organization

When software delivery operations are integrated into a single CD pipeline, individuals and teams collaborate and communicate more effectively.

CD ensures that developers are constantly aware of, and can help contribute to the work that QA engineers and IT Ops admins are performing. It helps to prevent gaps or delays in communication that slow innovation, waste employees' time and hamper the experience of end users.

Maximal Efficiency through Automation

A complete CD pipeline enables as much automation as possible in software delivery processes. Rather than relying on manual hand-offs of tasks from one siloed team to another, or requiring manual feedback loops to ensure that developers are aware of the challenges faced by admins, CD automates workflows.

The result of automation is greater efficiency, a lower risk of problems stemming from human error and better use of staff members' time. With CD, your engineers can focus on innovating, rather than performing tedious manual tasks.

THE CHALLENGE OF ACHIEVING CD

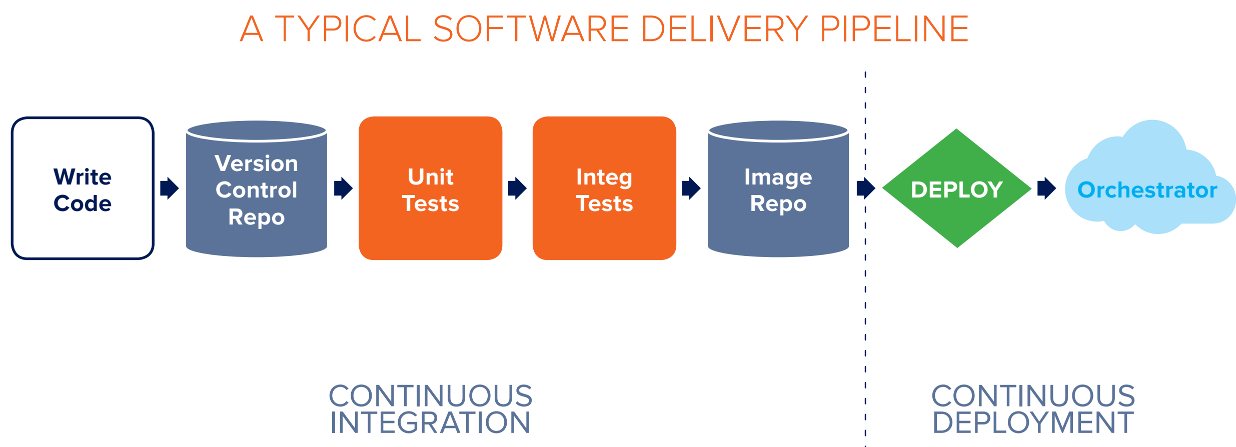
Despite the clear advantages of CD, and the popularity of the CI/CD concept within the DevOps world, DevOps teams today often fail to achieve full CD. They instead remain dependent on software delivery operations that are mostly limited to CI for the following reasons.

Confusing CI with CD

The fact that CI and CD are so frequently used in the same sentence makes it easy to conflate the terms, or assume that CI necessarily leads to CD.

This is a mistake. CI is only one step in a CD chain. CI servers automate the process of integrating code updates into an application's codebase, but they do nothing to address the other stages of the software delivery pipeline.

A TYPICAL SOFTWARE DELIVERY PIPELINE



CI servers are subject to other shortcomings, too. Their scripting is brittle and often breaks when applications are updated or new dependencies are introduced. CI tools don't automate deployment into environments like Kubernetes, the de facto management tool for modern software delivery. And in most cases, CI servers don't record changes, so the work they perform cannot be automatically reversed in the event that something goes wrong.

Every CD Pipeline is Unique

Attempts to evolve a software delivery operation from CI to CD are often hampered by a failure to recognize that there is no one-size-fits-all toolchain for building a CD pipeline. Instead, every

application is unique, and requires CD tooling adapted to its needs.

For example, the types of automated testing tools that you adopt to support CD are likely to be very different if the application you are delivering is traditional or web-based. Similarly, your release automation tools may vary depending on the types of operating systems and infrastructure (on-premises servers, the cloud, containers, virtual machines, or something else) that host your application.

Effective CD is never as simple as installing a specific set of tools. CD needs to be custom-built to support the application that it involves.

Overlooking Culture

Even if you have all of the right tools in place, you won't create an effective CD pipeline if you don't also make cultural changes.

A successful transition to CD requires building a culture of communication, collaboration and visibility. These values must be bricked into the way your engineers approach problems, no matter which specific role the engineers fill.

Fear of Risk

Innovation always entails some amount of risk, and migrating from existing CI routines to full CD is no exception. Risk-averse organizations may therefore be hesitant to embrace the change and disruption that are required for transitioning to CD.

This is especially true in situations where DevOps teams lack the visibility and metrics to determine whether their CD transition is successful, and the reversibility to revert tool and process changes if something goes wrong. As we explain below, however, visibility and reversibility can be achieved when you take the right approach to CD.

FIVE STEPS TO ACHIEVING CONTINUOUS DELIVERY

The challenges outlined above are real. They impede organizations every day from modernizing software delivery.

But with the right strategy, organizations can overcome these barriers and migrate from CI to CD. The following five steps are the keys to making an effective transition.

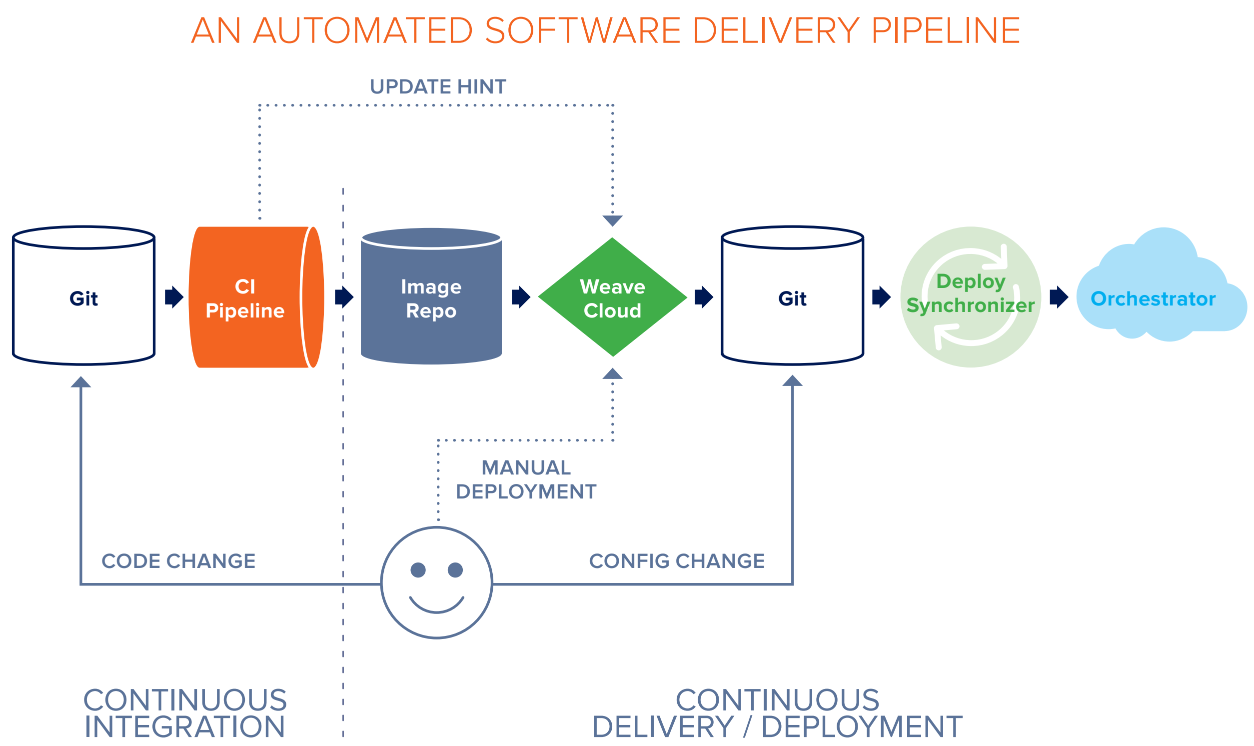
1 Fully Automate the Delivery Chain with Release Tools

Automating some parts of the software delivery chain (such as code integrations, builds and tests) is easy and obvious. These are the processes that organizations typically automate.

A successful CD pipeline, however, requires more automation than this. Your delivery chain should also include release tools to automate deployment, even if you deploy to multiple environments. Automated deployments should be tailored to your application architecture, which may be different for a microservices application than for a monolith.

Monitoring and observability must be automated, too. Manual monitoring is not enough. Neither is a monitoring strategy that incorporates some automated tools (such as Nagios) lacking the automated notifications and reports that you need to gain meaningful, real-time insights and observability for your environment.

Not every software delivery task can be fully automated, of course. You'll still need manual intervention by your engineers to respond to issues like failed integration tests or infrastructure problems in a production environment. Still, wherever there is an opportunity to automate, you should take it if you want to build a complete CD pipeline and optimize your operations.



2 Implement Application-Aware Release Automation

As noted above, applications vary widely, and so do the types of CD tools and processes that support them.

For this reason, the release management tools that you use to automate software delivery and build a CD chain should be tailored to your application's needs. They should be able to support the language or languages in which the application is written. If your application is stateful (meaning that it requires data storage), they must be equipped to handle data management. They should be designed to accommodate your application's architecture, whether it is monolithic or microservices-based.

In short, your release automation toolset should be application-aware.

3 Ensure Post-Deploy Validation

Deployment may appear to be the last major step in the software delivery chain, but it's not.

Monitoring your application after it has been deployed into a production environment in order to ensure that it performs as intended is a crucial additional step. Without post-deployment validation, you lack observability and the insights necessary to know whether your application is meeting its goals.

By implementing post-deploy validation, you can create a continuous feedback loop that enables developers to gain constant visibility into the state of an application in production, then use that information to plan new updates. Continuous feedback ensures that your developers' time is used most efficiently and that your engineers have the visibility necessary to deliver software that meets the needs of your business and users.

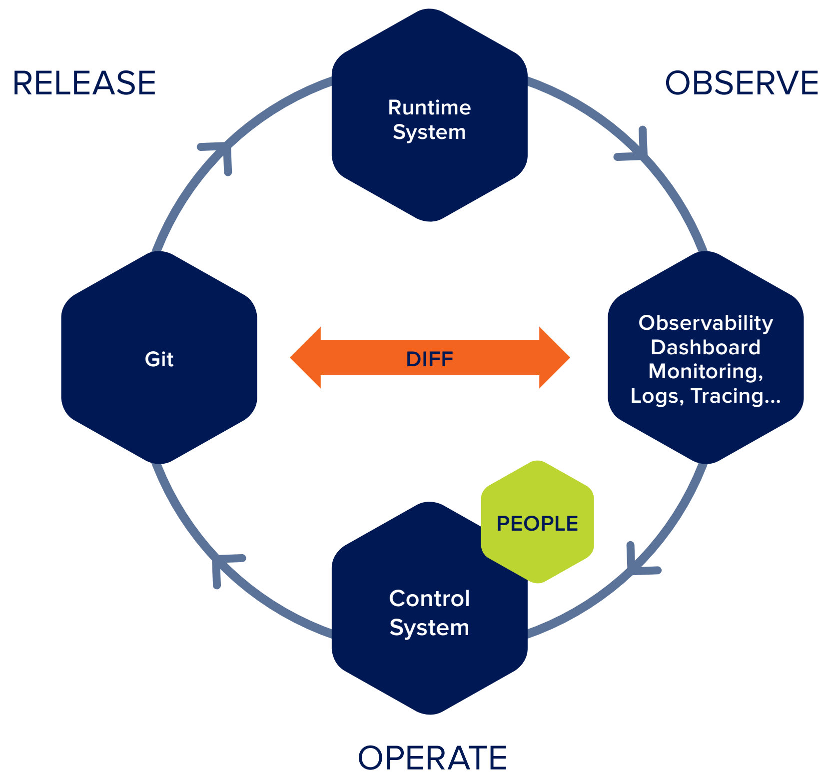
4 Implement Strong Alerting and Analytics

Observability is essential not just for the post-deployment stage of your CD pipeline, but also at all levels.

This fact can be easy to overlook. Most organizations focus on monitoring production environments, but assume that pre-production software does not require advanced monitoring, because failures and mistakes will not impact end users.

The reality is that the benefits of monitoring, alerting and analytics amount to more than simply mitigating user-impacting problems. Keeping your end users happy is one goal of monitoring, but it is only one. You also need analytics and alerts to notify you about software problems early in the pipeline because problems that are identified early are much easier and less costly to fix than those that are not discovered until your software is in production.

In addition, analytics can help you to improve the efficiency of your delivery process. For example, you might track the average time it takes to resolve different types of errors in order to identify pain points that your team can focus on improving in order to make the CD pipeline more efficient.



5 Embrace Fail-Forward Culture and Processes

As noted above, you can't achieve CD without taking risks. You can, however, plan for failures in ways that allow you to fail forward.

To do this, you need to build a culture that accepts risks as a part of innovation. You must also adopt tools and processes that allow you to revert when necessary in order to cope with failures.

Avoiding risk in software delivery may seem like a safer idea, but in the long run, it leads to a failure to innovate, and an inability to achieve CD.

CONCLUSION: WORKING TOWARD CONTINUOUS IMPROVEMENT

Each organization develops its own path to CD. There is no one-size-fits-all strategy for achieving CD.

However, the steps outlined above should serve as a guide for making the move from CI to full CD, and laying the foundations of a software delivery culture that prioritizes continuous improvement in all respects. Improving alerting, adding new and more intelligent forms of automation, validating workflows more effectively and embracing failure constructively are all components of a healthy software delivery strategy.

Continuous improvement requires the right cultural and philosophical underpinnings. But it also requires tooling that is designed to support the innovations described above. When it comes to tooling, solutions like Weave Cloud can automate and simplify the tasks required to build a complete CI/CD pipeline and embrace continuous improvement.

We invite you to **try Weave Cloud for free** today.