An Introduction to CI/CD Best Practices

Posted April 3, 2018 37.4k CI/CD CONCEPTUAL

By: Justin Ellingwood

Introduction

Continuous integration, delivery, and deployment, known collectively as CI/CD, is an integral part of modern development intended to reduce errors during integration and deployment while increasing project velocity. CI/CD is a philosophy and set of practices often augmented by robust tooling that emphasize automated testing at each stage of the software pipeline. By incorporating these ideas into your practice, you can reduce the time required to integrate changes for a release and thoroughly test each change before moving it into production.

CI/CD has many potential benefits, but successful implementation often requires a good deal of consideration. Deciding exactly how to use the tools and what changes you might need in your environments or processes can be challenging without extensive trial and error. However, while all implementations will be different, adhering to best practices can help you avoid common problems and attain improvements faster.

In this guide, we'll introduce some basic guidance on how to implement and maintain a CI/CD system to best serve your organization's needs. We'll cover a number of practices that will help you improve the effectiveness of your CI/CD service. Feel free to read through as written or skip ahead to areas that interest you.

Keep Your Pipelines Fast

CI/CD pipelines help shepherd changes through automated testing cycles, out to staging environments, and finally to production. The more comprehensive your testing pipelines are, the greater assurance you have that changes won't introduce unforeseen side effects into your production deployment. However, since each change must go through this process, keeping your pipelines fast and dependable is incredibly important to not inhibit development velocity.

The tension between these two requirements can be difficult to balance. There are some straightforward steps you can take to improve speed, like scaling out your CI/CD infrastructure and optimizing tests. However, as time goes on, you may be forced to make critical decisions about the relative value of different tests and the stage or order where they are run. Sometimes, paring down your test suite by removing tests

with low value or with indeterminate conclusions is the smartest way to maintain the speed required by a heavily used pipelines.

When making these significant decisions, make sure you understand and document the trade-offs you are making. Consult with team members and stakeholders to align the team's assumptions about what the test suite is responsible for and what the primary areas of focus should be.

Isolate and Secure Your CI/CD Environment

From an operational security standpoint, your CI/CD system represents some of the most critical infrastructure to protect. Since the CI/CD system has complete access to your codebase and credentials to deploy in various environments, it is essential to secure it to safeguard internal data and guarantee the integrity of your site or product. Due to its high value as a target, it is important to isolate and lock down your CI/CD as much as possible.

CI/CD systems should be deployed to internal, protected networks, unexposed to outside parties. Setting up VPNs or other network access control technology is recommended to ensure that only authenticated operators are able to access your system. Depending on the complexity of your network topology, your CI/CD system may need to access several different networks to deploy code to different environments. If not properly secured or isolated, attackers that gain access to one environment may be able to *island hop*, a technique used to expand access by taking advantage of more lenient internal networking rules, to gain access to other environments through weaknesses in your CI/CD servers.

The required isolation and security strategies will depend heavily on your network topology, infrastructure, and your management and development requirements. The important point to keep in mind is that your CI/CD systems are highly valuable targets and in many cases, they have a broad degree of access to your other vital systems. Shielding all external access to the servers and tightly controlling the types of internal access allowed will help reduce the risk of your CI/CD system being compromised.

Make the CI/CD Pipeline the Only Way to Deploy to Production

Part of what makes it possible for CI/CD to improve your development practices and code quality is that tooling often helps enforce best practices for testing and deployment. Promoting code through your CI/CD pipelines requires each change to demonstrate that it adheres to your organization's codified standards and procedures. Failures in a CI/CD pipeline are immediately visible and halt the advancement of the affected release to later stages of the cycle. This is a gatekeeping mechanism that safeguards the more important environments from untrusted code.

To realize these advantages, however, you need to be disciplined to ensure that every change to your production environment goes through your pipeline. The CI/CD pipeline should be the only mechanism by which code enters the production environment. This can happen automatically at the end of successfully testing with continuous deployment practices, or through a manual promotion of tested changes approved and made available by your CI/CD system.

Frequently, teams start using their pipelines for deployment, but begin making exceptions when problems occur and there is pressure to resolve them quickly. While downtime and other issues should be mitigated as soon as possible, it is important to understand that the CI/CD system is a good tool to ensure that your

changes are not introducing other bugs or further breaking the system. Putting your fix through the pipeline (or just using the CI/CD system to rollback) will also prevent the next deployment from erasing an ad hoc hotfix that was applied directly to production. The pipeline protects the validity of your deployments regardless of whether this was a regular, planned release, or a fast fix to resolve an ongoing issue. This use of the CI/CD system is yet another reason to work to keep your pipeline fast.

Maintain Parity with Production Wherever Possible

CI/CD pipelines promote changes through a series of test suites and deployment environments. Changes that pass the requirements of one stage are either automatically deployed or queued for manual deployment into more restrictive environments. Early stages are meant to prove that it's worthwhile to continue testing and pushing the changes closer to production.

For later stages especially, reproducing the production environment as closely as possible in the testing environments helps ensure that the tests accurately reflect how the change would behave in production. Significant differences between staging and production can allow problematic changes to be released that were never observed to be faulty in testing. The more differences between your live environment and the testing environment, the less your tests will measure how the code will perform when released.

Some differences between staging and production are expected, but keeping them manageable and making sure they are well-understood is essential. Some organizations use blue-green deployments to swap production traffic between two nearly identical environments that alternate between being designated production and staging. Less extreme strategies involved deploying the same configuration and infrastructure from production to your staging environment, but at a reduced scale. Items like network endpoints might differ between your environments, but parameterization of this type of variable data can help make sure that the code is consistent and that the environmental differences are well-defined.

Build Only Once and Promote the Result Through the Pipeline

A primary goal of a CI/CD pipeline is to build confidence in your changes and minimize the chance of unexpected impact. We discussed the importance of maintaining parity between environments, but one component of this is important enough to warrant extra attention. If your software requires a building, packaging, or bundling step, that step should be executed only once and the resulting output should be reused throughout the entire pipeline.

This guideline helps prevent problems that arise when software is compiled or packaged multiple times, allowing slight inconsistencies to be injected into the resulting artifacts. Building the software separately at each new stage can mean the tests in earlier environments weren't targeting the same software that will be deployed later, invalidating the results.

To avoid this problem, CI systems should include a build process as the first step in the pipeline that creates and packages the software in a clean environment. The resulting artifact should be versioned and uploaded to an artifact storage system to be pulled down by subsequent stages of the pipeline, ensuring that the build does not change as it progresses through the system.

Run Your Fastest Tests Early

While keeping your entire pipeline fast is a great general goal, parts of your test suite will inevitably be faster than others. Because the CI/CD system serves as a conduit for all changes entering your system, discovering failures as early as possible is important to minimize the resources devoted to problematic builds. To achieve this, prioritize and run your fastest tests first. Save complex, long-running tests until after you've validated the build with smaller, quick-running tests.

This strategy has a number of benefits that can help keep your CI/CD process healthy. It encourages you to understand the performance impact of individual tests, allows you to complete most of your tests early, and increases the likelihood of fast failures, which means that problematic changes can be reverted or fixed before blocking other members' work.

Test prioritization usually means running your project's unit tests first since those tend to be quick, isolated, and component focused. Afterwards, integration tests typically represent the next level of complexity and speed, followed by system-wide tests, and finally acceptance tests, which often require some level of human interaction.

Minimize Branching in Your Version Control System

One of the main principles of CI/CD is to integrate changes into the primary shared repository early and often. This helps avoid costly integration problems down the line when multiple developers attempt to merge large, divergent, and conflicting changes into the main branch of the repository in preparation for release. Typically, CI/CD systems are set to monitor and test the changes committed to only one or a few branches.

To take advantage of the benefits that CI provides, it is best to limit the number and scope of branches in your repository. Most implementations suggest that developers commit directly to the main branch or merge changes from their local branches in at least once a day.

Essentially, branches that are not being tracked by your CI/CD system contain untested code that should be regarded as a liability to your project's success and momentum. Minimizing branching to encourage early integration of different developers' code helps leverage the strengths of the system, and prevents developers from negating the advantages it provides.

Run Tests Locally Before Committing to the CI/CD Pipeline

Related to the earlier point about discovering failures early, developers should be encouraged to run as many tests locally prior to committing to the shared repository. This makes it possible to detect certain problematic changes before they block other team members. While the local developer environment will unlikely be able to run the entire test suite in a production-like environment, this extra step gives individuals more confidence that the changes they are making pass basic tests and are worth trying to integrate with the larger codebase.

To ensure that developers can test effectively on their own, your test suite should be runnable with a single command that can be run from any environment. The same command used by developers on their local machines should be used by the CI/CD system to kick off tests on code merged to the repository. Often, this is coordinated by providing a shell script or makefile to automate running the testing tools in a repeatable, predictable manner.

Run Tests in Ephemeral Environments When Possible

To help ensure that your tests run the same at various stages, it's often a good idea to use clean, ephemeral testing environments when possible. Usually, this means running tests in containers to abstract differences between the host systems and to provide a standard API for hooking together components at various scales. Since containers run with minimal state, residual side effects from testing are not inherited by subsequent runs of the test suite, which could taint the results.

Another benefit of containerized testing environments is the portability of your testing infrastructure. With containers, developers have an easier time replicating the configuration that will be used later on in the pipeline without having to either manually set up and maintain infrastructure or sacrifice environmental fidelity. Since containers can be spun up easily when needed and then destroyed, users can make fewer compromises with regard to the accuracy of their testing environment when running local tests. In general, using containers locks in some aspects of the runtime environment to help minimize differences between pipeline stages.

Conclusion

While each CI/CD implementation will be different, following some of these basic principles will help you avoid some common pitfalls and strengthen your testing and development practices. As with most aspects of continuous integration, a mixture of process, tooling, and habit will help make development changes more successful and impactful.

To learn more about general CI/CD practices and how to set up various CI/CD services, check out other articles with the CI/CD tag.

By: Justin Ellingwood

Related Tutorials

CI/CD Tools Comparison: Jenkins, GitLab CI, Buildbot, Drone, and Concourse

An Introduction to Continuous Integration, Delivery, and Deployment

How To Set Up Buildbot on FreeBSD

An Introduction to the Kubernetes DNS Service

Webinar Series: Building Blocks for Doing CI/CD with Kubernetes

0 Comments

Leave a comment...

Log In to Comment