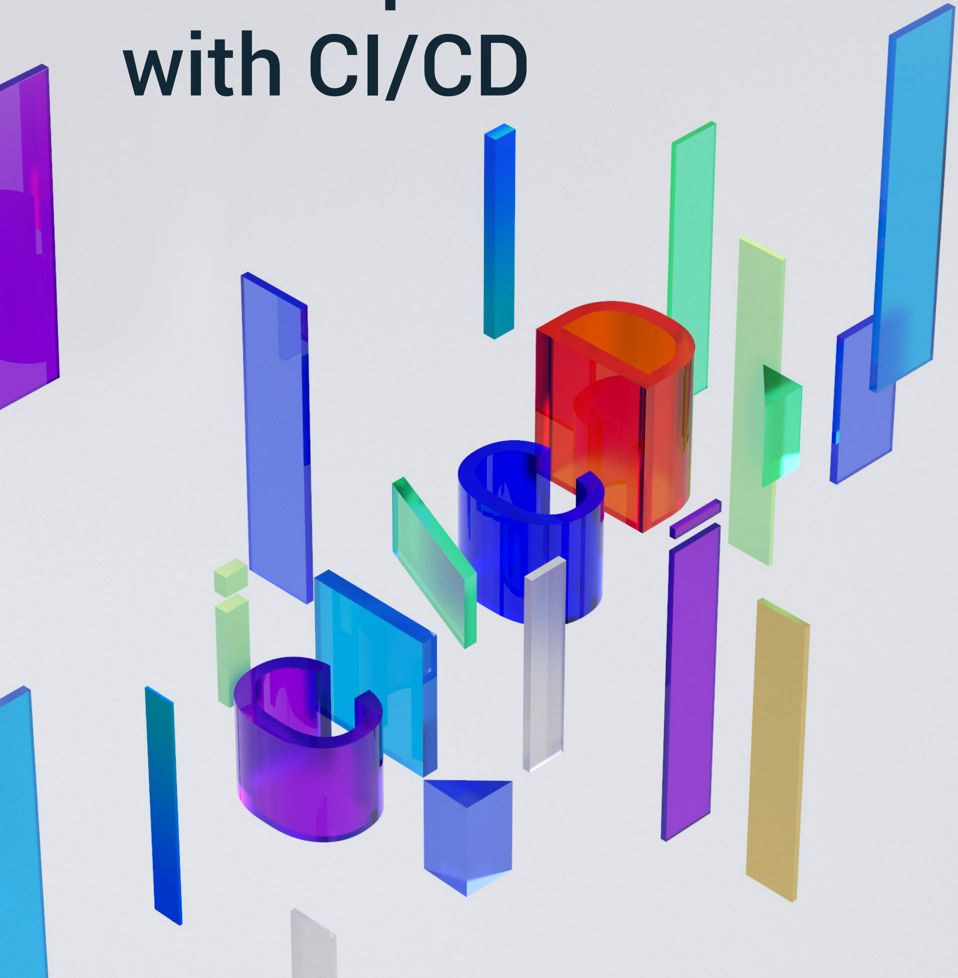
Kong

Boosting Microservice Development with CI/CD



Boosting Microservice Development with CI/CD

A microservice architecture involves building software as suites of collaborating services. Such architectures have been accepted as a better way to build many types of new applications – yet there are a number of challenges in attempting to apply monolithic deployment and delivery techniques to microservices. For most teams, continuous delivery has become an essential component of any software delivery practice.

Flexibility and shorter release cycles are two major reasons to pursue a microservices architecture. However, the lack of a continuous integration and continuous delivery (CI/CD) process will keep your team from reaching the level of agility that is necessary to support microservices development and delivery. This eBook describes a number of challenges and also provides recommendations on approaches to blending CI/CD with microservices application development. We also look at how the microservices development impacts the organization of a CI/CD pipeline.

Content

What is continuous integration, delivery and deployment?

CI/CD consists of several integral processes:

- **Continuous integration —** Developers frequently merge code changes into a main branch. All build and test activities are automated and executed daily, which ensures that the main branch code is continuously production-ready.

- **Continuous delivery —** All changes to code that are approved through the CI gates will automatically publish to a pre-production environment. Typically, explicit approval is necessary to deploy into an active production environment. However, the process of deployment is otherwise highly automatic. The goal here is to ensure that all integrated code is always ready for production deployment.

- **Continuous deployment —** All changes to code that pass integration and delivery will automatically deploy into production.

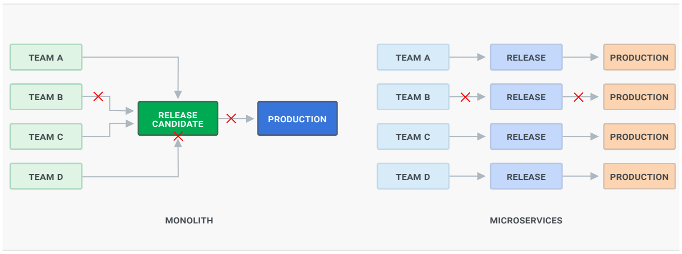
Conventional pipelines for monolithic applications

Historically, CI/CD pipelines for monolithic applications have shared the same characteristics of the application that is in development. Each project has a single, fairly complex pipeline; different projects have entirely different pipelines. Each monolithic pipeline is nearly always connected to a single Git repository.

The complexity of each pipeline makes it difficult to maintain. Moreover, each pipeline has been typically controlled by a small team of specialists who are familiar with both the deployment environment and the internals of the application. For each project, operators maintain the pipeline structure while the developers only work with the source code. This runs counter to the DevOps paradigm, in which all teams share responsibility for common infrastructure and collaborate on common problems.

The importance of a CI/CD pipeline

In a conventional monolithic development effort, there is only one pipeline—and it contains the application executable as its final output. All of the development work products converge into that pipeline. When the team finds a critical bug, the team must code, integrate, test and publish a fix that corresponds to that bug. Multiple fixes can cause delays in releasing new functionality and features. To some extent, a development team can minimize the impact of any code changes by factoring modules properly and employing feature branches. However, when the application inevitably increases in complexity, a monolith release process will very likely become more and more brittle.



To achieve a higher release velocity and minimize risk, a CI/CD pipeline must be highly automatic—and yet also highly reliable. Regressions and service disruptions must be kept to a minimum if your team wants to perform a production release once or more each day. However, your team must also construct a pipeline that is resilient enough to withstand a bad deployment update and quickly roll forward or roll back.

The primary goal of continuous delivery is to minimize time iterations for the entire code-test-deliver-and-measure cycle. Increasing deliverable throughput by optimizing this cycle is the key to productivity and higher quality code. If you're new to these concepts, this might seem counter-intuitive, but the code can be fixed and refined through this cycle. Ultimately, less time is spent on deployment—which the team can redirect to additional design and quality tasks.

The complexity of CI/CD pipelines

The size and complexity of a non-trivial CI/CD pipeline can easily become a huge pain point. Though there are a number of tools available for continuous integration of the constituent parts of a monolithic application, continuous deployment is different. CD has forced many companies to create an extensive set of custom scripts to manage deployment.

As the number of applications grows within an organization, management of all of the corresponding pipelines has become a burden. It's common to find that many of these pipelines amount to little more than a spaghetti bowl of configuration management tools, Python scripts and bash scripting.

Such messes are especially common within organizations that have not yet begun the adoption of microservices. Much of the reason that this persists is that most developers work on a single application, and therefore they don't feel pain of pipeline maintenance. This thankless job is done by operators who spend much of their time maintaining existing pipelines and creating new ones.

Not all teams are ready for microservices

Despite the benefits and popularity of microservices, some organizations aren't ready to begin taking this journey. A primary consideration is team size. If your team is small (one to three people), microservices development may add unnecessary additional overhead. Microservices are especially useful for scaling operations and managing thousands of requests. Microservices are best employed by larger teams that are split into smaller teams that are building different features and functions.

Microservices are also necessary when the architecture calls for distributing a workload across multiple clusters and data centers. If an organization doesn't have this explicit need, then it may not be vital to convert to microservices development. If your organization is moving forward with microservices development, then it is important to consider continuous integration and delivery automation to manage the complexity of this type of development.

CI/CD for microservices

These are the primary advantages of a robust CI/CD process for a microservices architecture:

- Each team can independently build, integrate and deploy its own microservices—without affecting or disrupting other teams.
- Before a new version of a microservice deploys to production, it will first deploy to the development, test and QA environments for validation. Quality gates are enforced at each level of integration.
- It's rather easy to deploy and evaluate a new version of a microservice side-by-side with one or more previous versions.
- It's easy to establish access control that is segmented and easily managed.

In a microservices development team, it should never be the case that every team has to get in line behind a lengthy release train. A team that builds one microservice can release its update at any time, without having to wait for changes in one or more other microservices to be integrated, tested and deployed.

Challenges for CI/CD with microservices development

Though it is perhaps even more beneficial for microservice development efforts, CI/CD brings with it a number of challenges:

- **Safely, rapidly and continuously releasing new features**—Managing frequent feature releases requires vigilance, especially when such features involve changes in multiple microservices.
- **Managing deployments among complex technology stacks**—You may encounter many tedious challenges when developing microservices into an environment that includes disparate technology stacks.
- **Maintaining the integrity of complex distributed systems**—The overall complexity of a system will increase if your project involves decomposing a monolithic system into smaller microservices. It's likely that your team will now need to grapple with distributed systems concerns.

Scalability issues with microservice pipelines

Microservices development and deployment can offer several advantages, and it also entails a number of scalability challenges. Quite simply, microservice pipelines and repositories become more difficult to manage as the team increases the number of applications and services that it supports.

A typical development organization may have to deal with one to five pipelines to manage a set of monolithic applications (corresponding to as many as five projects). However, the number can easily rise to 25 or more pipelines if the move is made to divide each monolith into five microservices each. Of course, these numbers vary by organization. It is common for a microservices architecture to employ 10 or more microservices. Consider a company that maintains 50 applications: its operator team will need to manage more than 500 pipelines. Such a large number means that such a company must find a way to automate the management of so many pipelines. It continues to be a source of industry frustration that several CI solutions cannot even handle a large number of pipelines.

At this point, we come to the largest pitfall in pipeline management in the era of microservices. There have been some innovations in the area of shared microservice pipeline segments. This sounds rather appealing, but let's follow the workflow. First, operators must locate the common parts of pipelines with applications. Next, a shared pipeline segment registry is built to hold all of these common parts. Then, pipelines in existing projects need to be rebuilt so that they depend on the common segments. Finally, new projects must first examine the library of common pipeline segments and choose what is already there. The result is that the unified pipeline is actually composed of steps that are common to other pipelines and also steps that are specific to that project only.

This has led to the rise of several solutions that attempt to centralize common pipeline parts and reuse them as "libraries" within the constituent software projects. The biggest challenge here is that this approach requires a huge investment of effort and also a disciplined team that can communicate and cooperate on all of the following:

- Careful detection of which pipeline segments are common
- Maintaining the library of common pipeline segments
- Prohibiting the copy-and-pasting of any pipeline
- Developing new pipelines, as necessary
- Initiation and pipeline startup for each new project

In practice, the number of microservice applications will continue to grow, and many teams will find it quite difficult to follow all of these principles when creating a succession of new microservice projects.

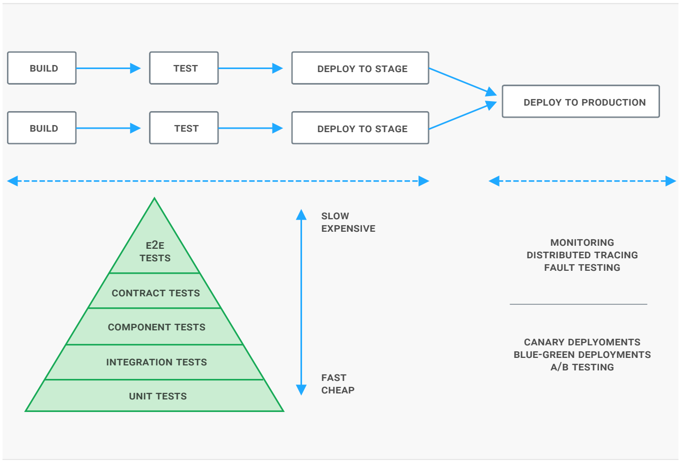
Best CI/CD practices for microservice development

Here are five solid practices that can help your team design a CI/CD workflow for managing microservices development.

1. Craft a solid test strategy

Testing and verifying systems of microservices is substantially more complex than testing a conventional monolithic application. An effective test strategy must account for isolated testing of individual microservices and also verifying behavior of the entire system.

Conventional approaches are still applicable and highly suitable for upstream testing of microservices, especially for isolated testing. The test pyramid remains useful for your team in maintaining balance among all of the



various types of tests. However, this approach has limited effectiveness when testing many services together. A major reason for this is that there are a number of errors that you won't be able to simulate. Examples of this include problems that arise from inconsistencies in a highly distributed system or hardware/network failures that cause failures in the system.

Because of these challenges, it will be necessary to supplement conventional testing with techniques such as lightweight UAT, synthetic user testing and fault-injection testing.

2. Carefully design your environments

An environment plan outlines how you intend to use each environment and the strategies to move/promote all types of artifacts through each environment. First, it's important to think about the various environments that your team will need to handle all of the use cases. Keep in mind that different groups throughout your organization will have different needs. So, your environment planning should account for all of these disparate requirements.

It's also important to consider how your team might employ cloud infrastructure so that you can dynamically create these environments. In addition, you need to craft a strategy for promoting all artifacts from one environment to the next. Since CD pipelines continually generate many artifacts, think carefully about how many artifacts you will need to manage and how many repositories will be necessary.

3. Revisit your CI practices

Continuous integration is a critical practice in any successful CD strategy. This goes beyond the basic considerations pertaining to build definitions and build servers. Trunk-based development and feature toggles are two additional vital practices that are highly effective in building out a robust CI process.

With trunk-based development, developers work together on code that lives in a one branch known as the trunk. The aim here is to avoid deviation in development branches and also avoid the merging frustrations that result. Trunk-based development also requires that you implement controls known as feature toggles.

Feature toggles enable varied, multiple commits of both work-in-progress and features that are complete.

These toggles give your team the ability to disable incomplete features temporarily after they've gone in to production. You can enable these features once they reach completion and pass all relevant tests. Typically, feature toggles are kept in a configuration or specification file near the codebase. The CD pipeline automation mechanisms will enable or disable these toggles in the relevant environments, as appropriate.

Similarly, you can employ other types of toggles such as release toggles that restrict access to incomplete code. Other types include ops toggles that restrict the function of production code, experimental toggles for multi-variable testing and permissions toggles that enable specific behavior for privileged users.

4. Strategically manage your configuration

The configuration for an application consists of everything that pertains to the deployment, so the configuration files should be kept distinct from the code. As you can imagine, configuration is different when managing deployments for one or more groups of microservices.

One useful approach is to centrally manage the deployment configuration in a repository such as Vault or Consul. Distributing deployment configurations among tools such as CD Pipeline or Chef makes it more difficult to manage. Another approach is to standardize how the configuration is distributed, irrespective of the tech stack that supports your services. With this approach, the services themselves will consume the configuration according to the stack. Your team could, for example, employ 12-factor recommendations and avoid the distribution of configuration files entirely.

Finally, your team needs to establish a process to guard secrets such as certificates to ensure they are appropriately managed. Typically, this process is manual, but it’s vital to think about it early and set it in place.

5. Prepare for failures

TIn microservices system development, updates are continuously and frequently happening to multiple services. What is the best response when a service deployment introduces a bug or some kind of instability?

In many cases, the best remediation response is to roll forward. This means determining the root cause of the failure and then promptly applying a fix. Keep in mind that one requirement for rolling forward is to ensure that your team has set up the ability to perform a hot-fix branch release directly into production. Because of timing and coordination, it may not be best to handle a fix for a production problem by means of the pipeline.

By contrast, rollbacks are usually problematic in production systems. It’s easy to roll back, in most cases, a granular change that accommodates other services well. However, if the deployment includes more elaborate changes, such as database schema changes, then it will be necessary to deploy the database changes separate from the code changes. This will need to be done in separate deployments to ensure compatibility of the database changes and the earlier code versions.

Wrapping it up

Throughout this book, we've learned that shorter and flexible release cycles are two major advantages of pursuing a microservices architecture. However, the lack of a continuous integration and delivery process will keep your team from having the agility that is necessary to support microservices development and delivery. We looked at the challenges and made recommendations on some approaches in blending CI/CD practices with microservices application development.

Research sources

https://www.gocd.org/2018/04/25/five-considerations-continuous-delivery-microservices/

https://codefresh.io/continuous-deployment/ci-cd-pipelines-microservices/

https://tech.findmypast.com/achieving-continuous-delivery-of-microservices/

https://tech.findmypast.com/achieving-continuous-delivery-of-microservices/

https://www.weave.works/blog/microservices-and-continuous-delivery

Kong

Konghq.com

Kong Inc.
contact@konghq.com

251 Post St, 2nd Floor
San Francisco, CA
94108 USA