Kong

EBOOK

Smarter, Faster, More Productive:

Creating a Developer Platform that Empowers Publishers and Consumers

Content

Abstract

Organizations embracing distributed architectures risk slowing development velocity, increasing management overhead and reducing service discoverability due to their legacy API platforms. Legacy solutions lack CI/CD integrations, cannot deploy natively on Kubernetes and don't provide visibility into services. Organizations will be able to maximize speed without sacrificing quality or control by bringing end-to-end automation, streamlined developer onboarding, and native integrations with containers and Kubernetes to their developer platforms.

Introduction

Building a compelling, attractive and highly useful developer platform has become essential for many software, computing and service companies. For our purposes, we define a developer platform that is built for distributed architectures as one that features end-to-end automation, streamlined developer onboarding, and native integrations with containers and Kubernetes. A high-performance platform maximizes speed without sacrificing quality or control.

The days are long gone in which API consumers are seen as a passive audience. Today, many are potential distribution partners and play a key role in redefining business value. Increasingly, modern business is focusing on facilitating specific roles and types of transactions, and less on owning the means of production. The leaders of future interconnectable ecosystems will run centers of excellence. They will create and maintain platforms on which customers have many incentives to innovate, curate and consume non-linear transactions in a collaborative developer ecosystem.

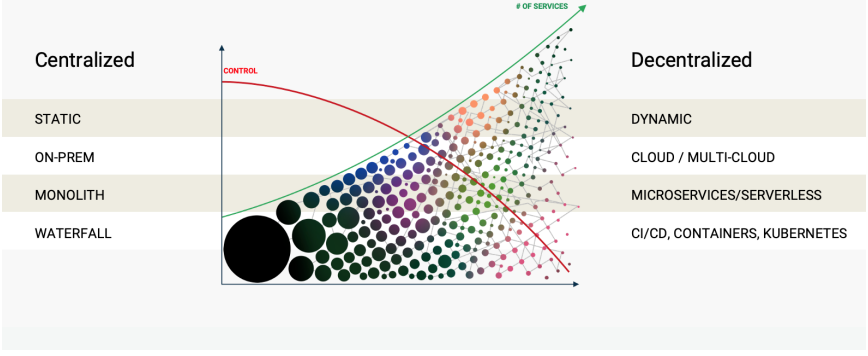
Leveraging an API strategy that moves your organization toward a developer platform may have been a decision you made long ago when first exploring an API-centric business model. To be successful in building a solid, readily useful ecosystem around one or more APIs, your company must also commit and invest resources to cultivate an engaged developer community.

Developers are very particular *business* customers. They have serious objectives, motivations, perspectives and pain points. Typically, developers are cynical in the face of conventional marketing and look for solutions that are quickly implementable. Beyond this generalization, it's important to keep in mind that many types of developers might want to use your API.

For organizations that want to succeed in building an engaging developer platform, a pressing need is to achieve a proper balance of infrastructure flexibility and performance elasticity to address developer needs, while also managing increasing geographic dispersion and unpredictable usage volumes. Such companies realize that it's necessary to move to decentralized architecture and infrastructure, which includes serious consideration of cloud native platforms, establishment of a CI/CD pipeline, containerization and microservices, to name a few. Furthermore, organizations are facing these developments with a strong sense of urgency, recognizing an imperative to adapt quickly or risk becoming irrelevant.

The World is Changing

APIs and their Connections are becoming the backbone of applications and the business



To better prepare for the future, it's important to understand this rising technology trend. In this e-book, we examine the challenges to building a developer platform for a world of distributed architectures and explore how to overcome these challenges and take advantage of the future of cloud native software.

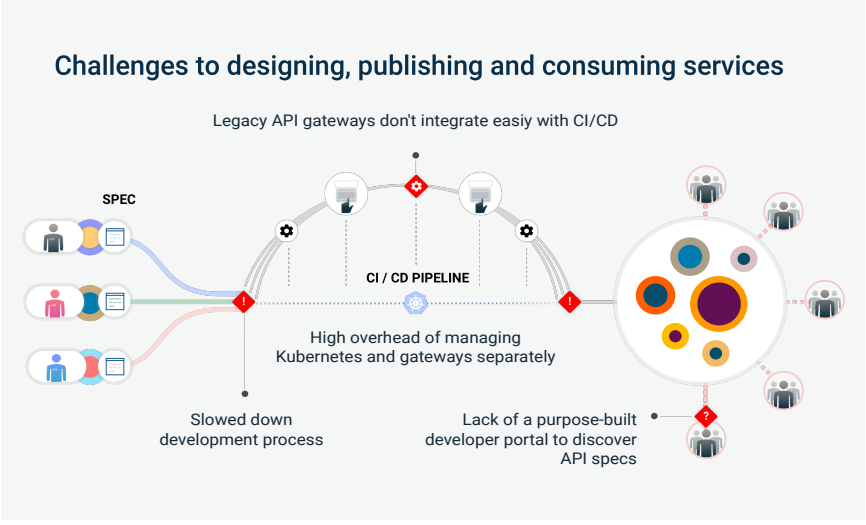
Challenges to building a developer platform

Because of the inefficiency and inflexibility of their legacy API platforms, organizations that seek to embrace distributed architectures actually risk slowing development velocity, increasing management overhead and reducing service discoverability. Such legacy systems commonly lack CI/CD integrations, cannot deploy natively on Kubernetes and don't provide visibility into services. These problems typically nullify the benefits of distributed architectures.

There are the serious challenges that face any development teams that aim to provide an accessible, navigable, programmable API platform:

- Slowed development process during API design
- Legacy API gateways cannot easily integrate with CI/CD
- Lack of a purpose-built developer portal to discover API specs
- Managing Kubernetes and gateways separately

After considering each of these challenges below, we'll have a look at how best to address them.



Slowed development process during API design

Ironically, rushing into API implementation often ends up slowing down the development process. When API designers complete an implementation and fail to adequately test it or get feedback from consumers, the result is needing to go back to the drawing board. At the core of the challenge is the specification. API designers end up needing to revise the specification once their implementation runs into challenges. Not only does this slow down the whole process—it also increases the risk of breaking changes (modifying a specification and implementation in a way that impacts the API consumers negatively).

This is why mocking is an important feature for enabling developers to test your API. Mocking provides a sandbox that mimics API interactions, without imposing the need to create an actual account or alter live-system data. The sandbox should be easy to deploy and reset whenever the developer needs to do so. Developers need to have the ability to use the sandbox very much like they would employ your production API. The primary difference here is that the sandbox endpoint should have its own URL. The goal is to have the developer test against the sandbox, gain experience and confidence in using it, and then quickly move to the production API when they are ready.

Legacy API gateways don't easily integrate with CI/CD

Legacy systems tend to be tightly coupled, static, monolithic blocks consisting of interwoven procedural code. A majority of legacy systems have been built long ago to operate in batch mode—from the core outward. Testing often requires cycling and backtracking through multiple business days, as these systems were not built for the CI/CD workflows that focus on daily, incremental, iterative deliveries to common repositories governed by testing automation. This flatly limits the level of agility that you would otherwise want to cultivate in an organization.

There are a variety of solution approaches for overcoming such architectural challenges, ranging from killing an application entirely, migrating to modern platforms or refactoring much of the legacy code. It should come as no surprise

that the best approach depends largely on your context. The promise of a microservices architecture is certainly appealing, but there are many questions to tackle prior to any transition from monolith to microservices. (Learn more in the Kong e-book, Blowing Up the Monolith: Adopting a Microservices Architecture.)

Lack of a purpose-built developer portal to discover API specs

Most portals are found to be lacking in many important respects, such as initialization, troubleshooting resources and poor support.

Familiarity and initialization is where it's won or lost for a developer portal. How long does it take a developer to start using your API? The signup process should be very straightforward, but a developer should be able to readily obtain all the necessary information. On the homepage of your API, a developer should be able to get going with the basics within five minutes:

- Understand the purpose of your API within one minute
- Identify the entry point within the next minute
- Create a new account, make a call to the system and capture a key result in less than three minutes

When it comes to the quality of the portal, developers are without question the most important focus for your API. If you built your API like a product and treat your developers as target customers, then you will surely offer a better portal

and a better experience. The API reflects the product you're selling. If you haven't gotten to the point of a solid product-market fit, then the time is now. Start to pivot and rethink your strategy.

High overhead of managing Kubernetes and gateways separately

Today, building applications using a microservices design pattern and deploying these services onto Kubernetes has become commonplace for running cloud native applications. In a microservice architecture, a single application consists of many microservices—each of which is typically built and managed by a small team.

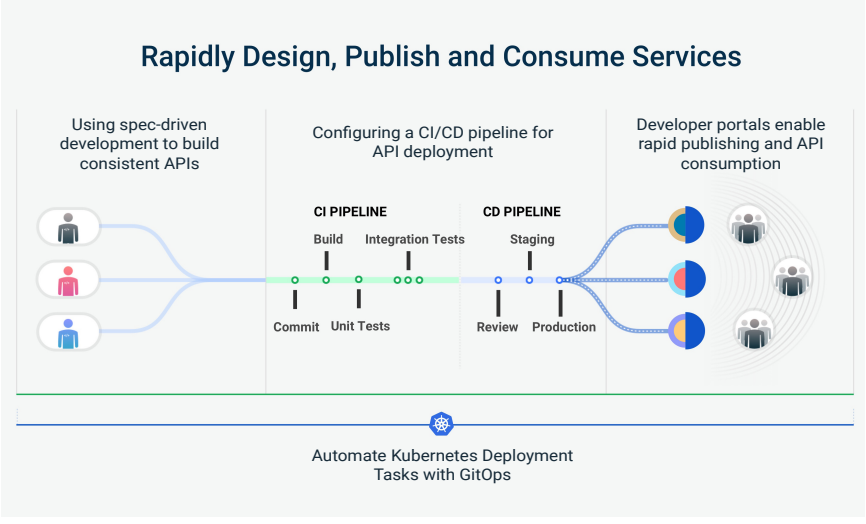
As these teams are increasingly managing their environments via Kubernetes, it is becoming painful to rely on API management solutions that lack native CRD support. This leads to two big challenges in managing a Kubernetes gateway and app configuration separately, namely (a) the risk of incurring substantial downtime and (b) configuration drift. Furthermore, the responsibility for addressing these challenges commonly extends from the edge of the system where the user requests arrive through to the service's business logic and down into the associated messaging and data store schema.

How to overcome these challenges

Overcoming these challenges involves:

- Using spec-driven development to build consistent APIs
- Configuring a CI/CD development pipeline
- Building developer portals to enable rapid publishing and consuming of APIs
- Kubernetes-native declarative configuration for your gateway

Using spec-driven development to build consistent APIs



One of the easiest ways to build consistent, readily consumable APIs is to implement spec-driven development. Essentially, this means building your API in two different stages: design and development. In the design stage, the development team crafts an API specification with extensive user input. The goal here is to create a blueprint of how the API should work. There are several templates available to help you do this, including the Open API Specification (OAS, formerly Swagger).

Spec-driven development is a process that employs an API specification as the key guide to implementation. Inputs and outputs become test cases to ensure that the API does everything it promises. There is one essential requirement to use this approach: Write the API documentation first. The more that can be done beforehand, the better. It's quite sensible to go agile and increment the documentation in cycles, similar to what is done when coding.

The OAS has strong documentation and is highly compatible with lesser-used languages. It offers quick setup and a solid support community. Most importantly, OAS is a bottom-up specification in that it specifies the behavior which affects the API to create complex, integral systems.

The second stage of spec-driven development is the actual writing of the code for all of the API layers. With a strong blueprint in hand, your developers can quickly and confidently build out the API, with little worry about how each of their resources will work with those of another developer—or how the schemas will match.

The big idea here is this: in the design phase, you eliminate nearly all of the design flaws and potential design bugs. Of course, this depends on the team conforming to the design specification all the way through development. If there is any deviation (without a carefully and sensible change to the design specification), then the result will be a 'seat-of-the-pants' API that is poorly documented and difficult to consume.

Configuring a CI/CD pipeline for API deployment

If you have APIs, it's usually beneficial to deploy your API from a CI/CD pipeline. For many API development teams, deploying an API from a CI/CD pipeline is an essential activity of comprehensive API lifecycle management.

Here are the key principles for API deployment from a CI/CD pipeline:

Use a contract-first approach

Although a code-first approach doesn't prevent you from deploying your API from a CI/CD pipeline, using a contract-first approach will reshape your processes to be much more efficient and reliable.

In this approach, the team crafts the API contract well in advance of the implementation phase. Writing this contract should be a collaboration among the architect, product owner, developers and early adopters of the product.

Ensure the testability of your API

Thorough testing is absolutely critical to configure deployment of your API from a CI/CD pipeline. At a minimum, you need to accommodate these types of tests into your pipeline:

- Unit tests — individual testing of each software component
- Integration tests — test modules and collections of software components
- Acceptance tests — these ensure that business expectations are met
- End-to-end tests — verify that all software components function collaboratively and comprehensively—in an environment that is very close to production conditions
- Performance tests — verify that performance remains acceptable and doesn't degrade by the introduction of any fix or new feature

Conform to semantic versioning

For any new version of your API, it's critical to adhere to the semantic versioning. This improves the ability of your CI/CD pipeline to handle the new release. A new minor version will be backward-compatible, and it is deployable in-place. Any major version will have to be deployed side-by-side to provide an option for existing customers.

Developer portals enable rapid publishing and API consumption

In the early stages of an API program, a developer portal may simply be a documentation repository, but its role should expand as the program grows and extends to external API consumers.

At minimum, a developer portal should include:

- Simply service discovery via search
- Engaging documentation with interactive consoles and sandbox environments
- Code snippets as well ascode for a sample application—in a number of languages that match well with your target
- Terms of Service and Service Level Agreement
- Self-service API registration
- A public-facing API specification format file that uses the OAS or other human- and machine-readable metadata about your API
- An interactive API console that provides sample responses from each of the resources in the API
- A developer playground, preferably driven by an API mocking service, in which developers can sample with their own code
- Enterprise support

GitOps (automating Kubernetes deployment tasks)

GitOps leverages a version control system such as Git to contain all documentation, information and code for a Kubernetes deployment. This sets the stage for deployment automation and programmable infrastructure. The benefits of employing GitOps to automate Kubernetes deployment tasks encompass and enable all stages of the development lifecycle—including designing, publishing and consuming APIs.

The key benefits for GitOps include:

- Productivity increases through continuous deployment automation
- Significant enhancements to the developer experience by enabling the pushing of code—not containers
- Improved stability through automatic audit logging of all cluster changes
- Higher reliability from the Git revert-rollback and forking capabilities—which are built-in and result in a single source of truth
- Standardization and consistency end-to-end workflows

If you're convinced of these GitOps benefits, the next step is understanding Kubernetes, Docker and Git—the tools necessary to actually implement GitOps. To paraphrase The New Stack: Kubernetes is probably the most influential system in the world of information technology to arise in recent years. Quite simply, you can't implement GitOps without a running Kubernetes cluster.

By definition, Kubernetes is an open source system for automating deployment, scaling and management of container applications. Everything in GitOps begins and ends with Kubernetes.

Kubernetes can't function without Docker. Indeed, you must first install Docker before you can install Kubernetes on your server. Docker is a set of platform-as-a-service products for managing and delivering software in packages known as containers. A container is a package of software that bundles code and all its dependencies so the application runs quickly and reliably in multiple computing environments. A Docker container image is a standalone, lightweight, executable package of software that includes everything necessary to run an application: code, runtime, system tools, system libraries and settings.

Finally, Git is essential for GitOps. It's the central hub for GitOps, since it stores all of your documentation, configurations and code necessary to deploy and maintain your cluster. However, don't think of Git only as a repository for use by humans to make the Kubernetes cluster function. There are also automators that cooperate with Git to make everything seamless and automatic. Configurations are read from Git, and if any changes are found,the cluster automatically updates in accordance with the changes.

Conclusion

We've identified how distributed architectures change the requirements for developer platforms and presented key strategies to overcoming these challenges. While tactically the next step is considering which aspects of the vision shared are most relevant for your team and organization, the broader takeaway is understanding why taking a legacy approach does not work in a world of distributed architectures. Your team wants to take its service to market faster and reduce the total cost of ownership for doing so. With teams looking for every opportunity to do more with less, organizations who have created a developer platform for the distributed world are dramatically reducing TCO and increasing productivity. For example, Cargill achieved 65x faster deployments by integrating Kong with CI/CD tools and using Kong with Kubernetes to auto-scale based on demand. Reach out to Kong, and we will be happy to share how we have seen hundreds of our enterprise customers reduce costs by automating the design, publishing and consumption of APIs and services.

References

https://metova.com/yes-you-have-apis-but-they-suck/
https://www.codeproject.com/Articles/837868/Building-Your-API-using-Spec-Driven-Development
https://www.programmableweb.com/news/abcs-building-api-developer-portals/analysis/2017/09/27
https://nordicapis.com/5-reasons-why-developers-are-not-using-your-api/
https://nordicapis.com/how-to-design-frictionless-apis/
https://www.pingidentity.com/en/company/blog/posts/2014/why-developers-hate-your-api.html
https://developers.redhat.com/blog/2019/07/26/5-principles-for-deploying-your-api-from-a-ci-cd-pipeline/
https://devops.com/devops-legacy-systems-mission-impossible/
https://yos.io/2018/02/14/api-developer-portal-best-practices/
https://thenewstack.io/the-tools-you-need-to-run-gitops-based-automated-deployments/

Konghq.com

Kong Inc.
contact@konghq.com

150 Spear Street, Suite 1600
San Francisco, CA 94105
USA