Building Continuous Delivery Pipelines

Deliver better features, faster

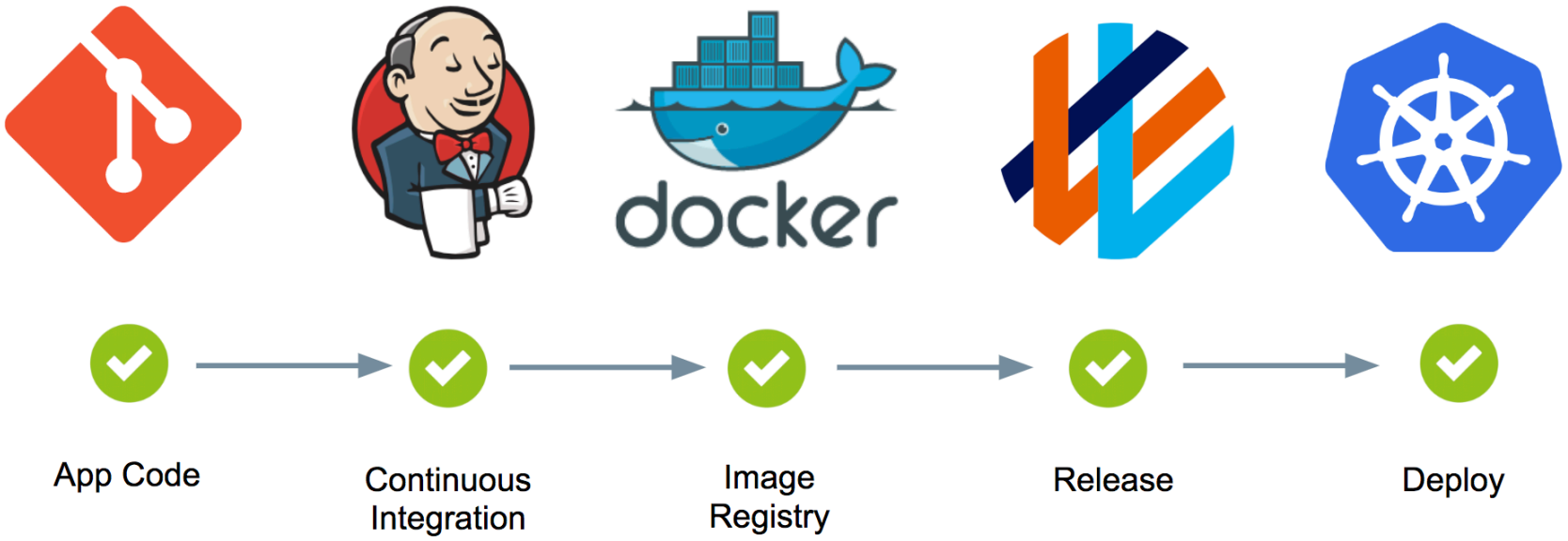
weaveworks

Table of Contents

Building Continuous Delivery Pipelines

A continuously automated flow of features is what distinguishes DevOps from other software development philosophies and practices, unlike the waterfall model where development follows an orderly sequence of stages. Continuous of course means without interruption, but that doesn't mean your engineers are working 24/7 updating code, or that they are deploying updates every time a line of code is changed. Rather "continuous" refers to software changes and new features rolling out on a regular basis.

By continuously pushing feature updates, businesses are more agile, can respond more quickly to customer demands and are more competitive in the marketplace.



CI/CD

Continuous Integration, Delivery and Deployment

CI/CD - Continuous Integration

The first step toward an automated pipeline is implementing **Continuous Integration**. With Continuous integration, developers are integrating small changes to the code base on a regular basis to ensure that their change hasn't broken the build.

When a developer commits a change to the code base, the CI system automatically builds and tests the changed code to make sure that it can safely merge into the main branch. Performing this integration check each time a developer commits changes, maintains the integrity of the code base and also minimizes the chance of a breaking change.

Continuous Delivery vs. Continuous Deployment

Continuous delivery takes the idea of continuous integration one step further and advances automation along the pipeline. With continuous delivery, code is not only integrated with changes on a regular basis, but as a second stage it is also deployed to a given environment, such as staging or production.

Some teams use the term “continuous delivery” interchangeably with the similar DevOps term “continuous deployment.” The difference between the two terms is subtle but important. Continuous delivery means that the updated code base is available to move onto the next development stage, whether that be staging, user review, or production.

Continuous deployment is a natural progression of continuous delivery. Once all software changes pass automated and functional testing, as well as the build stages, they are automatically deployed to production without human intervention. Continuously deploying code without human intervention is the goal that most companies hope to achieve.

METHOD	UPFRONT COSTS	WHAT YOU GAIN
Continuous Integration	• Unit tests need to be written for every new function and bug fix. • An integration server to manage the tests and to integrate with the code base. • Developers must check in code frequently.	• Low level errors are caught early before they hit production. • An integrated code base that ensures that the build always works. • Developers are alerted early when the build breaks, resulting in less context-switching for the team. • Massive reduction in testing costs.
Continuous Delivery	• Integration tests must have good code base coverage. • Fully automated deployments without human intervention should be possible. • Embrace feature flags so that incomplete features don't affect other developers on your team or customers in production.	• Increases the number of releases and features that you can deliver. • Features are iterated on more quickly, based on customer feedback, resulting in a more effective feedback loop.
Continuous Deployment	• Establish a good test process and have confidence in your automated test suites. • Make feature flags central to your deployments so that your company can easily co-ordinate new requests. • Other processes in the company need to keep up with the rapid pace of deployments.	• Since every change triggers the deployment pipeline, even faster releases are possible. • Releases are done in small batches so customers see improvements on a daily basis rather than quarterly or monthly.

BENEFITS OF AN AUTOMATED CI/CD PIPELINE

There Are a Number of Benefits to Automating Your CI/CD Pipeline

Increased Speed

Reduce your time to market from weeks and months to days or hours. With an automated pipeline, development teams improve both the velocity of releases as well as the quality of the code. New features, improvements and fixes are added continuously in small increments resulting in a product with fewer defects, allowing you to be more competitive.

A Strong Development Team

Because all stages of the delivery pipeline are available for anyone on the team to examine, improve upon and verify, a sense of ownership over the build is created that encourages strong teamwork and collaboration across your entire organization. This results in better communication between employees.

Reduced Risks and Costs

Automation encourages developers to verify code changes in stages before moving forward, reducing the chances of defects ending up in production. Also with an automated pipeline, deployments are more clearly decoupled from releases. This allows changes to be tested in multiple ways before they are deployed to production.

Less Work in Progress

A CD pipeline provides a rapid feedback loop starting from development through to your customers and around the loop again. This iteration cycle not only helps you build the right product, but it allows developers to make product improvements more quickly and also leaves less work in-progress to linger.

START YOUR AUTOMATION JOURNEY

Start Your Automation Journey

Introducing continuous delivery into your organization requires thought and planning as well as team commitment and buy-in. The transition can't be made overnight and it involves a number of intermediate and incremental steps to shift away from your current development methodology.

Depending on how far along you are on the Cloud Native journey and whether you are running a Kubernetes cluster on one of the public cloud providers or on-premise in your own data centre, there are several approaches to consider and plan for before fully automating your pipeline.

Step 1

Start Small

A mistake made by many organizations is that they take on too much too soon. Implementing CD is not easy, so it is always better to start off with a small proof of concept project before going ahead and converting your whole development processes all at once.

Step 2

Analyze Your Current Process

Write up your current development process so that you can see which procedures need to change and which ones can be automated. You may find that you can leverage existing CI tools that you already have in place and build out an automated pipeline from that.

Step 3

Create an Open Culture

Create transparency around the process of automation. Allow developers to make mistakes, so that gaps in the process can be addressed and corrected. This is also true once you've achieved automation where developers should have full ownership over the pipeline. In case a test fails, it should be possible to roll back quickly.

Step 4

Apply GitOps Principles

With 'declarative infrastructure' like Kubernetes cluster configuration, infrastructure can be kept in Git alongside your code. This not only means that there is a 'source of truth' for both your infrastructure and application code, but that when disaster strikes, your infrastructure can be quickly restored from Git. We discuss GitOps in more detail further on in this document.

Step 5

Measure Your Success

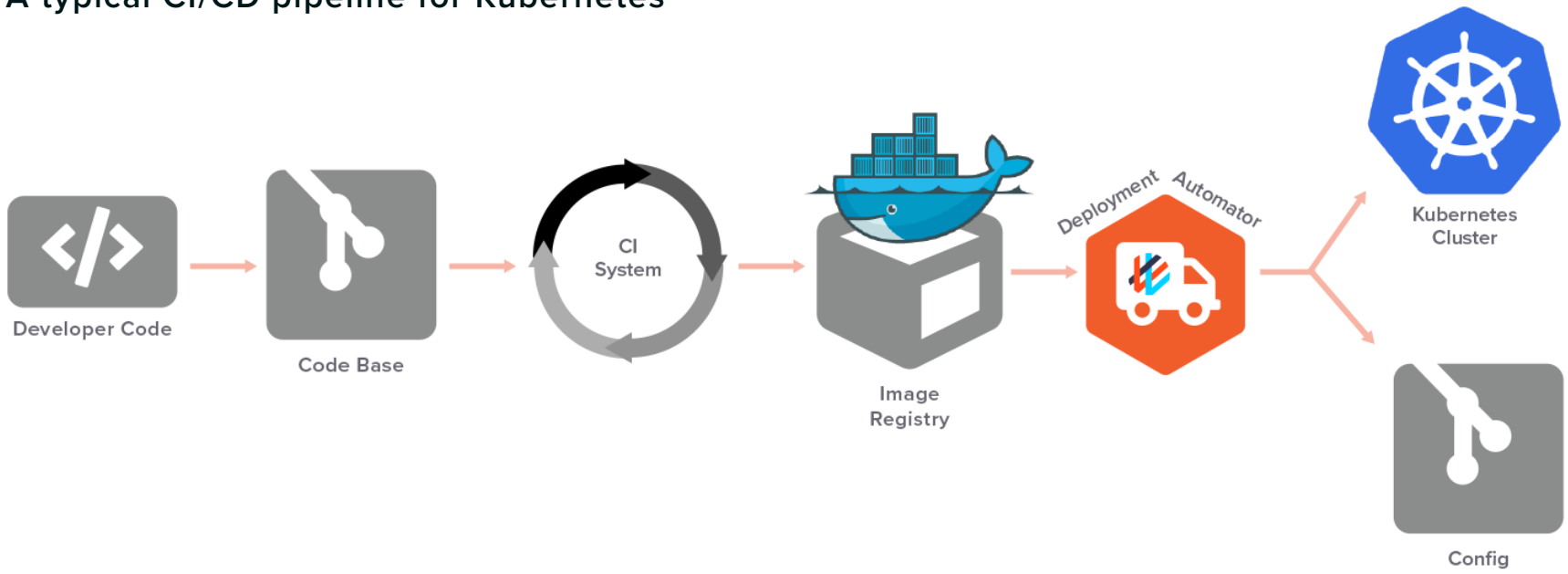
Define success metrics before you begin the transition to CD automation. It will allow you to consistently analyze your progress and help refining where necessary.

UNDERSTANDING PRODUCTION READY

Understanding Production Ready

A Continuous Delivery pipeline involves a number of stages — coding, commiting the code to source control, unit testing and integration, building and then deployment to production. Between each of these stages, code typically goes through many different suites of automated tests before the new feature lands in production.

A typical CI/CD pipeline for Kubernetes



3 Steps From Git To Production

STAGE 1

Continuous Integration and Build

The pipeline starts by the developer checking in code into Git which kicks off the CI tool. This consists of running some unit tests and integrating the changes into the rest of the code base before it builds a Docker image. This stage of the pipeline provides the first indicator of “health” of the new code.

STAGE 2

Deployment Automation

With the Docker container in a registry, the newly built container image runs through a series of automated tests: functional, security, compliance or performance. These tests provide further verification stages for the code change. And until you’ve achieved full automation, and you are confident in your tests and the process, some of these verification stages may need some human intervention.

STAGE 3

Test Automation and Verification

This is the final stage of the pipeline where the fully tested, and verified Docker image is deployed to a staging environment. At this point the change may be deployed to a subset of your production environment and initially monitored or it can undergo further user testing before being fully rolled out and released into production.

Deployments vs. Releases

An important distinction to be aware of is the difference between a deployment and a release. A deployment is when software has been tested and installed into a particular environment; whereas, a release is when those changes actually get into the hands of your end-users. Adding a deployment stage before you release to your customers allows you to do smoke tests, or more advanced testing like blue-green deployments, canary or A/B testing or you may want to deploy with feature flags.

SMOKE TEST: This test is essential if you want to ensure that the deployment was successful, and in particular to test that your configuration settings for the production environment are set up correctly.

BLUE-GREEN DEPLOYMENTS: This occurs when you run two identical deployments in parallel: one on production and another on a staging server. Final testing is done on one server and after testing, the live server is switched over from the tested version. If you need to rollback at a later point, this deployment strategy allows you to do that quickly and without any downtime.

CANARY DEPLOYMENTS: This is useful when you want to test out a new feature with a subset of users. Similar to a blue-green deployment, you might have two almost identical servers: one that goes to all users and another with the new features that gets rolled out to only a subset of users. These types of deployments allow you to test new features with a trusted group of users to ensure that you're building the right product.

A/B TESTING: This is a form of user testing where you compare two different layouts or designs on a subset of users to see which is the most effective for your use case.

FEATURE FLAGS: These are used when a feature may have passed the unit tests in the integration phase but is not fully complete. When a feature flag is turned on, it effectively hides it from the rest of the build. This allows the rest of the development team to release other complete features, without holding up a deployment or affecting customers in production.

Rollbacks

The goal is to be able to deploy a new change and get that change into the hands of your customers as quickly as possible. But equally important is that when defects are found, you can rollback the change so that it can be fixed. Rollbacks allow your team to more quickly iterate back and forth across the pipeline with feedback received via a failed test or from a customer or other stakeholder. At any point in the pipeline, developers must be able to smoothly rollback a change and get back to a stable state.

By implementing rollbacks, you and your team can benefit from increased confidence and speeding up your feedback loop.

Five Key Performance Indicators (KPIs) for Success

Deployment or Cycle Time

A term borrowed from manufacturing, “cycle time” is the time it takes for a feature to go from build to production. By measuring the phases of your development process, for example, coding, review process, test and release to production, the average cycle time is obtained. This metric provides insight into the bottlenecks in your process and the overall speed or time to deployment.

Deployment Frequency

Not only do more frequent deployments give you an opportunity to improve upon your software, but by measuring frequency, it allows you to analyze any bottlenecks you may find during your automation journey. The general rule is that more frequent smaller releases reduces the risk of defects, and maybe more importantly, increases the ability to fix them when found. This metric is an overall measure of your team’s efficiency and cohesiveness.

Change Lead Time

Measures the start of development to deployment. Like deployment frequency, this metric is also an indicator of your entire development process and how well your team works together. If the lead time is too long, it may suggest bottlenecks in your process or that your code and development systems are overly complicated.

Change Failure Rate

This metric focuses on the number of times your deployment was successful versus the number of times the deployment failed. This metric is one that should decrease over time. It is generally a useful measure of the success of your DevOps process.

MTTR vs. MTTF

The Mean Time to Recovery (MTTR) is the amount of time it takes for your team to recover from a failure; either a critical bug or a complete system failure. The Mean Time to Failure (MTTF) on the other hand measures the amount of time between fixes or outages. Both metrics are a reflection of your team’s ability to respond and fix problems. Generally you will want to see a downward trend for these metrics.

BEST PRACTICES FOR CONTINUOUS DELIVERY

Best Practices for Continuous Delivery

When you're ready to start putting the components together for your automated pipeline, you'll need to select the right tools for your team. Let's review a couple of best practices when it comes to Mean Time to Recovery (MTTR) and security.

Reduced Mean Time to Recovery with GitOps

Git enables declarative tools and for optimized Mean Time to Recovery (MTTR), you should implement GitOps.

GitOps works by using Git as a single source of truth for declarative infrastructure and applications within your entire system. Automated delivery pipelines roll out changes to your infrastructure when changes are made to Git.

Not only is there a 'source of truth' for both your infrastructure and application code, but when disaster strikes, infrastructure can be quickly and easily restored from Git, reducing your MTTR from hours to minutes.

For more on GitOps

See "**Operations by Pull Request**" and "**GitOps: High velocity CICD for Kubernetes**".

Best Practices for Continuous Delivery (cont')

Securing Your Pipeline

Push Pipelines

Most of the CI/CD tools available today use a push-based model. This means that code goes through the pipeline starting with the CI system and continues its path through a series of encoded scripts or by manually pushing changes to the Kubernetes cluster.

Push Type Patterns Can be Insecure

The reason you don't want to use your CI system as the deployment catalyst or do it manually on the command line is because of the potential to expose credentials outside of your cluster. While it is possible to make those interactions secure in both your CI/CD scripts and manually on the command line, you are working outside the trust domain of the cluster. This is generally not good practice and is why CI systems can be known as attack vectors for production.

Best Practices for Continuous Delivery (cont')

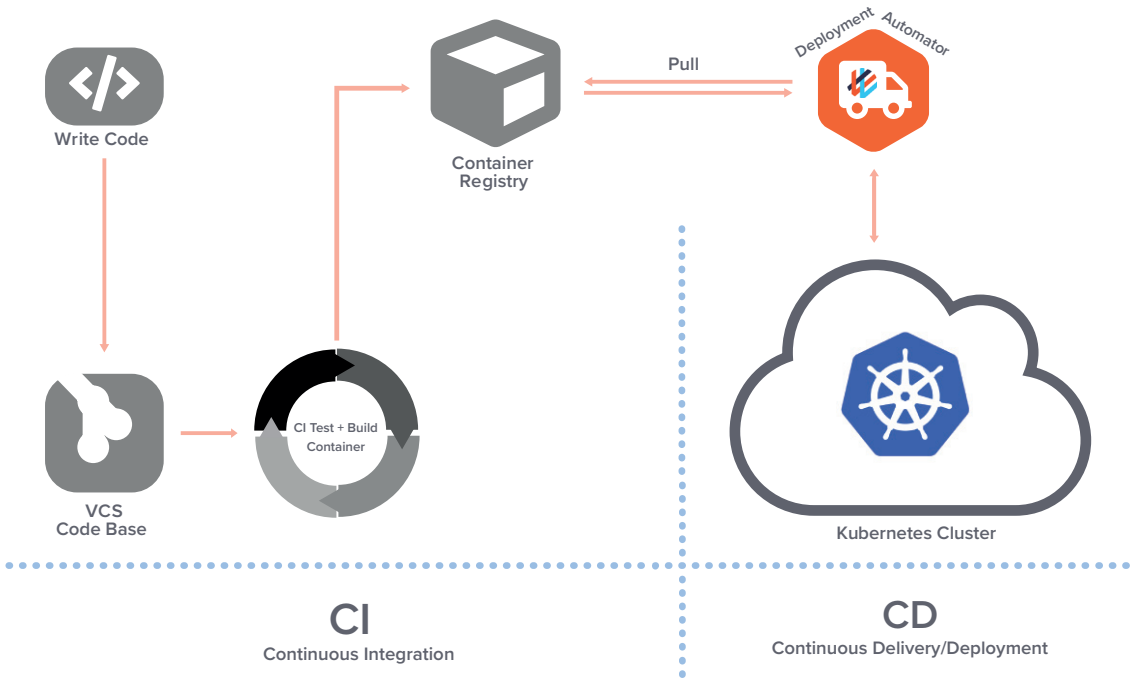
Pull Pipelines and Weave Cloud

Our product Weave Cloud, is a management and operations platform for containers. It uses a pull strategy and consists of two key components: a Deployment Automator that watches the image registry and a Deployment Synchronizer in the cluster to maintain its state.

At the centre of our pull pipeline pattern is Git (or any source control repo) that not only versions code, but also provides the single source of truth for the Kubernetes manifests. Developers push their updated code to the code base repository; where the change is picked up by the CI tool, tests are run and a Docker image is built and uploaded to an image repository.

This is where Weave Cloud's deployment features come into play. The deployment automator watches the image repository for new Docker containers, and when it sees one, it updates the manifest file in Git and automatically deploys the new image to the cluster.

A pull pipeline for CI/CD



Best Practices for Continuous Delivery (cont')

The Benefits of a Pull-Based Pipeline

- CI is decoupled from CD, and therefore the CI tool doesn't have elevated access to your cluster. Secure cluster credentials always remain with the cluster and are not widely shared across the pipeline or teams.

- Deployment policies can be set and changed (automatic or manual) in one place instead of keeping them scattered across your CI pipelines and embedded in custom scripts.

- Since all Kubernetes manifests are versioned in Git, if your cluster ever goes down, developers can restore everything and recover in minutes. Keeping manifests in Git also means that cluster configuration is always in-sync with what's being kept in Git.

CI/CD AND YOUR BUSINESS

CI/CD and Your Business

Achieving automated Continuous Delivery within your organization is not easy. The biggests hurdles you're likely to face are the cultural shifts that will need to be made within your organization to make the successful transition. Everyone within your company will be stakeholders in the new "Continuous Software" paradigm.

DevOps ≠ New Ops Team

One of the biggest misconceptions that some companies make is that they feel they need to create a 'DevOps team' that will implement and enforce best practices. For example, you don't want to transform your Ops team into the DevOps team. Instead, you want your Ops team to assist with the tools so that developers can be self-sufficient in their ability to deploy applications safely and securely to your cluster.

DevOps is less about forming yet another silo and is more about adopting a new philosophical approach to software development.

Cultural Impacts of CD

The success of DevOps depends a lot on company culture. Internal teams must be able to adopt cross-functional methods to ensure that software is iterated on with a continuous cadence but also complements the marketing and business goals of the company. Making the actual switch to continuous delivery may be the easiest part in your company; getting those changes to stick and also propagating them throughout your organization is probably the most difficult part of the process.

Culture and Your Journey to Automatic Continuous Delivery

Transparency

Everyone on the team should know how to fix or rollback a build, allowing them to take responsibility for the build and not leave it to one particular team. Creating cross-functional teams throughout your organization ensures that everyone has a stake in the outcome and new ideas are encouraged.

Ownership

While collaboration is important, it is equally important that developers take full ownership over their production-ready code. This is the DevOps mantra that says, 'you build it, you run it' and it means that developers should not have to rely on Ops or any other team to deploy their changes. The automation pipeline must be able to accommodate developers deploying code to clusters safely and securely.

Collaboration

Following on the notion of transparency, it's also important that when the pipeline breaks or if a bug is found, that teams can effectively work together to figure out the best person to make the fix.

Create a No Blame Atmosphere

Allow developers to make mistakes, so that gaps in the process can be addressed and corrected. Transitioning to continuous delivery can be challenging, you need to give your team the space to experiment and to fail before they will be successful.

About Weaveworks

Weaveworks makes it fast and simple for developers to build and operate containerized applications. The Weave Cloud operations-as-a-service platform provides a continuous delivery pipeline for building and operating applications, letting teams connect, monitor and manage microservices and containers on any server or public cloud. Weaveworks also contributes to several open source projects, including Weave Scope, Weave Cortex, and Weave Flux. It was one of the first members of the Cloud Native Computing Foundation. Founded in 2014, the company is backed by Google Ventures and Accel Partners and others.

Sign up today for a **free 30 day trial** for Weave Cloud.

Weaveworks now offers **Production Grade Kubernetes Support** for enterprises. For the past 3 years, Kubernetes has been powering Weave Cloud, our operations as a service offering, so we couldn't be more excited to share our knowledge and help teams embrace the benefits of cloud native tooling. **Contact us for more information**.