Kong

EBOOK

Full Lifecycle Service Management

Addressing the Challenges of Modern Service Environments

In a world with multiple, emerging options for interfacing between microservices, a fixed, myopic viewpoint will fail a growing business. Modern teams succeed through the diversity of the tools at their disposal for service management. Moving beyond traditional API management requires cultivating a vision that prepares you for the future with flexible, agnostic and adaptive strategies. In this eBook, we introduce a framework for Full Lifecycle Service Management and explain how this approach will address the challenges of modern service environments.

Content

We live in an exciting time for software; we are witnessing a monumental shift in how applications are built. We have the opportunity to participate in the large-scale movement from centralized applications to decentralized, highly performant software architectures.

API and REST standards revolutionized the exchange of data and connection between systems and applications across the internet. The advent of containers led to widespread adoption of microservices on an unprecedented scale. Now as we move into a cloud native, microservices-first world, we can no longer rely on RESTful APIs alone to meet the needs of the modern organization.

Instead, the move from monolith applications to microservices requires a shift from API management to service management. A service is any means for interfacing data exchange between applications. Modern teams succeed through the diversity of the tools at their disposal for service management. They work in different languages with diverse deployment strategies in multiple clouds. They use the best tools at their disposal to not only create the rules that drive their businesses but also to intelligently derive the rules using machine learning. They cannot be limited by protocols, environments or cloud providers.

In a world with multiple, emerging options for interfacing between microservices – from GraphQL to Kafka to gRPC – a fixed, myopic viewpoint will fail a growing business. Moving beyond traditional API management requires cultivating a vision of flexible, agnostic and adaptive strategies that prepares you for the future. In this eBook, we introduce a framework for Full Lifecycle Service Management and explain how this approach will address the challenges of modern service environments.

Understanding the Modern Era

We live in a world where services are launched and updated faster than ever before – and on a much greater scale. In a world where Netflix and Amazon handle trillions of API calls every year, spinning up capacity in a dynamic fashion while orchestrating across a vast ecosystem of self-healing containers all designed to ensure performance at scale, there is truly no way to keep up without purpose-built tools.

As technology leaders advance what is possible, consumer expectations rise accordingly. From a consumer perspective, we live in an age of 24/7 availability, where services can be accessed anywhere, anytime regardless of device or platform in use. Consumers vote with the businesses they continue to patronize, just as developers vote with the platforms and protocols they support via contributions to the ecosystem.

It is truly an exciting time to be a developer with entirely new platforms for development emerging every day. Now that mobile adoption has swept the world, the emergence of the IoT category is already leading another wave of service explosion. Rather than simply preparing for this new platform, the true imperative is to observe that regardless of the emerging platforms today and even over the next 1-3 years, the world will continue to see the emergence of new platforms, as well as new protocols and new means for interfacing between applications.

Just in the last few years, we have seen compelling alternatives emerge to RESTful APIs, each

focused on a different use case for the technology infrastructure community. A single GraphQL query can get all the data your application needs, potentially calling thousands of RESTful APIs in the process. gRPC offers performance and ease of development benefits, thanks to being based on cross-platform, language-agnostic protocol buffers. Kafka supports asynchronous microservice-to-microservice communication by acting as an event collector, allowing developers to replay event series or reproduce the state data of a given timestamp.

All this is occurring despite the state of the legacy architecture in the majority of enterprises, most of which was created in a pre-iPhone world – before the advent of containers, explosion of microservices and need to coordinate complex, networked applications on a massive scale. No one would ever believe that the scale of innovation we are witnessing in the broader ecosystem would be so terribly disconnected from the actual state of affairs in most enterprises, which is weighed down by systems coming from a SOAP and ESB era.

Delving into the Data Problem

Every team has its preferred languages for development and connecting services. However, when these differ from team to team within an enterprise, how can a business deliver the speed that consumers expect without seamless communication between platforms? It could put hundreds of hours of in-house or outsourced engineering toward connecting disparate systems, but the opportunity cost of that decision is all of the mission-critical

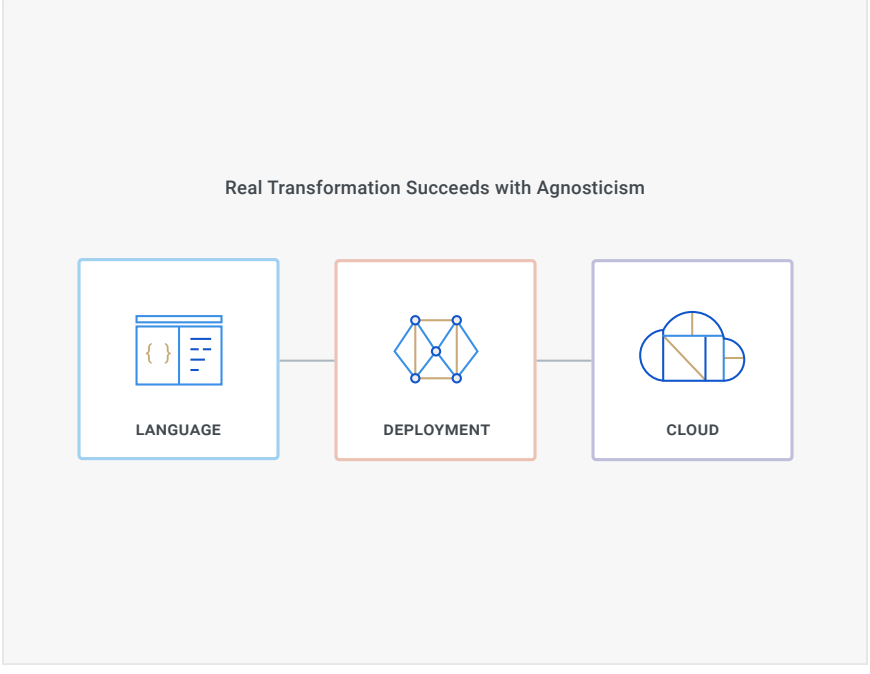
applications that could have been developed with that same effort. With the exchange of information between the microservices in an enterprise, the diversity of languages and protocols makes data availability a mission-critical problem.

The emergence of new channels and velocity with which new services are being launched threatens to leave behind businesses that are not able to adapt. Still telling is the classic example of Blockbuster failing to adapt its business model for movie rentals in light of the disruptive innovation introduced by Netflix' streaming rentals in 2007[1]. Even though Blockbuster devoted two years to building a digital strategy and launching an on-demand, API-powered offering in 2010[2], it was too little too late, and the company declared bankruptcy only a few months later. Conversely, those who are leading the charge are seeing a reward for their efforts. Stripe revolutionized online payments by taking a developer-friendly, API-first approach in contrast to the incumbent PayPal, leading to adoption by more than 1 million businesses worldwide[3].

While some argue toward expanding the definition of an API to escape the concept being conflated with REST, we should take a bolder stance: let's abandon the concept of API management entirely since it no longer serves a world of such diverse service integration scenarios. Instead, we should consider APIs as simply one class of services and move our focus up a level to the broader class of service management, where a "service" is defined as a means for interfacing data exchange between applications.

[1] https://www.thrillist.com/entertainment/nation/netflix-history-streaming-in-2007
[2] https://www.theatlantic.com/technology/archive/2010/05/blockbuster-gone-digital/56276/
[3] https://www.wired.co.uk/article/stripe-payments-apple-amazon-facebook

This new focus on “service management” reflects the underlying complexity of environments today. The way forward is taking an agnostic approach – designing services that work across any language. However, it is not enough to stop there. Agnosticism should also extend to deployment type – whether monolithic, microservices or even bare metal – and cloud provider. The point is to abstract away from any specific technology or vendor so your team can focus on designing the optimal service for your business and customers.

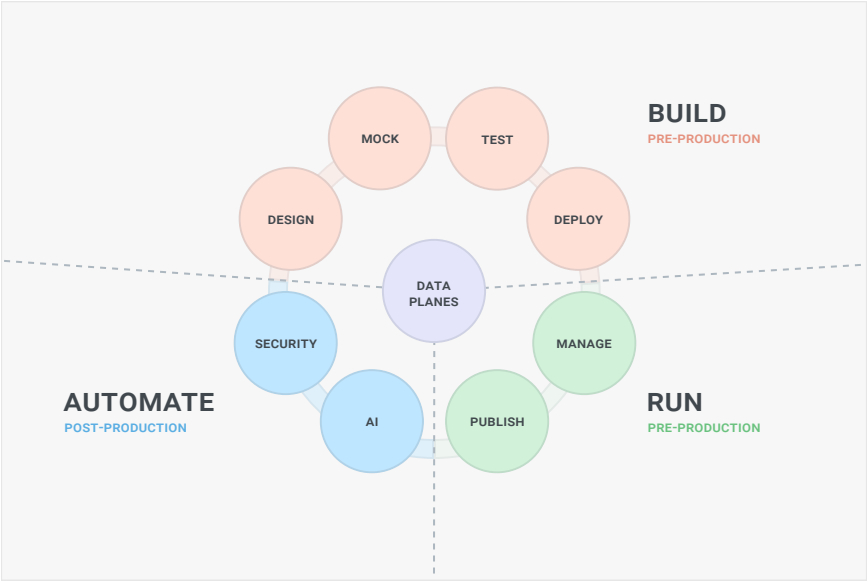


Envisioning the Next Era of Software

We've taken a step back and re-evaluated where we are focused. Rather than focusing on what we build and making sure that all our services are RESTful, we've shifted to thinking about how we build our services. Taking a note from the DevOps principle of unifying software development and operations, when it comes to brokering information between mission critical systems, it is time for us to bring this full lifecycle approach to this domain as well.

It's not about whether an API is RESTful, gRPC or GraphQL. What ultimately matters is that each service is a contract between the producer/provider and the consumer. The consumer of a service does not care about whether other parts of your organization are adopting gRPC. The consumer cannot tolerate breaking changes because they lead to large scale failures of business-critical applications they have built.

In the next era of software, we prevent this problem by keeping the full lifecycle of a service in mind through pre-production, production and post-production. We have a powerful tool to do this with the advent of spec-driven development, which dissolves the boundary between a specification and the implementation of that specification.



The diagram above offers a framework for Full Service Lifecycle Management. Let's examine this vision in three main phases – build, run and automate.

Build

> **The Build phase represents pre-production activities for a service.**

The first step of this phase is Design. In contrast to past approaches where a specification was created after implementation and then became out of date when changes were made to an implementation later, design in a full service lifecycle management context takes a spec-driven approach. Spec-driven development starts with the spec and dissolves any disparity between the spec and the implementation.

The advantage of this approach is a much faster cycle to mock and test a service, as well as a reduced risk of breaking changes in the future. Beyond the obvious operational benefits of this approach, spec-driven development underscores taking a design and development approach that assumes responsibility for the successful operation of services as well, since any operational challenges in production or post-production will need to flow back into the specification before the implementation can be patched.

The second step is Mock. Mocking speeds up development by providing developers with a reliable imitation of how the service would work in real life. They can build other tools and interfaces based on assumptions about the behavior and structure of the service's responses.

The third step is Test. Testing provides greater confidence and earlier warnings to development teams before going into production. By using isolated, discrete services instead of huge monoliths, it's easier to test and find problems at a granular level and to surgically fix them one by one.

The final step is Deploy. After code has passed the relevant tests and reviews, this step enables release velocity by automating the decision to ship the code. Rather than taking a manual process with a fixed cadence, this step supports CI/CD and means teams can release new application code to production in minutes rather than only on predetermined schedules. Release quality also improves thanks to review process, tests and validation of traffic defined in the specification.

Run

> **The Run phase represents all production activities for a service.**

In the Manage step of this phase, an administrator has visibility over the entire span of services in production and can monitor their health in real time, as well as make changes as needed. With logic built into the endpoints of the system, workflows can be managed and automated between other key systems.

In the Publish step, the platform is live across all relevant systems, whether on premise or cloud-based, monolithic or serverless. The system is able to perform even above 10,000 transactions per second and to self-heal dynamically in case of unexpected outages of individual microservices. Further, the system is open to being extensible based on the needs of various business lines for customization due to specialized needs and use cases.

Automate

> **The Automate phase refers to all post-production activities. In the post-production world, we shift our focus to ensuring that the live system is optimized and compliant, leveraging real-time data and machine intelligence where possible to make our job easier.**

The Artificial Intelligence (AI) step of this phase entails using machine learning to reduce manual

tasks related to documentation and alerting teams to critical service information. Pushing documentation directly into an accessible, search-friendly developer portal and automating updates reduces manual effort significantly. Analyzing traffic patterns and detecting anomalies for business attention enables real-time threat resolution. Artificial intelligence can also be used to visualize information across all services, which is increasingly important as complexity and number of services increase.

The Security step of this phase involves creating policies to help ensure that internally the system is compliant with industry requirements. Role-Based Access Controls (RBACs) ensure the data being accessed aligns with the authorization levels at the organization.

Adopting a Full Lifecycle Mindset

Now that we have explained each of the phases of the Full Service Lifecycle Management framework, let's highlight how these elements combine to prepare an organization for the next era of software, and each depend upon each other for success of the whole. Remembering that these steps span the phases of pre-production, production and post-production helps paint this picture, and recognizing linkage across non-sequential steps underscores the importance of a "full-lifecycle" mindset.

In pre-production, our key challenge is to design and build a service network that meets the performance

requirements of the business and customer, offers maintainability and will not become irrelevant as new services and requirements emerge. Thus, the most important link is between the “Build” and “Automate” phases – public specification documentation must be the starting and ending point for creating and updating services relied upon by hundreds or thousands of consumers. The developers creating specs will still bear the responsibility to design with a minimization of breaking changes in mind, so this step also bears an important link to the Management step of the cycle.

In the Run phase, the benefit of abstraction in the Build phase comes to life. Instead of needing to worry about a giant, tangled system, the fruits of a successful Build phase are an orderly, transparent and well-encapsulated set of services. This abstraction is important, both for the later Automate phase and for day-to-day management. In order to ensure resilience and operational excellence, the Run phase must be front of mind even before and after. The Build phase is meant to make the Run phase more manageable – all so that the Automate phase can glean valuable information from it.

Running in production gives a wealth of information about usage, health, behavior, and access. Through Automation, systems can glean insights from that information and act on your behalf. Having taken a full-lifecycle approach means that you are set up to reduce many manual tasks since your production environment can be readily visualized and analyzed with machine intelligence. You also have the flexibility to adjust the policies that apply to your environment in the Automate phase, with these changes going into effect and directly impacting what you have in production.

Taken together, the approach laid out is the synthesis of lessons learned from organizations already in the trenches operating modern services.

Case Studies

Let's examine a few case studies to develop some feel for how this model could actually exist in the real world.

Medallia is a computer software company that empowers customer and employee engagement with real-time data, insights and tools captured from a variety of channels. As part of its growth efforts, Medallia understood the importance of leveraging APIs and microservices in order to increase speed and efficiency across its platforms. The organization needed increased governance over its APIs in order to eliminate redundancies in its code while maximizing efficiency. Tasked with designing the company's API structure, the engineers at Medallia were looking for a lightweight, performant solution that could handle all of their needs in the present and future.

To address the issue, Medallia decoupled services by separating business logic from the message layer and employing a lightweight structure to maximize scalability and resource efficiency. Medallia also implemented standardization across all API gateways. As a result, Medallia saw full application deployment times reduced by over 80 percent, increased DevOps efficiency and productivity, and was able to handle over 10,000 TPS with no performance issues.

Headquartered in Minneapolis, Minnesota, **Cargill** combines extensive supply chain experience with new technologies and insights to serve as a trusted partner for food, agriculture, financial and industrial customers across more than 125 countries. One of

the largest private companies operating today, Cargill employs over 150 thousand people worldwide and handles logistics for over 33 percent of the world's food supply.

However, Cargill's legacy IT systems were slowing its ability to create new digital products and services to address the evolving needs of its customers and partners. In response to these new digital imperatives, Cargill began the process of transforming its internal suite of APIs to be more dynamic, thereby enabling consumption across the entire company.

Cargill unified developer experience across legacy and cloud native systems, defining how services behave and then automatically scaling those services up or down depending on the situation. As a result, more than 400 new digital services were created with up to 65 times faster deployment times due to automated validations.

SoulCycle offers cycling-based fitness classes across the U.S., Canada and Europe. SoulCycle is a growing fitness market leader with more than 50,000 riders a week attending classes. It is committed to nurturing the health and happiness for the community.

With aggressive growth targets, SoulCycle needed to leverage technology to better integrate with partners and enable expansion into new business models. Challenging this vision, SoulCycle's monolithic architecture presented a critical bottleneck to service development and scalability. SoulCycle began transitioning several of its applications to a service-oriented architecture, leveraging containerization and Kubernetes to accelerate their development process.

Soulcycle used a single API platform to unify Kubernetes microservices and monoliths across its multi-cloud environment, with horizontal and vertical scaling enabled and a unified entrance point for all services. As a result, the company saw more than triple the improvement in developer efficiency, as measured by average releases per week before and after the solution went live.

These examples showcase a variety of problems across the range of Full Service Lifecycle Management and the tangible benefits seen from moving further toward a more mature service management capability in the enterprise.

Embracing Full Service Lifecycle Management

We've identified how the world is changing, why data is the key problem to solve and laid out a solution framework.What's the next step? When thinking about getting started with moving towards this vision, it's important to focus on taking manageable, incremental steps that minimize the risk of failure while also maximizing the basis for a strong foundation to grow toward eventual maturity.

The cycle begins with Design because spec-driven development is the best place to start. You can pick a few services to begin introducing spec-first development. Obvious candidates are any that are already leveraging OpenAPI (formerly Swagger) as a framework. The other benefit of starting with this step is it represents mostly process rather than organizational changes.

You can introduce spec-driven development without moving developers between departments – this step is mostly about expanding the purview of the developer at the point of design.

The next step is taking an inventory of the existing range of interfaces, languages and protocols in your current architecture and medium-term roadmap for connecting the services across your enterprise. Whether you are coming from a monolithic or serverless world, there is a set of services relevant for your immediate department as well as the broader ecosystem within the enterprise and the broader industry. The ideal inventory reflects mission-critical services, as well as what needs to be added and expanded for ideal operation and optimization of the enterprise at scale.

Finally, the Full Service Lifecycle Management framework presented in this piece is not meant as a rigid rulebook for developing your technology roadmap. Rather, consider this framework a resource for holistically considering how services interact within your enterprise and proactively identifying service management gaps that are critical to address. Do you have visibility into services in production across the enterprise with the ability to dynamically address issues as they occur? How well does your development team document updates to APIs and other services? Where are you using services beyond RESTful APIs, and what is your plan for interoperability and unified communication between those services and RESTful services?

Lastly, agnosticism is the best guiding principle for the challenges of modern services. Being able to bring your own technology – any language, deployment or cloud – to the table is key for success. After taking these steps, you will be able to carve out a portion of services to reimagine with the full service management lifecycle in mind.

References

1. http://itrevolution.com/the-three-ways-principles-underpinning-devops/
2. https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture
3. https://thenewstack.io/netflix-devops-scale/
4. https://apievangelist.com/2018/02/03/api-is-not-just-rest/
5. https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215
6. https://martinfowler.com/articles/consumerDrivenContracts.html
7. https://www.atlassian.com/blog/technology/spec-first-api-development
8. https://martinfowler.com/articles/microservices.html#SmartEndpointsAndDumbPipes
9. https://thenewstack.io/kong-at-1-0-a-service-control-platform/
10. Kong Enterprise Datasheet: https://2tjosk2rxzc21medji3nfn1g-wpengine.netdna-ssl.com/wp-content/uploads/2018/10/Kong-Datasheet-Next-generaton-API-Platform-2-12-19.pdf
11. https://buttercms.com/books/microservices-for-startups/breaking-up-a-monolith
12. https://levelup.gitconnected.com/grpc-in-microservices-5887caef195

Konghq.com

Kong Inc.
contact@konghq.com

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