NGINX
Part of F5

EBOOK

Taking Kubernetes from Test to Production

Improving Resilience, Visibility, and Security with Traffic Management Tools

By Jenn Gile
Manager of Product Marketing
NGINX Microservices and Kubernetes Solutions, F5, Inc.

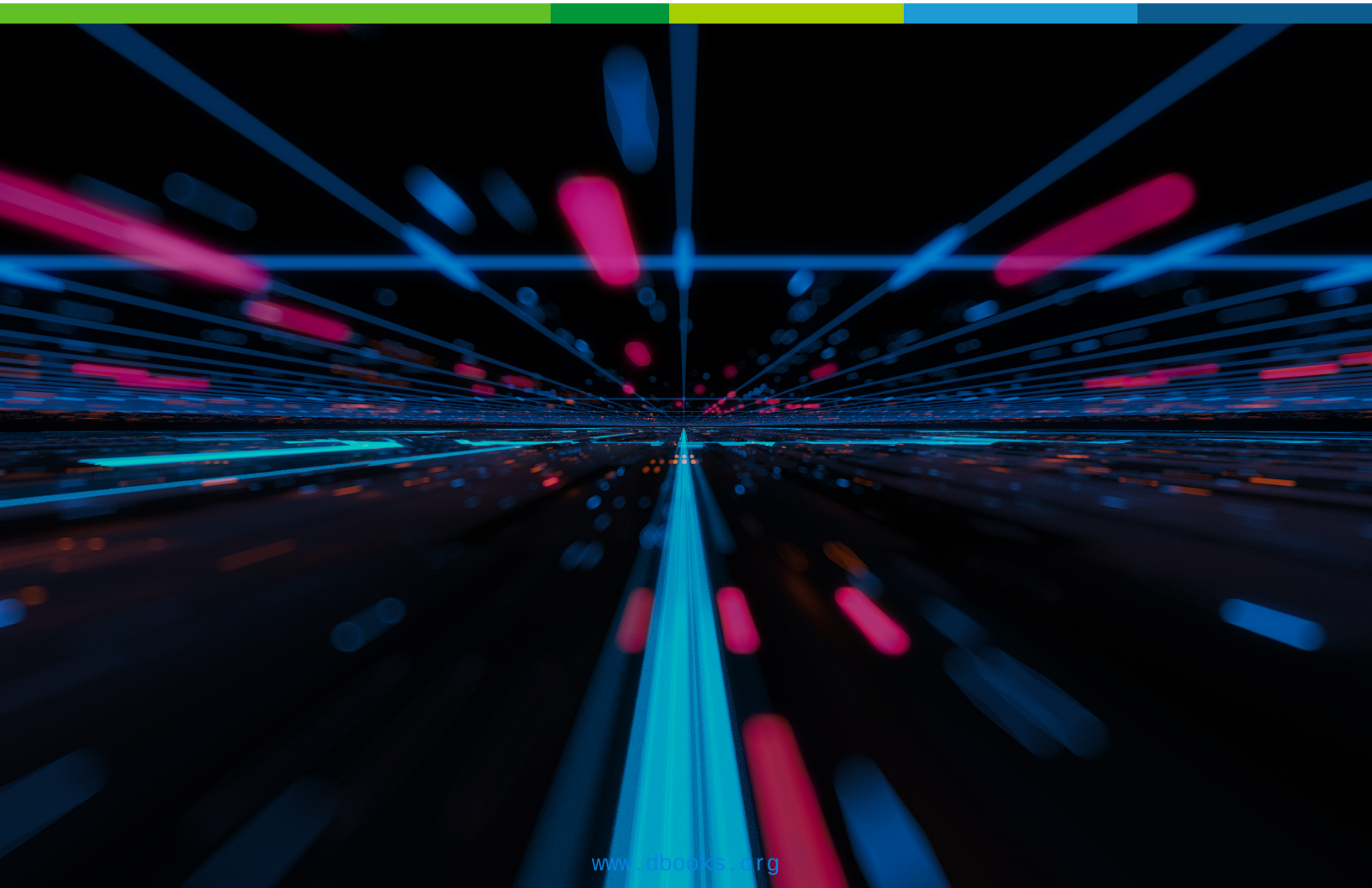


Table of Contents

Foreword 4

Part One:
Overcome Kubernetes Challenges with Kubernetes Traffic Management Tools 5

1. Reduce Complexity with Production-Grade Kubernetes 5

Modern Shifts and Trends 5

The Solution: Production-Grade Kubernetes 8

Kubernetes Priorities: Resilience, Visibility, and Security 11

2. Six Ways to Improve Resilience 12

Advanced Traffic Management Use Cases 13

1. Protect Services from Getting Too Many Requests 15

2. Avoid Cascading Failures 15

3. Test a New Version in Production 16

4. Ensure a New Version is Stable 17

5. Find Out if Customers Like a New Version Better than the Current Version 18

6. Move Users to a New Version Without Downtime 19

Using NGINX for Advanced Kubernetes Traffic Management 20

More Sophisticated Traffic Control and Splitting with Advanced Customizations 22

3. Two Problems You Can Solve with Better Visibility 23

How to Attain Insights Through Visibility 23

Troubleshooting Two Kubernetes Problems 24

Kubernetes Problem #1: Slow Apps 25

Kubernetes Problem #2: Resource Exhaustion 26

4. Six Methods to Enhance Security 28

Security and Identity Terminology 28

Make Security Everyone's Responsibility 30

1. Resolve CVEs Quickly to Avoid Cyberattacks 30

2. Stop OWASP Top 10 and DoS Attacks 32

3. Offload Authentication and Authorization from the Apps 35

4. Set Up Self-Service with Guardrails 36

5. Implement End-To-End Encryption 37

6. Ensure Clients Are Using a Strong Cipher with a Trusted Implementation 38

Part Two:
Choose the Best Kubernetes Traffic Managment Tools for Your Needs 41

5. How to Choose an Ingress Controller – Identify Your Requirements 41
What's an Ingress Controller? 42
Why Do You Need an Ingress Controller? 42
What Do Ingress Controllers Do? 42
What Problems Do You Want the Ingress Controller to Solve? 44
How Are You Going to Resource the Ingress Controller? 45

6. How to Choose an Ingress Controller – Risks and Future-Proofing 48
Ingress Controller Risks 48
Future-Proof Your Ingress Controller 51

7. How to Choose an Ingress Controller – Open Source vs. Default vs. Commercial 54
Open Source Ingress Controllers 54
Default Ingress Controllers 56
Commercial Ingress Controllers 57

8. How to Choose an Ingress Controller – NGINX Ingress Controller Options 59
NGINX vs. Kubernetes Community Ingress Controller 59
NGINX Open Source vs. NGINX Plus – Why Upgrade to Our Commercial Edition? 61

9. How to Choose a Service Mesh 65
Are You Ready for a Service Mesh? 67
How to Choose the Service Mesh That's Right for Your Apps 69
F5 NGINX Service Mesh 71
Not Ready for a Service Mesh? 75

Appendix:
Performance Testing Three Different NGINX Ingress Controller Options 76
Latency Results for the Static Deployment 77
Latency Results for the Dynamic Deployment 78
Timeout and Error Results for the Dynamic Deployment 79
Conclusion 80

Glossary 81

Foreword

Since launching in 2014, the open source project Kubernetes has taken the cloud infrastructure world by storm, rapidly becoming one of the most popular and widely deployed infrastructure management solutions in history. Kubernetes spawned an entire industry of tools and platforms designed to make deployment easier, more turnkey, and more approachable – from popular managed platforms like Red Hat OpenShift and SUSE Rancher to cloud provider platforms like Amazon Elastic Kubernetes Service, Azure Kubernetes Service, and Google Kubernetes Service. But even with these enterprise-grade platforms smoothing the way, getting value out of Kubernetes requires a steep learning curve.

With Kubernetes came many new concepts, particularly around networking and traffic management. Alongside these new concepts were entirely new classes of tools, designed for ephemeral, containerized, and distributed application deployments. In particular, Ingress controllers and service meshes did not exist prior to the Kubernetes era. Nor were Layer 4 and Layer 7 protocols and traffic typically managed from the same control plane. At a granular level, Kubernetes introduces new complexities around security and management. Simple tasks like load balancing are very different in a realm where infrastructure is 100% ephemeral and often moving constantly – both in terms of setting up new instances with fresh IP addresses and geographically moving around the globe.

These new traffic management concepts and tools have the power to vastly improve developer experience and accelerate app development and delivery cycles through greater resilience, higher performance, and better security. But when traffic management is overlooked or under-valued, organizations experience problems that make it challenging to attain that value and can even put the organization at risk. To achieve the Kubernetes dream, teams must understand and manage the traffic in a Kubernetes-native way.

This eBook prepares readers for a journey – from test to production – in the land of Kubernetes. Each chapter offers clear and lucid explanations that can inform a Kubernetes strategy and serve as a reference point for managing the flow of traffic proactively and effectively – from POC to canary, and from blue-green to full production. Insight on useful decision frameworks also help in deciding which traffic management solutions work best (or don't) for specific applications and situations. Complete with reference architectures and instantly usable code examples, this eBook can help readers at any Kubernetes skill level expand their knowledge of Kubernetes traffic management tools and concepts.

With its wealth of quick and clear foundational knowledge, this eBook deserves a place on your cloud infrastructure bookshelf and should become a trusted companion in your quest to master Kubernetes.

Jenn Gile
Manager of Product Marketing
NGINX Microservices and Kubernetes Solutions, F5, Inc.

PART ONE

Overcome Kubernetes Challenges with Kubernetes Traffic Management Tools

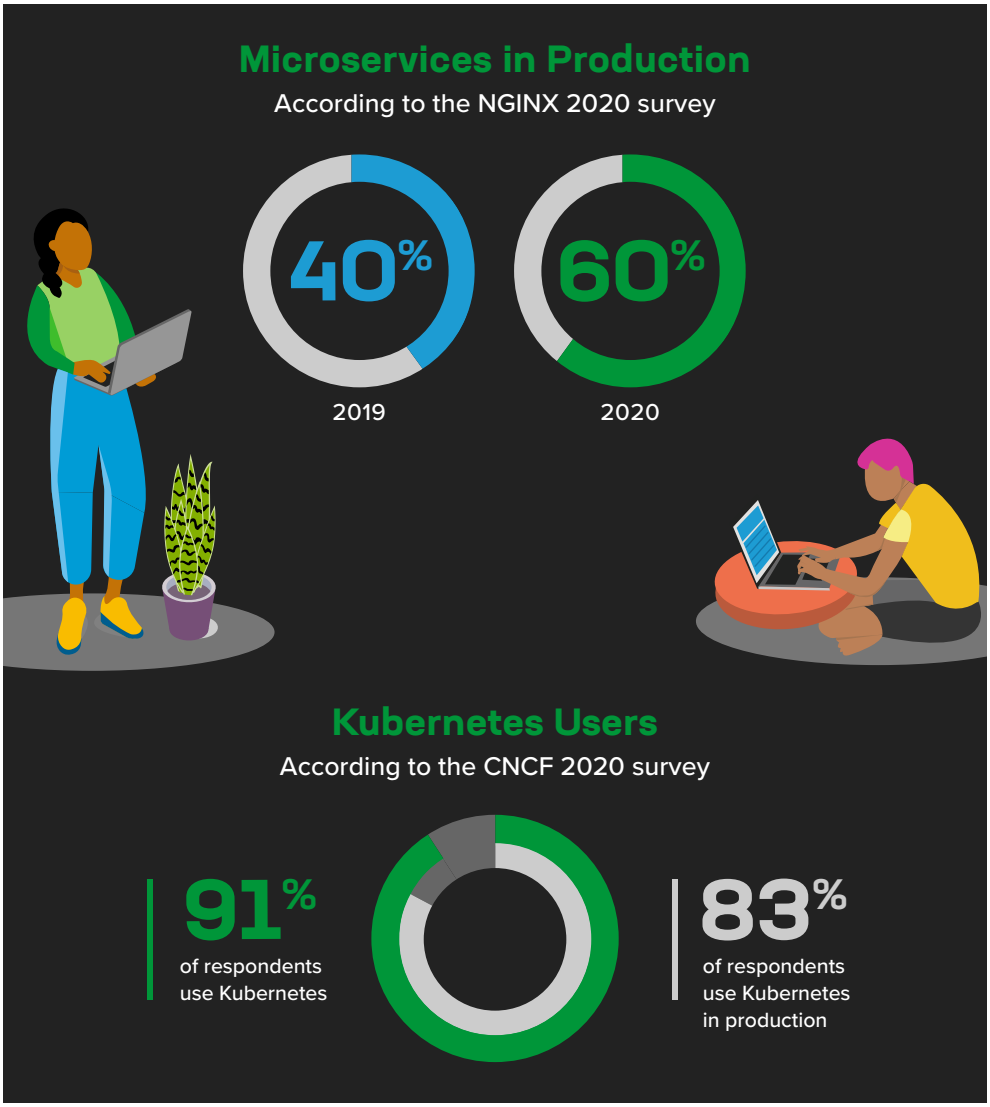
1. Reduce Complexity with Production-Grade Kubernetes

MODERN SHIFTS AND TRENDS

2020 was a year that few of us will ever forget. The abrupt shuttering of schools, businesses, and public services left us suddenly isolated from our communities and thrown into uncertainty about our safety and financial stability. Now imagine for a moment that this had happened in 2000, or even 2010. What would be different? *Technology*. Without the high-quality digital services that we take for granted – healthcare, streaming video, remote collaboration tools – a pandemic would be a very different experience. What made the technology of 2020 so different from past decades? Containers and microservices.

Microservices architectures – which generally make use of containers and Kubernetes – fuel business growth and innovation by reducing time to market for digital experiences. Whether deployed alongside traditional architectures or on their own, these modern app technologies enable superior scalability and flexibility, faster deployments, and even cost savings.

Prior to 2020, we found that most of our customers had already started adopting microservices as part of their digital transformation strategy, but the pandemic truly accelerated app modernization. Our 2020 survey of NGINX users found that 60% of respondents are using microservices in production, up from 40% in 2019, and containers are more than twice as popular as other modern app technologies.



Kubernetes is the de facto standard for managing containerized apps, as evidenced by the Cloud Native Computing Foundation (CNCF)'s 2020 survey, which found that 91% of respondents are using Kubernetes – 83% of them in production. When adopting Kubernetes, many organizations are prepared for substantial architectural changes but are surprised by the organizational impacts of running modern app technologies at scale.

If you're running Kubernetes, you've likely encountered all three of these business-critical barriers:

- **Culture**

  Even as app teams adopt modern approaches like agile development and DevOps, they usually remain subject to Conway's Law, which states that "organizations design systems that mirror their own communication structure". In other words, distributed applications are developed by distributed teams that operate independently but share resources. While this structure is ideal for enabling teams to run fast, it also encourages the establishment of siloes. Consequences include poor communication (which has its own consequences), security vulnerabilities, tool sprawl, inconsistent automation practices, and clashes between teams.

ORGANIZATIONS DESIGN SYSTEMS THAT MIRROR THEIR OWN COMMUNICATION STRUCTURE

- **Complexity**

  To implement enterprise-grade microservices technologies, organizations must piece together a suite of critical components that provide visibility, security, and traffic management. Typically, teams use infrastructure platforms, cloud-native services, and open source tools to fill this need. While there is a place for each of these strategies, each has drawbacks that can contribute to complexity. And all too often different teams within a single organization choose different strategies to satisfy the same requirements, resulting in "operational debt". Further, teams choose processes and tools at a point in time, and continue to use them regardless of the changing requirements around deploying and running modern microservices-driven applications using containers.

DIFFERENT TEAMS CHOOSE DIFFERENT STRATEGIES TO SATISFY THE SAME REQUIREMENTS

  The CNCF Cloud Native Interactive Landscape is a good illustration of the complexity of the infrastructure necessary to support microservices-based applications. Organizations need to become proficient in a wide range of disparate technologies, with consequences that include infrastructure lock-in, shadow IT, tool sprawl, and a steep learning curve for those tasked with maintaining the infrastructure.

- **Security**

  Security requirements differ significantly for cloud-native and traditional apps, because strategies such as ringfenced security aren't viable in Kubernetes. The large ecosystem and the distributed nature of containerized apps means the attack surface is much larger, and the reliance on external SaaS applications means that employees and outsiders have many more opportunities to inject malicious code or exfiltrate information. Further, the consequences outlined in the culture and complexity areas – tool sprawl, in particular – have a direct impact on the security and resilience of your modern apps. Using different tools around your ecosystem to solve the same problem is not just inefficient – it creates a huge challenge for SecOps teams who must learn how to properly configure each component.

THE DISTRIBUTED NATURE OF CONTAINERIZED APPS MEANS THE ATTACK SURFACE IS MUCH LARGER

THE SOLUTION: PRODUCTION-GRADE KUBERNETES

As with most organizational problems, the answer to overcoming the challenges of Kubernetes is a combination of technology and processes. This eBook focuses on how you can use traffic management tools to solve these problems – all the way from test to production.

Because Kubernetes is an open source technology, there are numerous ways to implement it. While some organizations prefer to roll their own vanilla Kubernetes, many find value in the combination of flexibility, prescriptiveness, and support provided by managed platforms.

Kubernetes platforms can make it easy to get up and running; however, they focus on breadth of services rather than depth. So, while you may get all the services you need in one place, they're unlikely to offer the feature sets you need for true production readiness at scale. Namely, they don't focus on advanced networking and security, which is where we see Kubernetes disappointing many customers.

To make Kubernetes production-grade, you need to add three more components in this order:

1. **A scalable ingress-egress tier to get traffic in and out of the cluster**
   This is accomplished with an Ingress controller, which is a specialized load balancer that abstracts away the complexity of Kubernetes networking and bridges between services in a Kubernetes cluster and those outside it. This component becomes production-grade when it includes features that increase resilience (for example, advanced health checks and Prometheus metrics), enable rapid scalability (dynamic reconfiguration), and support self-service (role-based access control [RBAC]).

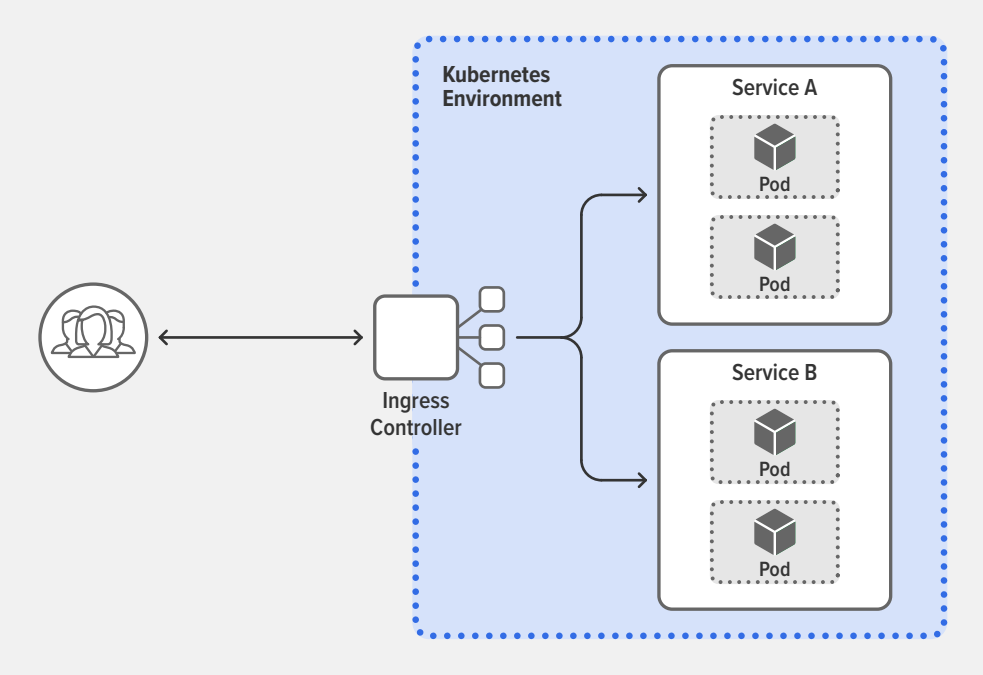


Figure 1: Part 1 of a Production-Grade Kubernetes Deployment Is a Scalable Ingress-Egress Tier

2. **Built-in security to protect against threats throughout the cluster**

   While “coarse-grained” security might be sufficient outside the cluster, “fine-grained” security is required inside it. Depending on the complexity of your cluster, there are three locations where you may need to deploy a flexible web application firewall (WAF): on the Ingress controller, as a per service proxy, and as a per pod proxy. This flexibility lets you apply stricter controls to sensitive apps – such as billing – and looser controls where risk is lower.

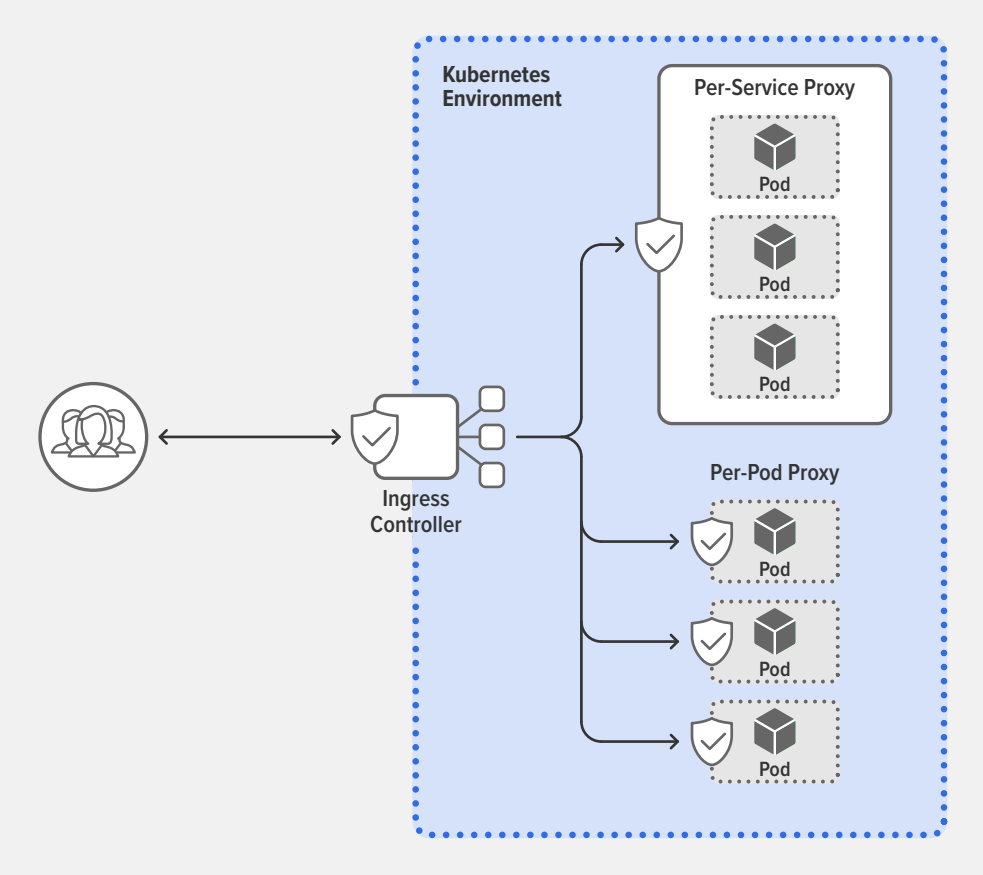


Figure 2: Part 2 of a Production-Grade Kubernetes Deployment Is Built-In Security, Such as a Flexible WAF

3. **A scalable east-west traffic tier to optimize traffic within the cluster**

   This third component is needed once your Kubernetes applications have grown beyond the level of complexity and scale that basic tools can handle. At this stage, you need a service mesh, which is an orchestration tool that provides even finer-grained traffic management and security to application services within the cluster. A service mesh is typically responsible for managing application routing between containerized applications, providing and enforcing autonomous service-to-service mutual TLS (mTLS) policies, and providing visibility into application availability and security. To learn more about when it's time to consider a service mesh, see chapter 9.

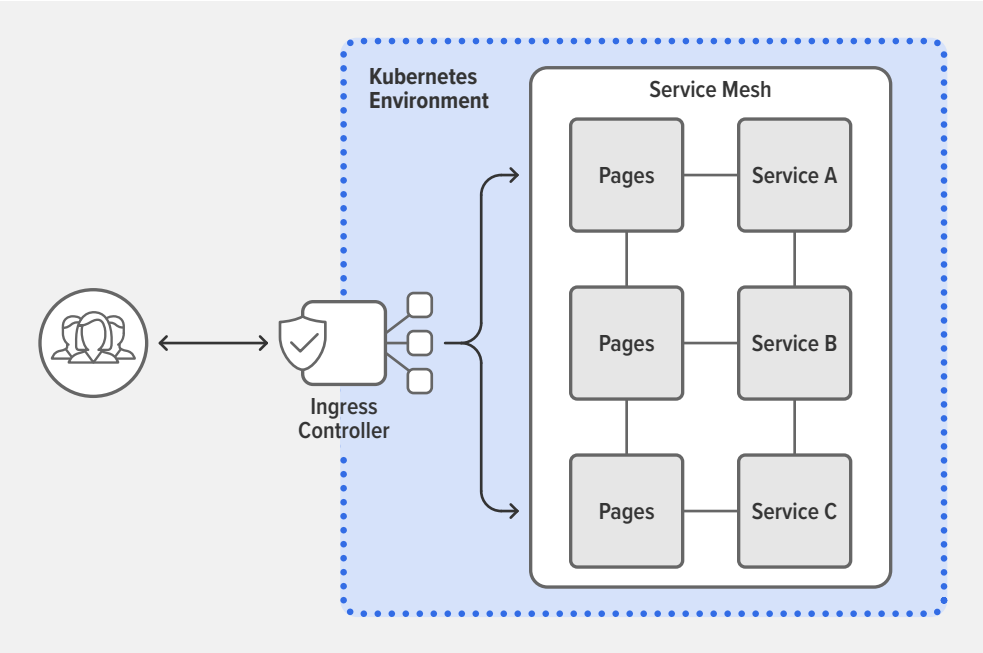


Figure 3: Part 3 of a Production-Grade Kubernetes Deployment Is a Scalable East-West Traffic Tier

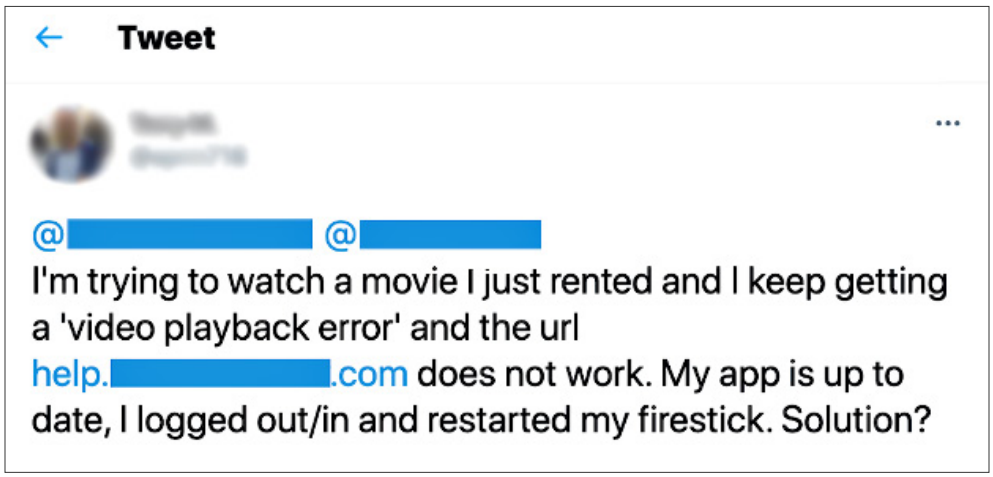
KUBERNETES PRIORITIES: RESILIENCE, VISIBILITY, AND SECURITY

For the remainder of Part One of this eBook, we address the three big Kubernetes problems you can solve with production-grade traffic management tools:

- **Resilience**
  Uptime is a standard SLA for a good reason: customers are fickle and will abandon products that are unreliable or unavailable. In chapter 2, we share six advanced traffic management methods for improving the resilience of your Kubernetes apps and infrastructure.
- **Visibility**
  The importance of visibility is hard to overstate. Without it, your teams simply can't gain the insights necessary to detect or prevent problems before they're discovered by customers. In chapter 3, we demonstrate how to use insights gained from traffic management tools and integrations to troubleshoot two common Kubernetes problems.
- **Security**
  Failure to secure Kubernetes environments and apps prevents organizations from achieving Kubernetes goals. Security vulnerabilities can result in the loss of customers, slowed agility, and stalled digital transformation. In chapter 4, we cover six use cases that will make your Kubernetes environments and apps more secure – all without slowing down your developers.

2. Six Ways to Improve Resilience

There's a very easy way to tell that a company isn't successfully using modern app development technologies – its customers are quick to complain on social media. They complain when they can't stream the latest bingeworthy release. Or can't access online banking. Or can't make a purchase because the cart is timing out.



Even if customers don't complain publicly, that doesn't mean their bad experience doesn't have consequences. One of our customers – a large insurance company – told us that they lose customers when their homepage doesn't load *within 3 seconds*.

All those user complaints of poor performance or outages point to a common culprit: resilience . . . or the lack of it. The beauty of microservices technologies – including containers and Kubernetes – is that they can significantly improve the customer experience by improving the resilience of your apps. How? It's all about the architecture.

I like to explain the core difference between monolithic and microservices architectures by using the analogy of a string of holiday lights. When a bulb goes out on an older-style strand, the entire strand goes dark. If you can't replace the bulb, the only thing worth decorating with that strand is the inside of your garbage can. This old style of lights is like a monolithic app, which also has tightly coupled components and fails if one component breaks.

But the lighting industry, like the software industry, detected this pain point. When a bulb breaks on a modern strand of lights, the others keep shining brightly, just as a well designed microservices app keeps working even when a service instance fails.

ADVANCED TRAFFIC MANAGEMENT USE CASES

Containers are a popular choice in microservices architectures because they are ideally suited for building an application using smaller, discrete components – they are lightweight, portable, and easy to scale. Kubernetes is the de facto standard for container orchestration, but there are a lot of challenges around making Kubernetes production-ready. One element that improves both your control over Kubernetes apps and their resilience is a mature traffic management strategy that allows you to control services rather than packets, and to adapt traffic management rules dynamically or with the Kubernetes API. While traffic management is important in any architecture, for high performance apps two traffic management tools are essential: traffic control and traffic splitting.

- **Traffic control** (sometimes called *traffic routing* or *traffic shaping*) refers to the act of controlling where traffic goes and how it gets there. It's a necessity when running Kubernetes in production because it allows you to protect your infrastructure and apps from attacks and traffic spikes. Two techniques to incorporate into your app development cycle are *rate limiting* and *circuit breaking*.
- **Traffic splitting** (sometimes called *traffic testing*) is a subcategory of traffic control and refers to the act of controlling the proportion of incoming traffic directed to different versions of a backend app running simultaneously in an environment (usually the current production version and an updated version). It's an essential part of the app development cycle because it allows teams to test the functionality and stability of new features and versions without negatively impacting customers. Useful deployment scenarios include *debug routing*, *canary deployments*, *A/B testing*, and *blue-green deployments*. (There is a fair amount of inconsistency in the use of these four terms across the industry. Here we use them as we understand their definitions.)

In this chapter, we first explore use cases you can resolve using traffic control and traffic shaping techniques. We then explain how to implement several techniques with NGINX solutions. The six use cases are:

1. Protect services from getting too many requests
2. Avoid cascading failures
3. Test a new version in production
4. Ensure a new version is stable
5. Find out if customers like a new version better than the current version
6. Move users to a new version without downtime

RESILIENCE USE CASE #1:
Protect Services from Getting Too Many Requests

Solution: Rate limiting

RATE LIMITING RESTRICTS THE NUMBER OF REQUESTS A USER CAN MAKE IN A GIVEN TIME PERIOD

Whether malicious (for example, brute force password guessing and DDoS attacks) or benign (such as customers flocking to a sale), a high volume of HTTP requests can overwhelm your services and cause your apps to crash. Rate limiting restricts the number of requests a user can make in a given time period. Requests can include something as simple as a GET request for the homepage of a website or a POST request on a login form. When under DDoS attack, for example, you can use rate limiting to limit the incoming request rate to a value typical for real users.

RESILIENCE USE CASE #2:
Avoid Cascading Failures

Solution: Circuit breaking

When a service is unavailable or experiencing high latency, it can take a long time for incoming requests to time out and clients to receive an error response. The long timeouts can potentially cause a cascading failure, in which the outage of one service leads to timeouts at other services and the ultimate failure of the application as a whole.

CIRCUIT BREAKERS PREVENT CASCADING FAILURE BY MONITORING FOR SERVICE FAILURES

Circuit breakers prevent cascading failure by monitoring for service failures. When the number of failed requests to a service exceeds a preset threshold, the circuit breaker trips and starts returning an error response to clients as soon as the requests arrive, effectively throttling traffic away from the service.

The circuit breaker continues to intercept and reject requests for a defined amount of time before allowing a limited number of requests to pass through as a test. If those requests are successful, the circuit breaker stops throttling traffic. Otherwise, the clock resets and the circuit breaker again rejects requests for the defined time.

RESILIENCE USE CASE #3:
Test a New Version in Production

Solution: Debug routing

DEBUG ROUTING LETS YOU DEPLOY AN APP PUBLICLY YET "HIDE" IT FROM MOST USERS.

Let's say you have a banking app and you're going to add a credit score feature. Before testing with customers, you probably want to see how it performs in production. Debug routing (also known as conditional routing) lets you deploy it publicly yet "hide" it from actual users by allowing only certain users to access it, based on Layer 7 attributes such as a session cookie, session ID, or group ID. For example, you can allow access only to users who have an admin session cookie – their requests are routed to the new version with the credit score feature while everyone else continues on the stable version

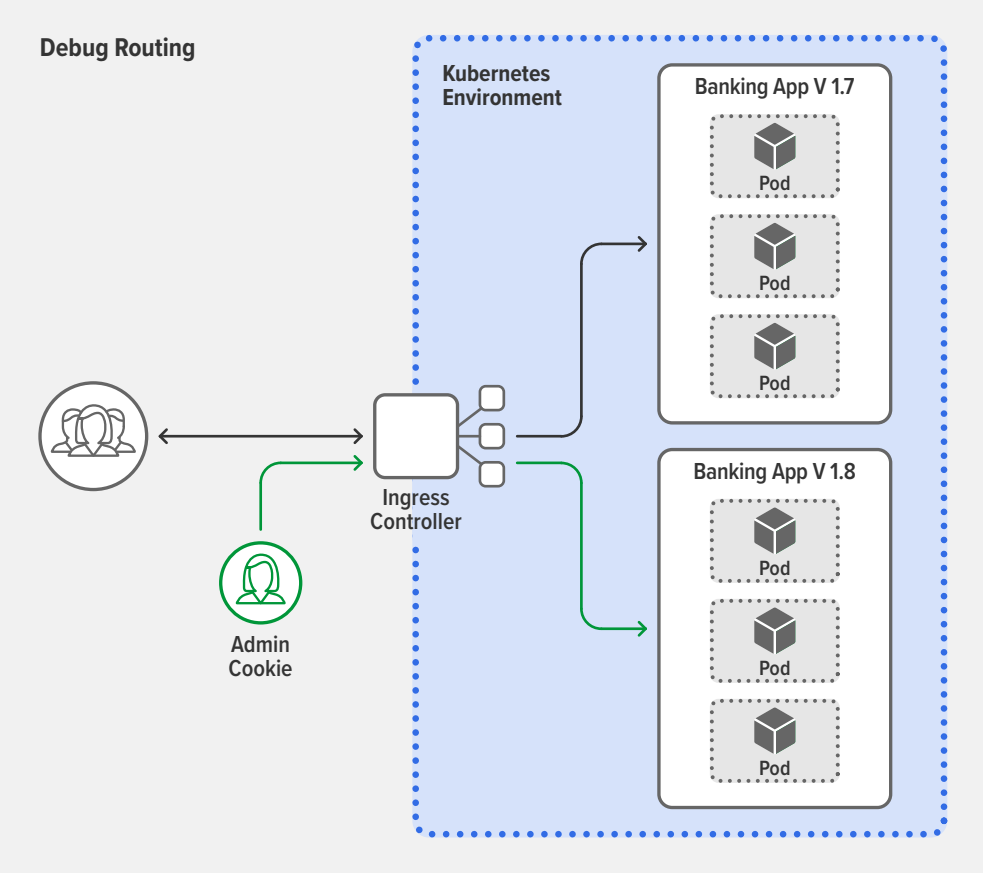


Figure 4: Debug Routing in a Kubernetes Environment

RESILIENCE USE CASE #4:
Ensure a New Version is Stable

Solution: Canary deployment

The concept of canary deployment is taken from an historic mining practice, where miners took a caged canary into a coal mine to serve as an early warning of toxic gases. The gas killed the canary before killing the miners, allowing them to quickly escape danger. In the world of apps, no birds are harmed! Canary deployments provide a safe and agile way to test the stability of a new feature or version. A typical canary deployment starts with a high share (say, 99%) of your users on the stable version and moves a tiny group (the other 1%) to the new version. If the new version fails, for example crashing or returning errors to clients, you can immediately move the test group back to the stable version. If it succeeds, you can switch users from the stable version to the new one, either all at once or (as is more common) in a gradual, controlled migration.

CANARY DEPLOYMENTS PROVIDE A SAFE AND AGILE WAY TO TEST THE STABILITY OF A NEW FEATURE OR VERSION

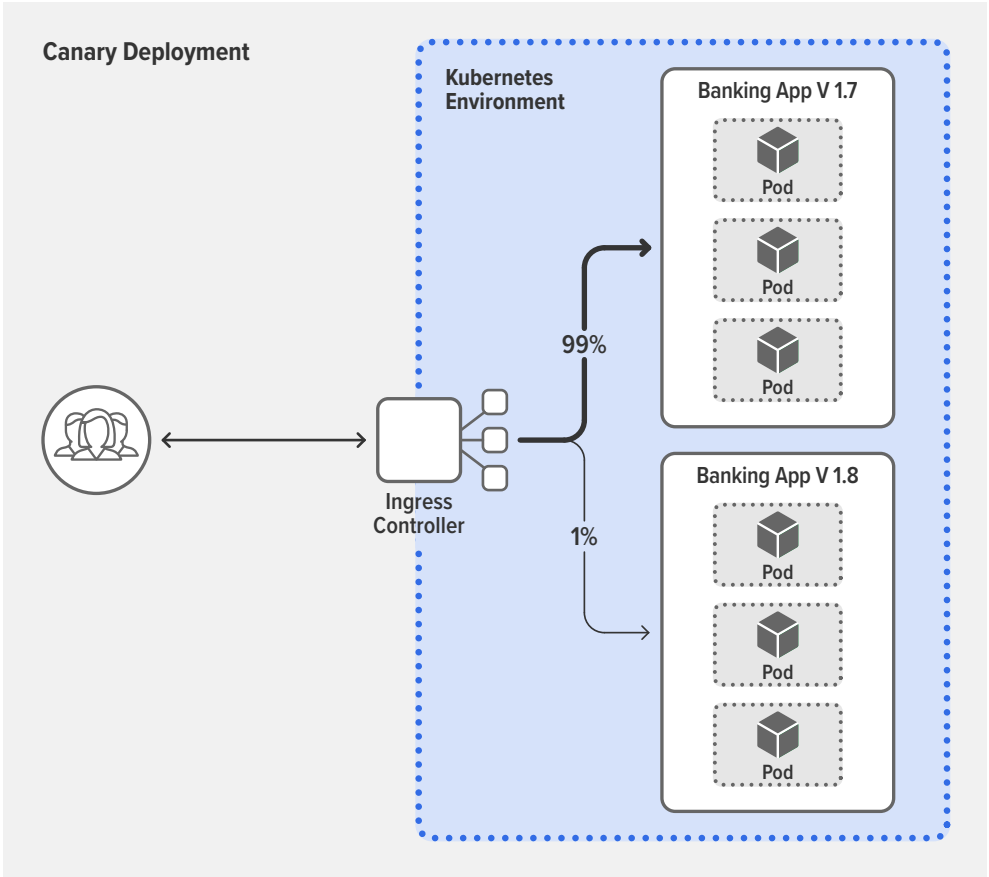


Figure 5: Canary Deployment in a Kubernetes Environment

RESILIENCE USE CASE #5:
Find Out if Customers Like a New Version Better than the Current Version

Solution: A/B testing

A/B TESTING IS A PROCESS USED ACROSS MANY INDUSTRIES TO MEASURE AND COMPARE USER BEHAVIOR

Now that you've confirmed your new feature works in production, you might want to get customer feedback about the success of the feature, based on key performance indicators (KPIs) such as number of clicks, repeat users, or explicit ratings. A/B testing is a process used across many industries to measure and compare user behavior for the purpose of determining the relative success of different product or app versions across the customer base. In a typical A/B test, 50% of users get Version A (the current app version) while the remaining 50% gets Version B (the version with the new, but stable, feature). The winner is the one with the overall better set of KPIs.

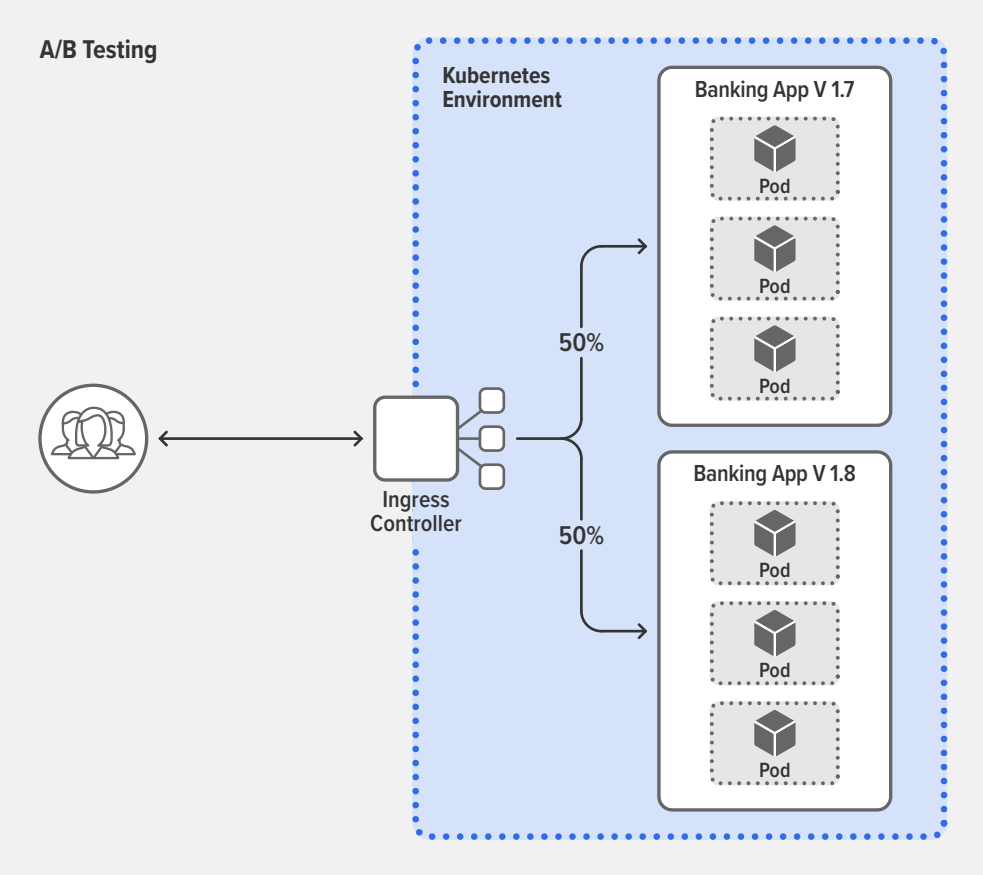


Figure 6: A/B Testing in a Kubernetes Environment

RESILIENCE USE CASE #6:
Move Users to a New Version Without Downtime

Solution: Blue-green deployment

BLUE-GREEN DEPLOYMENTS GREATLY REDUCE, OR EVEN ELIMINATE, DOWNTIME FOR UPGRADES

Now let's say your banking app is due for a major version change . . . congratulations! In the past, upgrading versions usually meant downtime for users because you had to take down the old version before moving the new version into production. But in today's competitive environment, downtime for upgrades is unacceptable to most users. Blue-green deployments greatly reduce, or even eliminate, downtime for upgrades. Simply keep the old version (blue) in production while simultaneously deploying the new version (green) alongside in the same production environment.

Most organizations don't want to move 100% of users from blue to green at once – after all, what if the green version fails?! The solution is to use a canary deployment to move users in whatever increments best meet your risk mitigation-strategy. If the new version is a disaster, you can easily revert everyone back to the stable version in just a couple of keystrokes.

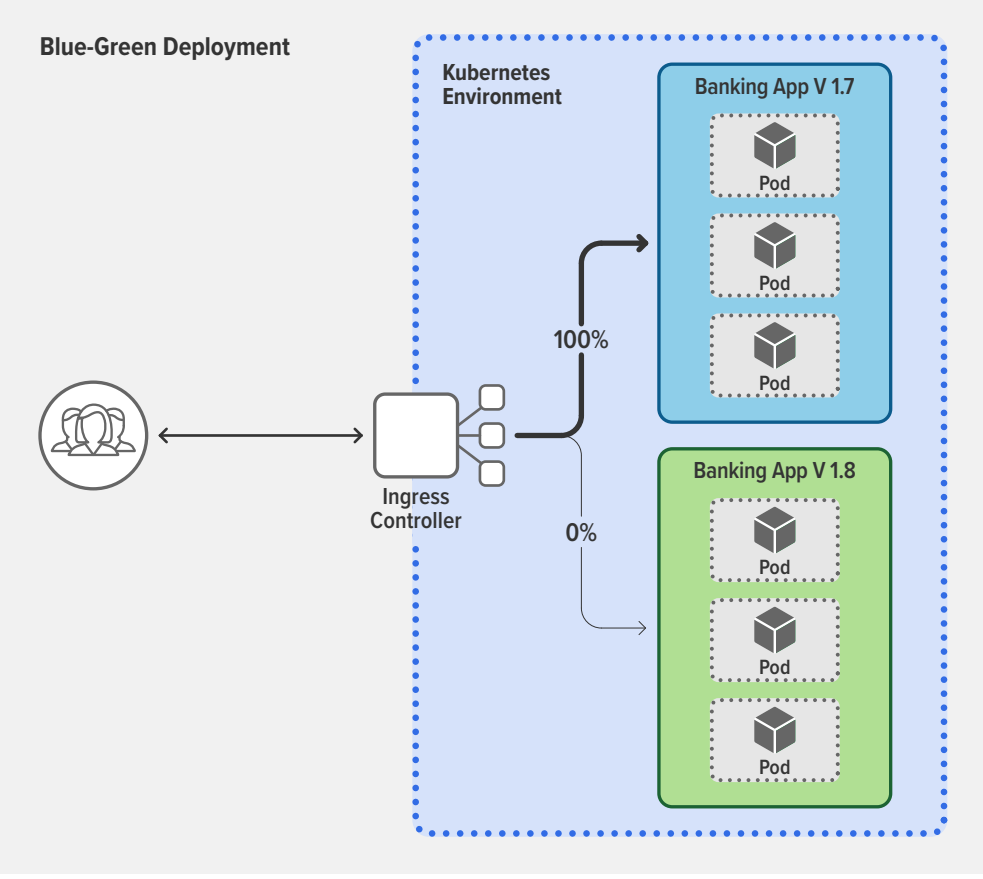


Figure 7: Blue-Green Deployment in a Kubernetes Environment

USING NGINX FOR ADVANCED KUBERNETES TRAFFIC MANAGEMENT

You can accomplish advanced traffic control and splitting with most Ingress controllers and service meshes. Which technology to use depends on your app architecture and use cases. For example, using an Ingress controller makes sense in these three scenarios:

- Your apps have just one endpoint, as with simple apps or monolithic apps that you have "lifted and shifted" into Kubernetes.
- There's no service-to-service communication in your cluster.
- There is service-to-service communication but you aren't yet using a service mesh.

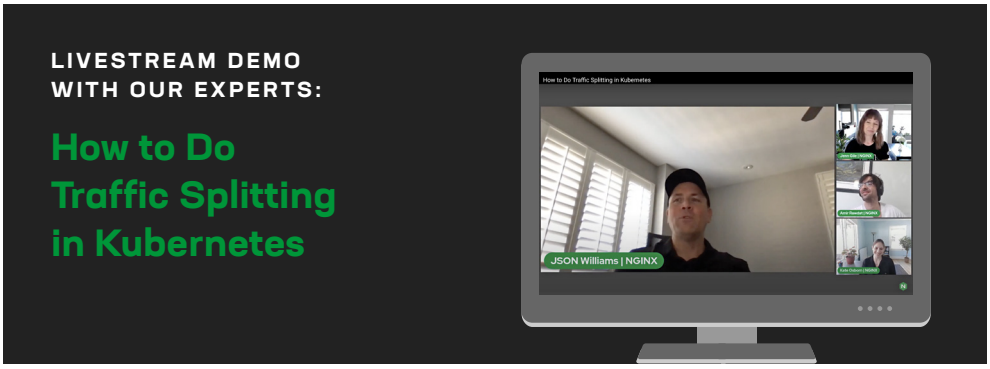
If your deployment is complex enough to need a service mesh, a common use case is splitting traffic between services for testing or upgrade of individual microservices. For example, you might want to do a canary deployment behind your mobile front-end, between two different versions of your geolocation microservice API.

However, setting up traffic splitting with some Ingress controllers and service meshes can be time-consuming and error-prone, for a variety of reasons:

- Ingress controllers and service meshes from various vendors implement Kubernetes features in different ways.
- Kubernetes isn't really designed to manage and understand Layer 7 traffic.
- Some Ingress controllers and service meshes don't support sophisticated traffic management.

EASILY CONFIGURE ROBUST TRAFFIC ROUTING AND SPLITTING POLICIES IN SECONDS

With F5 NGINX Ingress Controller and F5 NGINX Service Mesh, you can easily configure robust traffic routing and splitting policies in seconds with easier configs, advanced customizations, and improved visibility.



Easier Configuration with NGINX Ingress Resources and the SMI Specification

ANNOTATIONS, CONFIGMAPS, AND CUSTOM TEMPLATES LACK FINE-GRAINED CONTROL, ARE INSECURE, AND ARE ERROR PRONE AND DIFFICULT TO USE

These NGINX features make configuration easier:

- **NGINX Ingress resources for NGINX Ingress Controller** – While the standard Kubernetes Ingress resource makes it easy to configure SSL/TLS termination, HTTP load balancing, and Layer 7 routing, it doesn't include the kind of customization features required for circuit breaking, A/B testing, and blue-green deployment. Instead, non-NGINX users have to use annotations, ConfigMaps, and custom templates which all lack fine-grained control, are insecure, and are error prone and difficult to use.

  NGINX Ingress Controller comes with NGINX Ingress resources as an alternative to the standard Ingress resource (which it also supports). They provide a native, type-safe, and indented configuration style which simplifies implementation of Ingress load balancing. NGINX Ingress resources have an added benefit for existing NGINX users: they make it easier to repurpose load balancing configurations from non-Kubernetes environments, so all your NGINX load balancers can use the same configurations.

- **NGINX Service Mesh with SMI** – NGINX Service Mesh implements the Service Mesh Interface (SMI), a specification that defines a standard interface for service meshes on Kubernetes, with typed resources such as `TrafficSplit`, `TrafficTarget`, and `HTTPRouteGroup`. Using standard Kubernetes configuration methods, NGINX Service Mesh and the NGINX SMI extensions make traffic-splitting policies, like canary deployment, simple to deploy with minimal interruption to production traffic. For example, here's how to define a canary deployment with NGINX Service Mesh:

```
apiVersion: split.smi-spec.io/v1alpha2
kind: TrafficSplit
metadata:
name: target-ts
spec:
service: target-svc
backends:
- service: target-v1-0
   weight: 90
- service: target-v2-0
   weight: 10
```

Our tutorial How to Use NGINX Service Mesh for Traffic Splitting walks through sample deployment patterns that leverage traffic splitting, including canary and blue-green deployments.

MORE SOPHISTICATED TRAFFIC CONTROL AND SPLITTING WITH ADVANCED CUSTOMIZATIONS

These NGINX features make it easy to control and split traffic in sophisticated ways:

- **The key-value store for canary deployments** – When you're performing A/B testing or blue-green deployments, you might want to transition traffic to the new version at specific increments, for example 0%, 5%, 10%, 25%, 50%, and 100%. With most tools, this is a very manual process because you have to edit the percentage and reload the configuration file for each increment. With that amount of overhead, you might decide to take the risk of going straight from 5% to 100%. However, with the NGINX Plus-based version of NGINX Ingress Controller, you can leverage the key-value store to change the percentages without the need for a reload.
- **Circuit breaking with NGINX Ingress Controller** – Sophisticated circuit breakers save time and improve resilience by more quickly detecting failures and failing over, and even activating custom, formatted error pages for upstreams that are unhealthy. For example, a circuit breaker for a search service might be configured to return a correctly formatted but empty set of search results. To get this level of sophistication, the NGINX Plus-based version of NGINX Ingress Controller uses active health checks which proactively monitor the health of your TCP and UDP upstream servers. Because it's monitoring in real time, your clients will be less likely to experience apps that return errors.
- **Circuit breaking with NGINX Service Mesh** – The NGINX Service Mesh circuit breaker spec has three custom fields:
  - `errors` – Number of errors before the circuit trips
  - `timeoutSeconds` – Window for errors to occur within before tripping the circuit, as well as the amount of time to wait before closing the circuit
  - `fallback` – Name and port of the Kubernetes service to which traffic is rerouted after the circuit has been tripped

  While `errors` and `timeoutSeconds` are standard circuit breaker features, `fallback` boosts resilience further by enabling you to define a backup server. If your backup server responses are unique, they can be an early indicator of trouble in your cluster, allowing you to start troubleshooting right away.

SOPHISTICATED CIRCUIT BREAKERS SAVE TIME AND IMPROVE RESILIENCE BY MORE QUICKLY DETECTING FAILURES AND FAILING OVER

Interpreting Traffic Splitting Results

You've implemented traffic splitting . . . now what? It's time to analyze the result. This can be the hardest part because many organizations are missing key insights into how their Kubernetes traffic and apps are performing. For more on improving visibility to gain insight, continue to chapter 3.

3. Two Problems You Can Solve with Better Visibility

YOUR ARCHITECTURE MAY BE PUTTING YOU AT INCREASED RISK

Adoption of microservices accelerates digital experiences, but microservices architectures can also make those experiences more fragile. While your developers are running fast to get new apps out the door, your architecture may be putting you at increased risk for outages, security exposures, and time wasted on inefficient troubleshooting or fixing preventable problems. In this chapter we examine how components that provide traffic visibility can reduce complexity and improve security in your microservices environments.

HOW TO ATTAIN INSIGHTS THROUGH VISIBILITY

First, let's look at a couple of definitions:

- **Visibility** – The state of being able to see or be seen
- **Insight** – A deep understanding of a person or thing

In a 2021 survey from Splunk, 81% of respondents identified data as "very" or "extremely" valuable. We agree that data is key, especially in Kubernetes where it can be especially difficult to know what is deployed. And yet 95% of respondents to F5's The State of Application Strategy in 2021 reported that although they have a wealth of data, they're missing the insights into app performance, security, and availability that they need to protect and evolve their infrastructure and business. So why is insight important and how do you get it?

With Insight, You Can:

- Strengthen security and compliance by detecting vulnerabilities and possible attack vectors
- Reduce outages and downtime by discovering problems before your customers do
- Improve the efficiency of trouble-shooting by finding the root cause of app issues
- Confirm that your traffic is going only where you want it to go
- Know exactly what's running in your Kubernetes environments and whether it's properly configured and secured
- Figure out whether you're using the right amount of resources based on latency and performance history
- Predict seasonal needs based on past traffic patterns
- Measure response time to track performance against SLAs and serve as an early warning system before problems affect the user experience

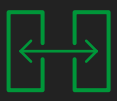
PEOPLE FAIL TO ATTAIN VALUABLE INSIGHTS DUE TO ORGANIZATIONAL FACTORS

To gain insight, you need two types of visibility data: real-time and historical. Real-time data enables you to diagnose the source of a problem that's happening right now, while historical data supplies perspective on what is "normal" versus what is an outlier. Combined, these two types of visibility sources can provide crucial insights into app and Kubernetes performance.

As with other technology investments, you also need a strategy for how to reap the benefits. The F5 report also indicates that people fail to attain valuable insights due to organizational factors related to hiring and employee development, strategy and processes, and consensus on what the data should be used for, when, and by whom. Those findings include:

Related Skillsets

It's no secret that there's a shortage of skilled technology professionals, as confirmed by 47% of respondents reporting that they struggle to find the talent they need.



Data Sharing Initiatives

Only 12% of respondents have processes and strategies in place for reporting data back to business decision makers to make them aware of the business impacts of resilient technology (or lack thereof).

The Purpose of Visibility

Most respondents use telemetry reactively (that is, for troubleshooting) while only 24% of respondents proactively use data and insights to watch for potential performance degradations and 16% to track SLA performance.

TROUBLESHOOTING TWO KUBERNETES PROBLEMS

Most Kubernetes deployments are already using a monitoring tool and don't need yet another one. But did you know you can use your production-grade traffic management tools – specifically an Ingress controller and service mesh – to gain insights? Because they touch every bit of traffic in your Kubernetes clusters, the Ingress controller and service mesh have the potential to provide data that can help you improve your uptime SLAs.

- **Insight into ingress-egress (north-south) traffic**
  The Ingress controller provides insight into the traffic entering and exiting your Kubernetes clusters.
- **Insight into service-to-service (east-west) traffic**
  A service mesh provides insight into the traffic flowing among containerized apps.

To make this level of visibility possible with NGINX tools, we instrumented the F5 NGINX Plus API for easy export of metrics and provide integrations with popular tools like OpenTracing and Grafana and Prometheus, so you can get a full picture of performance inside your clusters. You get targeted insights into app performance and availability with deep traces so you can understand how requests are processed across your microservices apps.

In the remainder of this chapter, we demonstrate six ways to use F5 NGINX Ingress Controller and F5 NGINX Service Mesh for troubleshooting two common Kubernetes problems:

1. Slow apps
2. Resource exhaustion

Kubernetes Problem #1: Slow Apps

YOU CAN'T BEGIN TO SOLVE THE PROBLEM UNTIL YOU FIGURE OUT EXACTLY WHERE IT LIES

Is your app running slow . . . or even completely down? Do you suspect a DDoS attack? Are users reporting errors from your website? You can't begin to solve the problem until you figure out exactly where it lies. Depending on the complexity of your apps and which NGINX tools you're using, you have three options for getting real-time metrics so you can diagnose what's happening "right now".

- **Option 1: Live monitoring with NGINX Ingress Controller**
  With NGINX Plus, the live activity monitoring dashboard (powered by the NGINX Plus API) displays hundreds of key load and performance metrics. Get fine-grained detail down to the level of a single pod so you can quickly and easily measure response times to apps and diagnose the source of an issue. As your Kubernetes environment grows, you automatically get dashboards for each additional NGINX Ingress Controller instance.

  As an example, two columns on the **HTTP Upstreams** tab give you an instant read on application and infrastructure status:

  – **Requests** – If the number of requests per second (**Req/s**) is dipping below the norm for the given application (for example, 5 requests per second when 40 is normal), the NGINX Ingress Controller or application might be misconfigured.

  – **Response time** – If response times are 10 milliseconds (ms) or less, you're in great shape. Latency upwards of 30–40ms is a sign of trouble with your upstream applications.

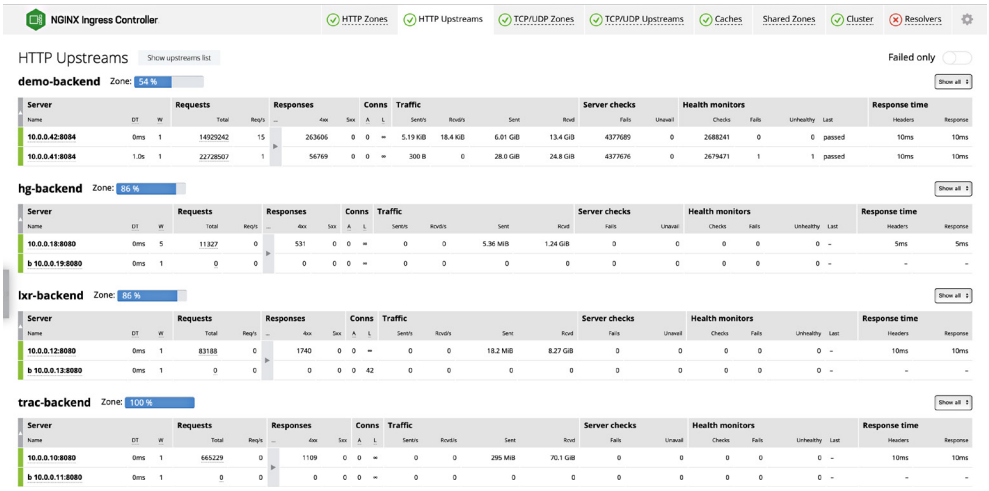


Figure 8: NGINX Ingress Controller Live Monitoring Dashboard

- **Option 2: Stub status for NGINX Ingress Controller**
  With NGINX Open Source, NGINX Ingress Controller includes a status page that reports eight basic metrics.
- **Option 3: OpenTracing with NGINX Service Mesh**
  NGINX Service Mesh supports OpenTracing with the NGINX OpenTracing module. As of this writing, the module supports Datadog, LightStep, Jaeger, and Zipkin.

Kubernetes Problem #2: Resource Exhaustion

503 AND 40X ERRORS INDICATE THAT THERE'S A PROBLEM WITH YOUR RESOURCES, WHILE 502s MEAN THAT A CONFIG CHANGE DIDN'T WORK

Got HTTP errors? 503 and 40x errors indicate that there's a problem with your resources, while 502s mean that a config change didn't work. To diagnose where you might be running out of resources, you need to be able to export metrics to visibility tools that plot values over time. In addition to discovering the source of your problems, historical data can also help you predict when a traffic surge will happen so you'll be ready to scale.

- **Option 1: Logging with NGINX Ingress Controller**
  The first step in diagnosing network issues is to check out the NGINX Ingress Controller logs, in which every log entry (including those about errors) identifies the associated Kubernetes service. The logs include detailed information about all the traffic that has come through NGINX Ingress Controller, including a timestamp, source IP address, and response status code. You can also export logs to popular aggregators such as Datadog, Grafana, and Splunk.
- **Option 2: Prometheus metrics**
  One of NGINX Ingress Controller's most popular features is its ever-expanding list of Prometheus metrics, which include metrics on network performance and Ingress controller traffic. The NGINX Plus-based NGINX Ingress Controller exports metrics about connections, caching, HTTP and TCP/UDP traffic handled by groups of NGINX workers that share data in a memory zone, HTTP and TCP/UDP traffic handled by groups of backend servers, and more.

  NGINX Service Mesh uses the NGINX Plus API to scrape metrics from NGINX Service Mesh sidecars and NGINX Ingress Controller pods. Prometheus scrape configurations are included so you can customize the metrics needed in your Prometheus configuration file.
- **Option 3: Grafana dashboards**
  We provide official Grafana dashboards for NGINX Ingress Controller and NGINX Service Mesh that visualize metrics exposed by the Prometheus Exporter. Users value the granularity of the data, which includes detail down to the millisecond, day-over-day overlays, and traffic spikes. For example, the NGINX Service Mesh dashboard can indicate your pods are at capacity by displaying the amount of traffic on any one service or pod and the number of active pods being monitored.

If you're ready to see the technology in action, check out this livestream demo and AMA with NGINX and Grafana experts, How to Improve Visibility in Kubernetes with Prometheus, Grafana, and NGINX. You'll see them demonstrate how to get live monitoring of key load balancing and performance metrics, export the metrics to Prometheus, and create Grafana dashboards for a view of cumulative performance.

4. Six Methods to Enhance Security

As discussed in Secure Cloud-Native Apps Without Losing Speed on our blog, we have observed three factors that make cloud-native apps more difficult to secure than traditional apps:

1. Cloud-native app delivery causes tool sprawl and offers inconsistent enterprise-grade services
2. Cloud-native app delivery costs can be unpredictable and high
3. SecOps struggles to protect cloud-native apps and is at odds with DevOps

While all three factors can equally impact security, the third factor can be the most difficult problem to solve, perhaps because it's the most "human". When SecOps isn't able or empowered to protect cloud-native apps, some of the consequences are obvious (vulnerabilities and breaches), but others are hidden, including slowed agility and stalled digital transformation.

WHEN THERE ARE SECURITY INCIDENTS IN A KUBERNETES ENVIRONMENT, MOST ORGANIZATIONS PULL THEIR KUBERNETES DEPLOYMENTS OUT OF PRODUCTION

Let's dig deeper into those hidden costs. Organizations choose Kubernetes for its promise of agility and cost savings. But when there are security incidents in a Kubernetes environment, most organizations pull their Kubernetes deployments out of production. That slows down digital transformation initiatives that are essential for the future of the organization – never mind the wasted engineering efforts and money. The logical conclusion is: if you're going to try to get Kubernetes from test to production, then security must be considered a strategic component that is owned by everyone in the organization.

The security tool ecosystem is a huge (and growing) industry with both tech giants and start-ups jumping to address Kubernetes-specific problems. But even with the adoption of security tools, such as vulnerability scanners and WAFs, organizations still find themselves vulnerable. Just like with your visibility and troubleshooting strategies, you can leverage Kubernetes traffic management tools to improve and simplify your security strategy.

SECURITY AND IDENTITY TERMINOLOGY

Before we jump into the use cases, here's an overview of the security and identity terms you'll encounter throughout this chapter. Already a security pro? Click here to jump to the use cases.

- **Authentication and authorization** – Functions required to ensure only the "right" users and services can gain access to backends or application components:
  - **Authentication** – Verification of *identity* to ensure that clients making requests are who they claim to be. Accomplished through ID tokens, such as passwords or JSON Web Tokens (JWTs).
  - **Authorization** – Verification of *permission* to access a resource or function. Accomplished through access tokens, such as Layer 7 attributes like session cookies, session IDs, group IDs, or token contents.

- **Critical Vulnerabilities and Exposures (CVEs)** – A database of publicly disclosed flaws "in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components".[1] CVEs may be discovered by the developers who manage the tool, a penetration tester, a user or customer, or someone from the community (such as a "bug hunter"). It's common practice to give the software owner time to develop a patch before the vulnerability is publicly disclosed, so as not to give bad actors an advantage.
- **Denial-of-service (DoS) attack** – An attack in which a bad actor floods a website with requests (TCP/UDP or HTTP/HTTPS) with the goal of making the site crash. Because DoS attacks impact availability, their primary outcome is damage to the target's reputation. A **distributed denial-of-service (DDoS) attack**, in which multiple sources target the same network or service, is more difficult to defend against due to the potentially large network of attackers. DoS protection requires a tool that adaptively identifies and prevents attacks. Learn more in What is Distributed Denial of Service (DDoS)?
- **End-to-end encryption (E2EE)** – The practice of fully encrypting data as it passes from the user to the app and back. E2EE requires SSL/TLS certificates and potentially mTLS.
- **Mutual TLS (mTLS)** – The practice of requiring authentication (via SSL/TLS certificate) for both the client and the host. Use of mTLS also protects the confidentiality and integrity of the data passing between the client and the host. mTLS can be accomplished all the way down to the Kubernetes pod level, between two services in the same cluster. For and excellent introduction to SSL/TLS and mTLS, see What is mTLS? at F5 Labs.
- **Single sign-on (SSO)** – SSO technologies (including SAML, OAuth, and OIDC) make it easier to manage authentication and authorization.
  - **Simplified authentication** – SSO eliminates the need for a user to have a unique ID token for each different resource or function.
  - **Standardized authorization** – SSO facilitates setting of user access rights based on role, department, and level of seniority, eliminating the need to configure permissions for each user individually.
- **SSL (Secure Sockets Layer)/TLS (Transport Layer Security)** – A protocol for establishing authenticated and encrypted links between networked computers. SSL/TLS certificates authenticate a website's identity and establish an encrypted connection. Although the SSL protocol was deprecated in 1999 and replaced with the TLS protocol, it is still common to refer to these related technologies as "SSL" or "SSL/TLS" – for the sake of consistency, we use SSL/TLS in this chapter.
- **Web application firewall (WAF)** – A reverse proxy that detects and blocks sophisticated attacks against apps and APIs (including OWASP Top 10 and other advanced threats) while letting "safe" traffic through. WAFs defend against attacks that try to harm the target by stealing sensitive data or hijacking the system. Some vendors consolidate WAF and DoS protection in the same tool, whereas others offer separate WAF and DoS tools.

1. Source: CVE Program.

- **Zero trust** – A security concept that is frequently used in higher security organizations, but is relevant to everyone, in which data must be secured at all stages of data storage and transport. This means that the organization has decided not to "trust" any users or devices by default, but rather require that all traffic is thoroughly vetted. A zero-trust architecture typically includes a combination of authentication, authorization, and mTLS with a high probability that the organization implements end-to-end encryption.

MAKE SECURITY EVERYONE'S RESPONSIBILITY

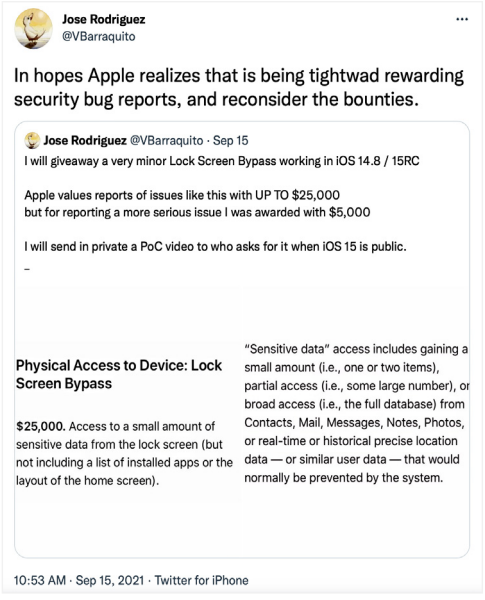
In this chapter, we cover six security use cases that you can solve with Kubernetes traffic management tools, enabling SecOps to collaborate with DevOps and NetOps to better protect your cloud-native apps and APIs. A combination of these techniques is often used to create a comprehensive security strategy designed to keep apps and APIs safe while minimizing the impact to customers.

1. Resolve CVEs quickly to avoid cyberattacks
2. Stop OWASP Top Ten and DoS attacks
3. Offload authentication and authorization from apps
4. Set up self-service with guardrails
5. Implement end-to-end encryption
6. Ensure clients are using a strong cipher with a trusted implementation

SECURITY USE CASE #1:
Resolve CVEs Quickly to Avoid Cyberattacks

Solution: Use tools with timely and proactive patch notifications

According to a study by the Ponemon Institute, in 2019 there was an average "grace period" of 43 days between the release of a patch for a critical or high-priority vulnerability and organizations seeing attacks that tried to exploit the vulnerability. At F5 NGINX, we've seen that window narrow significantly in the following years (even down to day zero in the case of Apple iOS 15 in 2021), which is why we recommend patching as soon as possible. But what if patches for your traffic management tools aren't available for weeks, or even months, after a CVE is announced?



IN ONE CASE, IT TOOK OPENRESTY FOUR MONTHS TO APPLY AN NGINX-RELATED SECURITY PATCH

Tools that are developed and maintained by community contributors (rather than a dedicated engineering team) have the potential to lag weeks to months behind CVE announcements, making it unlikely that organizations can patch within that 43-day window. In one case, it took OpenResty four months to apply an NGINX-related security patch. That left anyone using an OpenResty-based Ingress controller vulnerable for at least four months. However, realistically there's usually additional delay before patches are available for software that depends on an open source project.

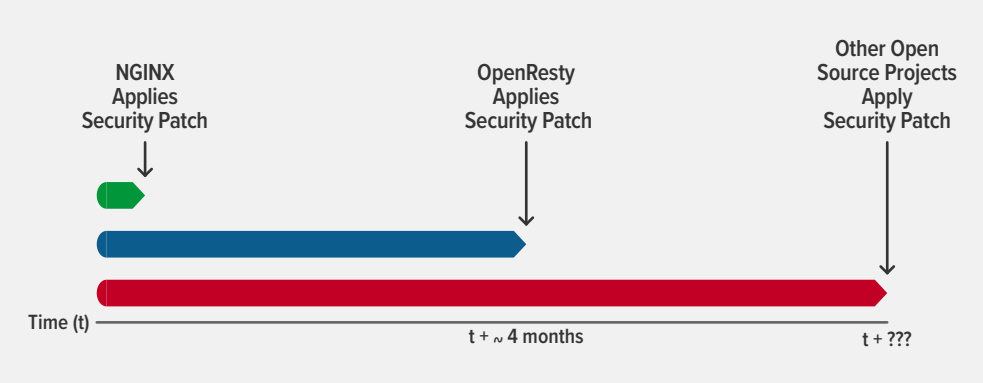


Figure 9: Example of Delay in Implementing a Security Patch

To get the fastest CVE patching, look for two characteristics when selecting traffic management tools:

- **A dedicated engineering team** – When tool development is managed by an engineering team instead of community volunteers, you can be confident a group of people are dedicated to the health of the tool and will prioritize release of a patch as soon as possible.
- **An integrated code base** – Without any external dependencies (like we discussed with OpenResty), patches are just an agile sprint away.

For more on CVE patching, read Mitigating Security Vulnerabilities Quickly and Easily with NGINX Plus on our blog.

SECURITY USE CASE #2:
Stop OWASP Top 10 and DoS Attacks

Solution: Deploy flexible, Kubernetes-friendly WAF and DoS protection

Choosing the right WAF and DoS protection for Kubernetes apps depends on two factors (in addition to features):

USING THE SAME TOOL FOR ALL YOUR DEPLOYMENTS ENABLES YOU TO REUSE POLICIES AND LOWERS THE LEARNING CURVE FOR YOUR SECOPS TEAMS

- **Flexibility** – There are scenarios when it's best to deploy tools inside Kubernetes, so you want infrastructure-agnostic tools that can run within or outside Kubernetes. Using the same tool for all your deployments enables you to reuse policies and lowers the learning curve for your SecOps teams.
- **Footprint** – The best Kubernetes tools have a small footprint, which allows for appropriate resource consumption with minimal impact to throughput, requests per second, and latency. Given that DevOps teams often resist security tools because of a perception that they slow down apps, choosing a high-performance tool with a small footprint can increase the probability of adoption.

While a tool that consolidates WAF and DoS protection may seem more efficient, it's actually expected to have issues around both CPU usage (due to a larger footprint) and flexibility. You're forced to deploy the WAF and DoS protection together, even when you don't need both. Ultimately, both issues can drive up the total cost of ownership for your Kubernetes deployments while creating budget challenges for other essential tools and services.

Once you've chosen the right security tools for your organization, it's time to decide where to deploy those tools. There are four locations where application services can typically be deployed to protect Kubernetes apps:

- **At the front door** (on an external load balancer such as F5 NGINX Plus or F5 BIG-IP) – Good for "coarse-grained" global protection because it allows you to apply global policies across multiple clusters
- **At the edge** (on an Ingress controller such as F5 NGINX Ingress Controller) – Ideal for providing "fine-grained" protection that's standard across a single cluster
- **At the service** (on a lightweight load balancer like NGINX Plus) – Can be a necessary approach when there are a small number of services within a cluster that have a shared need for unique policies
- **At the pod** (as part of the application) – A very custom approach that might be used when the policy is specific to the app

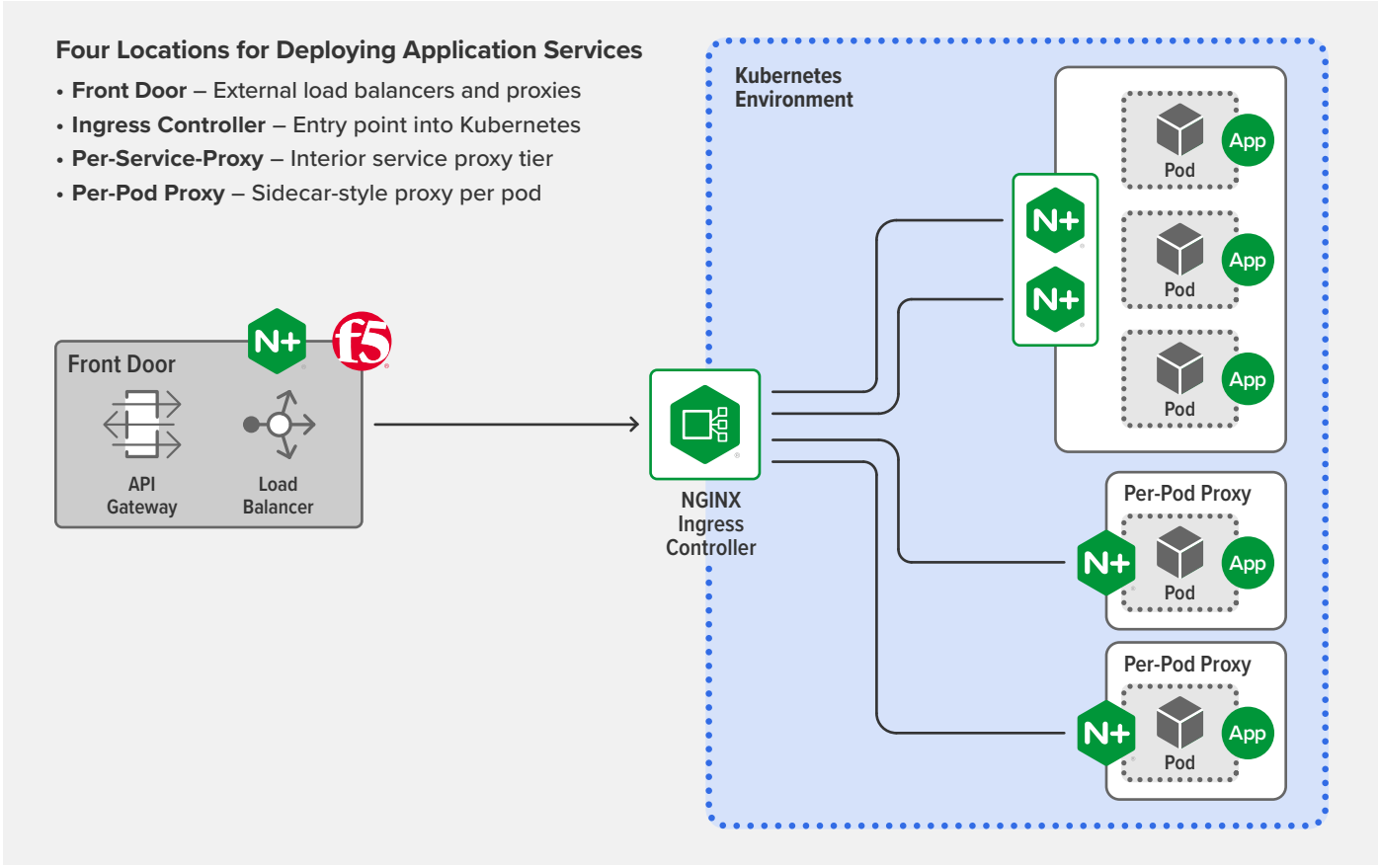


Figure 10: Four Locations for Deploying Application Services

So, out of the four options, which is best? Well . . . that depends!

Where to Deploy a WAF

First, let's look at WAF deployment options since that tends to be a more nuanced choice.

- **Front door and edge** – If your organization prefers a "defense in depth" security strategy, then we recommend deploying a WAF at both the external load balancer and the Ingress controller to deliver an efficient balance of global and custom protections.
- **Front door or edge** – In the absence of a "defense in depth" strategy, a single location is acceptable, and the deployment location depends on ownership. When a traditional NetOps team owns security, they may be more comfortable managing it on a traditional proxy (the external load balancer). However, DevSecOps teams that are comfortable with Kubernetes (and prefer having their security configuration in the vicinity of their cluster configs) may choose to deploy a WAF at the ingress level.
- **Per service or pod** – If your teams have specific requirements for their services or apps, then they can deploy additional WAFs in an à la carte fashion. But be aware: costs are higher for these locations. In addition to increased development time and a higher cloud budget, this choice can also increase operational costs related to troubleshooting efforts, such as when determining "Which of our WAFs is unintentionally blocking traffic?"

Where to Deploy DoS Protection

Protection against DoS attacks is more straightforward since it's only needed at one location – either at the front door or at the Ingress controller. If you deploy a WAF both at the front door and the edge, then we recommend that you deploy DoS protection in front of the front-door WAF, where it's the most global. That way, unwanted traffic can be thinned out before hitting the WAF, allowing you to make more efficient use of compute resources.

For more details on each of these scenarios, read Deploying Application Services in Kubernetes, Part 2 on our blog.

SECURITY USE CASE #3:
Offload Authentication and Authorization from the Apps

Solution: Centralize authentication and authorization at the point of ingress

A common non-functional requirement that gets built into apps and services is authentication and authorization. On a small scale, this practice adds a manageable amount of complexity that's acceptable when the app doesn't require frequent updates. But with faster release velocities at larger scale, integrating authentication and authorization into your apps becomes untenable. Ensuring each app maintains the appropriate access protocols can distract from the business logic of the app, or worse, can get overlooked and lead to an information breach. While use of SSO technologies can improve security by eliminating separate usernames and passwords in favor of one set of credentials, developers still have to include code in their apps to interface with the SSO system.

There's a better way: Offload authentication and authorization to an Ingress controller.

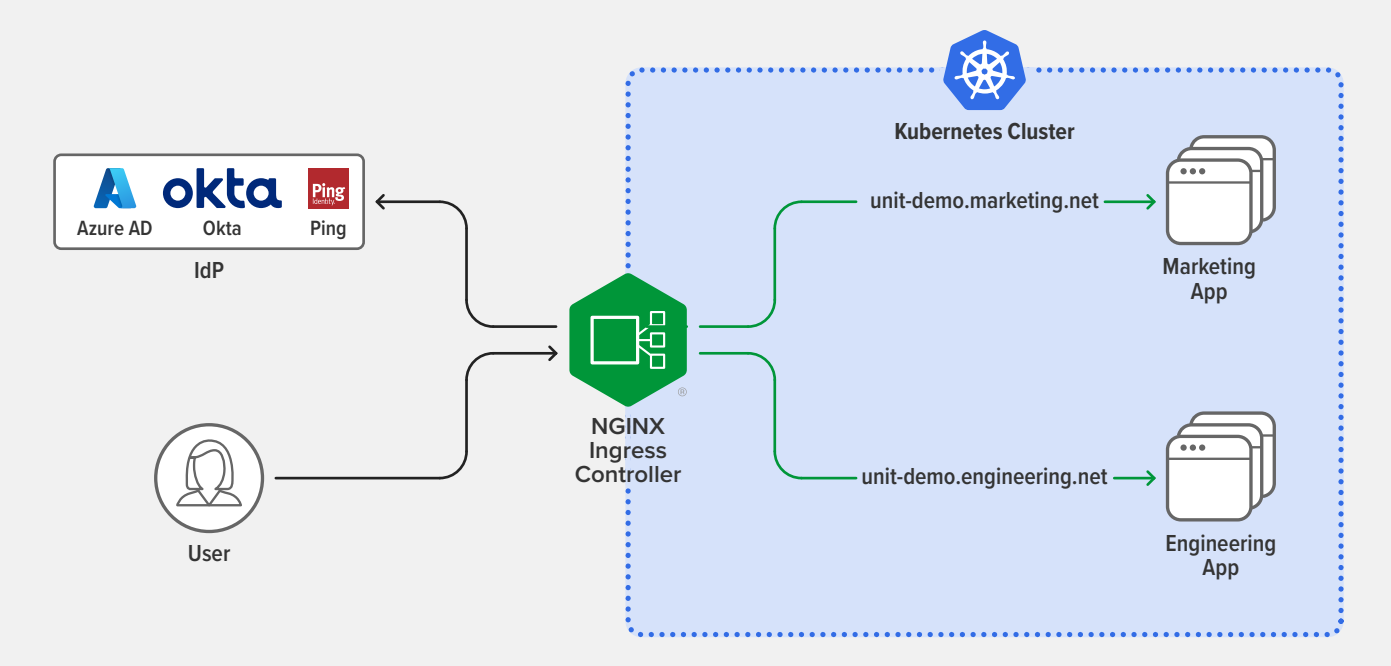


Figure 11: Authenticating and Authorizing with an Ingress Controller

Because the Ingress controller is already scrutinizing all traffic entering the cluster and routing it to the appropriate services, it's an efficient choice for centralized authentication and authorization. This removes the burden from developers of building, maintaining, and replicating the logic in the application code. Instead, they can quickly leverage SSO technologies at the ingress layer using the native Kubernetes API.

For more on this topic, read Implementing OpenID Connect Authentication for Kubernetes with Okta and NGINX Ingress Controller on our blog.

SECURITY USE CASE #4:
Set Up Self-Service with Guardrails

Solution: Implement role-based access control (RBAC)

Kubernetes uses RBAC to control the resources and operations available to different types of users. This is a valuable security measure as it allows an administrator or superuser to determine how users, or groups of users, can interact with any Kubernetes object or specific namespace in the cluster.

USERS GET GATED ACCESS TO THE FUNCTIONALITY THEY NEED TO DO THEIR JOBS WITHOUT WAITING AROUND FOR A TICKET TO BE FULFILLED

While Kubernetes RBAC is enabled by default, you need to take care that your Kubernetes traffic management tools are also RBAC-enabled and can align with your organization's security needs. With RBAC in place, users get gated access to the functionality they need to do their jobs without waiting around for a ticket to be fulfilled. But without RBAC configured, users can gain permissions they don't need or aren't entitled to, which can lead to vulnerabilities if the permissions are misused.

An Ingress controller is a prime example of a tool that can serve numerous people and teams when configured with RBAC. When the Ingress controller allows for fine-grained access management – even down to a single namespace – you can use RBAC to enable efficient use of resources through multi-tenancy.

As an example, multiple teams might use the Ingress controller as follows:

- **NetOps Team** – Configures external entry point of the application (like the hostname and TLS certificates) and delegates traffic control policies to various teams
- **DevOps Team A** – Provisions TCP/UDP load balancing and routing policies
- **DevOps Team B** – Configures rate-limiting policies to protect services from excessive requests
- **Identity Team** – Manages authentication and authorization components while configuring mTLS policies as part of an end-to-end encryption strategy
- **DevSecOps Team** – Sets WAF policies

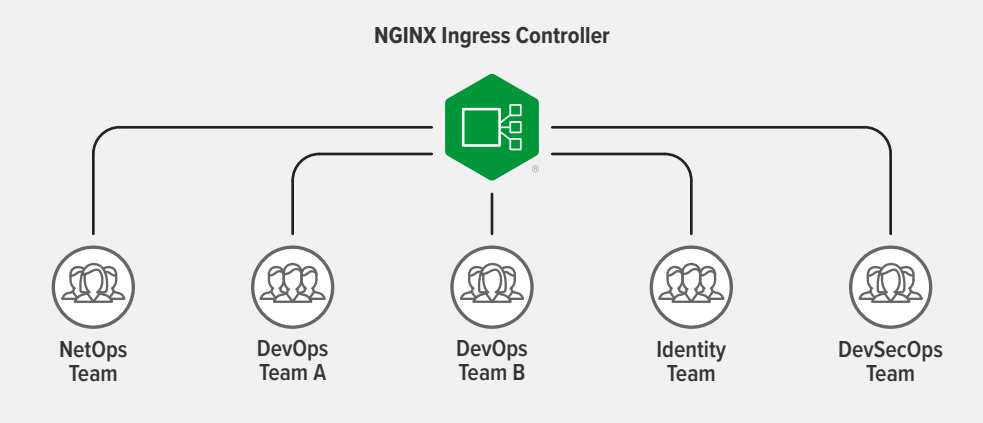


Figure 12: Example of Multiple Teams Administering an Ingress Controller

To learn more about RBAC in NGINX Ingress Controller, watch Advanced Kubernetes Deployments with NGINX Ingress Controller. Starting at 13:50, our experts explain how to leverage RBAC and resource allocation for security, self-service, and multi-tenancy.

SECURITY USE CASE #5:
Implement End-To-End Encryption

Solution: Use traffic management tools

End-to-end encryption (E2EE) is becoming an increasingly common requirement for organizations that handle sensitive or personal information. Whether it's financial data or social media messaging, consumer privacy expectations combined with regulations like GDPR and HIPAA are driving demand for this type of protection. The first step in achieving E2EE is either to architect your backend apps to accept SSL/TLS traffic or to use a tool that offloads SSL/TLS management from your apps (the preferred option for separation of security function, performance, key management, etc.). Then, you configure your traffic management tools depending on the complexity of your environment.

Most Common Scenario: E2EE Using an Ingress Controller

When you have apps with just one endpoint (simple apps, or monolithic apps that you've "lifted and shifted" into Kubernetes) or there's no service-to-service communication, then you can use an Ingress controller to implement E2EE within Kubernetes.

Step 1: Ensure your Ingress controller only allows encrypted SSL/TLS connections using either service-side or mTLS certificates, ideally for both ingress and egress traffic.

Step 2: Address the typical default setting that requires the Ingress controller to decrypt and re-encrypt traffic before sending it to the apps. This can be accomplished in a couple of ways – the method you choose depends on your Ingress controller and requirements:

- If your Ingress controller supports SSL/TLS passthrough, it can route SSL/TLS-encrypted connections based on the Service Name Indication (SNI) header, without decrypting them or requiring access to the SSL/TLS certificates or keys.
- Alternately, you can set up SSL/TLS termination, where the Ingress controller terminates the traffic, then proxies it to the backends or upstreams – either in clear-text or by re-encrypting the traffic with mTLS or service-side SSL/TLS before forwarding it to your Kubernetes services.

Less Common Scenario: E2EE Using an Ingress Controller and Service Mesh

IN E2EE, A SERVICE MESH'S ROLE IS TO ENSURE THAT ONLY SPECIFIC SERVICES ARE ALLOWED TO TALK TO EACH OTHER AND THAT THEY DO SO IN A SECURE MANNER

If there's service-to-service communication within your cluster, you need to implement E2EE on two planes: ingress-egress traffic with the Ingress controller and service-to-service traffic with a service mesh. In E2EE, a service mesh's role is to ensure that only specific services are allowed to talk to each other and that they do so in a secure manner. When you're setting up a service mesh for E2EE, you need to enable a zero-trust environment through two factors: mTLS between services (set to require a certificate) and traffic access control between services (dictating which services are authorized to communicate). Ideally, you also implement mTLS between the applications (managed by a service mesh and the ingress-egress controller) for true E2EE security throughout the Kubernetes cluster.

For more on encrypting data that's been exposed on the wire, read The mTLS Architecture in NGINX Service Mesh on our blog.

SECURITY USE CASE #6: Ensure Clients Are Using a Strong Cipher with a Trusted Implementation

Solution: Comply with the Federal Information Processing Standards (FIPS)

BECOMING FIPS-COMPLIANT CAN BE A SMART MOVE REGARDLESS OF YOUR INDUSTRY OR GEOGRAPHY, BECAUSE FIPS IS ALSO THE DE FACTO GLOBAL CRYPTO-GRAPHIC BASELINE

In the software industry, FIPS usually refers to the publication specifically about cryptography, FIPS 140-2 Security Requirements for Cryptographic Modules, which is a joint effort between the United States and Canada. It standardizes the testing and certification of cryptographic modules that are accepted by the federal agencies of both countries for the protection of sensitive information. "*But wait!*" you say. "*I don't care about FIPS because I don't work with North American government entities.*" Becoming FIPS-compliant can be a smart move regardless of your industry or geography, because FIPS is also the de facto global cryptographic baseline.

Complying with FIPS doesn't have to be difficult, but it does require that both your operating system and relevant traffic management tools can operate in FIPS mode. There's a common misconception that FIPS compliance is achieved simply by running the operating system in FIPS mode. Even with the operating system in FIPS mode, it's still possible that clients communicating with your Ingress controller aren't using a strong cipher. When operating in FIPS mode, your operating system and Ingress controller may use only a subset of the typical SSL/TLS ciphers.

Setting up FIPS for your Kubernetes deployments is a four-step process:

Step 1: Configure your operating system for FIPS mode

Step 2: Verify the operating system and OpenSSL are in FIPS mode

Step 3: Install the Ingress controller

Step 4: Verify compliance with FIPS 140-2 by performing a FIPS status check

In the example below, when FIPS mode is enabled for both the operating system and the OpenSSL implementation used by NGINX Ingress Controller, all end-user traffic to and from NGINX Ingress Controller is decrypted and encrypted using a validated, FIPS-enabled crypto engine.

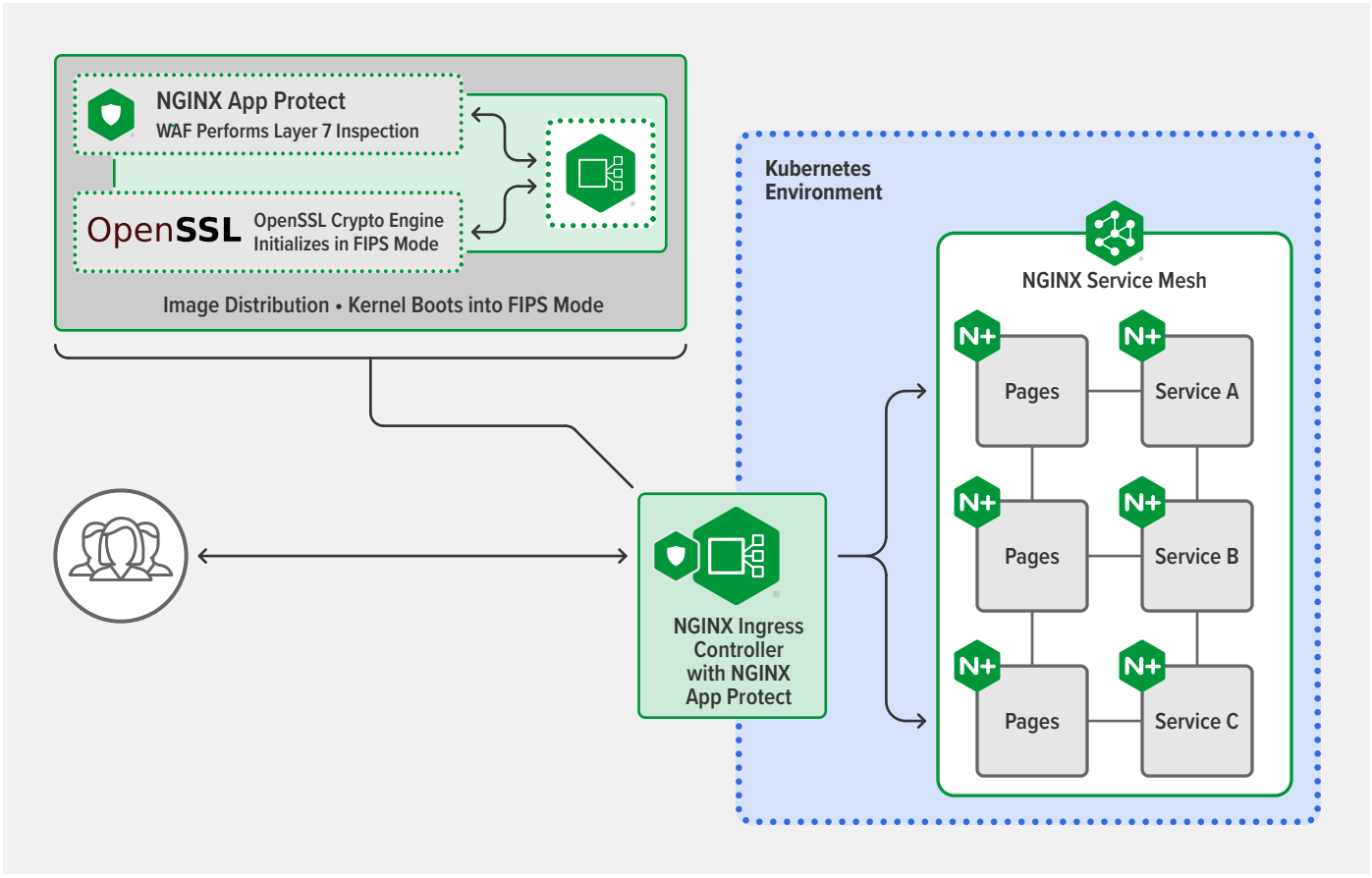


Figure 13: FIPS in a Kubernetes Environment

Read more about FIPS in Achieving FIPS Compliance with NGINX Plus on our blog.

Improve Kubernetes Resilience, Visibility, and Security with NGINX

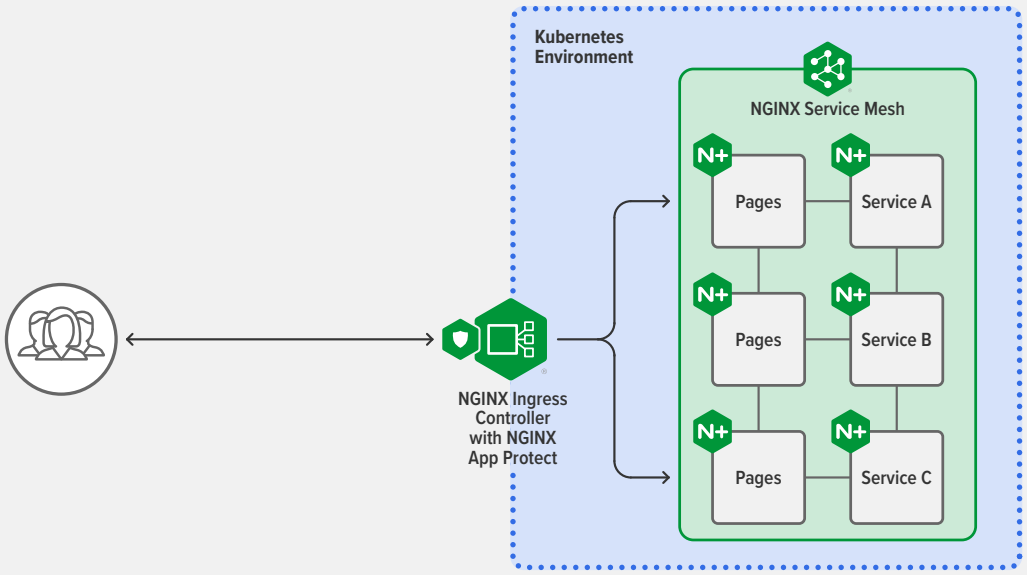
If you're ready to implement some (or all) of the methods discussed in this eBook, you need to make sure your tools have the right features and capabilities to support your use cases. NGINX can help with our suite of production-grade Kubernetes traffic management tools:

NGINX Ingress Controller – NGINX Plus-based Ingress controller for Kubernetes that handles advanced traffic control and shaping, monitoring and visibility, authentication and SSO, and acts as an API gateway

NGINX App Protect – Holistic protection for modern apps and APIs, built on F5's market-leading security technologies, that integrates with NGINX Ingress Controller and NGINX Plus. Use a modular approach for flexibility in deployment scenarios and optimal resource utilization:

- **NGINX App Protect WAF** – A strong, lightweight WAF that protects against OWASP Top 10 and beyond with PCI DDS compliance
- **NGINX App Protect DoS** – Behavioral DoS detection and mitigation with consistent and adaptive protection across clouds and architectures

NGINX Service Mesh – Lightweight, turnkey, and developer-friendly service mesh featuring NGINX Plus as an enterprise sidecar



Get started by requesting your free 30-day trial of NGINX Ingress Controller with NGINX App Protect WAF and DoS, and download the always-free NGINX Service Mesh. Learn more today at nginx.com.

PART TWO

Choose the Best Kubernetes Traffic Managment Tools for Your Needs

5. How to Choose an Ingress Controller - Identify Your Requirements

AS YOU SCALE, YOUR CHOICE OF INGRESS CONTROLLER BECOMES MORE IMPORTANT

When organizations first start experimenting with Kubernetes, they often don't put a lot of thought into the choice of Ingress controller. They might think that all Ingress controllers are alike, and in the interests of getting up and running quickly it's easiest to stick with the default Ingress controller for the Kubernetes framework they did choose. And it's true that just about any Ingress controller is fine in testing or low-volume production environments. But as you scale, your choice of Ingress controller becomes more important. That's because Ingress controllers can provide much more than the basic functionality of moving your traffic from point A to point B.

From advanced traffic management to visibility to built-in security, an Ingress controller can be one of the most powerful tools in your Kubernetes stack. In fact, at F5 NGINX we consider it to be the foundation of any production-grade Kubernetes deployment. But many developers and Platform Ops teams don't realize the full power of an Ingress controller – or the consequences of choosing one that can't scale. Choosing an Ingress controller that doesn't scale well or protect complex environments can prevent you from getting Kubernetes out of testing and into production. In this second part of the eBook, we aim to educate you on the basics of Ingress controllers and how to make a wise choice that delivers the functionality and security you need – today and tomorrow.

WHAT'S AN INGRESS CONTROLLER?

An Ingress controller is a specialized load balancer that manages the Layer 4 and Layer 7 traffic entering Kubernetes clusters, and potentially the traffic exiting them. So that we're all on the same page, here are the terms we use at NGINX (and they're largely the same across the industry):

- **Ingress traffic** – Traffic entering a Kubernetes cluster
- **Egress traffic** – Traffic exiting a Kubernetes cluster
- **North-south traffic** – Traffic entering and exiting a Kubernetes cluster (also called *ingress-egress* traffic)
- **East-west traffic** – Traffic moving among services within a Kubernetes cluster (also called *service-to-service* traffic)
- **Service mesh** – A traffic management tool for routing and securing service-to-service traffic

WHY DO YOU NEED AN INGRESS CONTROLLER?

BY DEFAULT, APPLICATIONS RUNNING IN KUBERNETES PODS (CONTAINERS) ARE NOT ACCESSIBLE FROM THE EXTERNAL NETWORK

By default, applications running in Kubernetes pods (containers) are not accessible from the external network, but only from other pods within the Kubernetes cluster. Kubernetes has a built-in configuration object for HTTP load balancing, called *Ingress*, which defines how entities outside a Kubernetes cluster can connect to the pods represented by one or more Kubernetes services.

When you need to provide external access to your Kubernetes services, you create an Ingress resource to define the connectivity rules, including the URI path, backing service name, and other information. On its own, however, the Ingress resource doesn't do anything. You must deploy and configure an Ingress controller application (using the Kubernetes API) to implement the rules defined in Ingress resources.

WHAT DO INGRESS CONTROLLERS DO?

- Accept traffic from outside the Kubernetes environment, potentially modify (shape) it, and distribute it to pods running inside the environment. The Ingress controller replaces the default `kube-proxy` traffic distribution model, giving you additional controls like those that application delivery controllers (ADCs) and reverse proxies provide in non-Kubernetes environments.
- Monitor the individual pods of a service, guaranteeing intelligent routing and preventing requests from being "black holed".
- Implement egress rules to enhance security with mutual TLS (mTLS) or limit outgoing traffic from certain pods to specific external services.

What Does An Ingress Controller Do?

Monitoring and Visibility

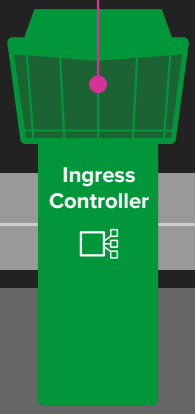
The **Ingress controller** can give you insight into issues impacting app and infrastructure performance, and help you predict when traffic surges will strike.

The **Ingress controller** is a specialized load balancer that manages Layer 4 and Layer 7 ingress and egress ("north-south") traffic.

It can also be used for:
- Traffic control
- Traffic shaping
- Monitoring and visibility
- Routing API traffic as an API gateway
- Authentication and SSO
- WAF integration

Security

The **Ingress controller** can protect your environment from unauthorized or malicious traffic via centralized authentication, SSO, and as the ideal point for a WAF.

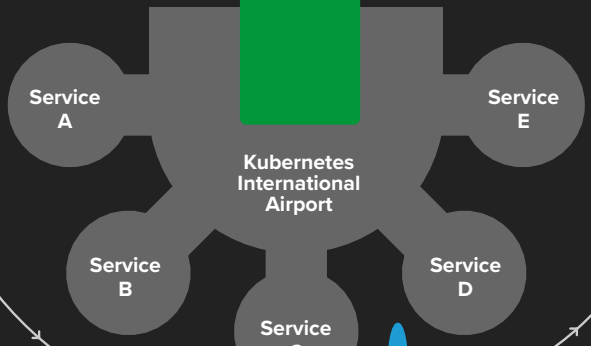


Ingress traffic is traffic entering a Kubernetes cluster.

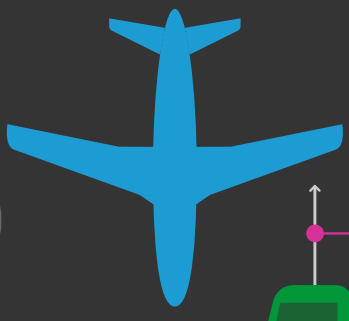
The **Ingress controller** accepts ingress traffic, potentially modifies (shapes) it, and distributes it to pods running inside the environment.

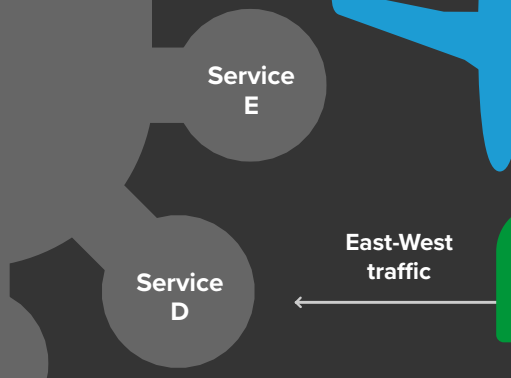
Egress traffic is traffic exiting a Kubernetes cluster.

The **Ingress controller** implements egress rules to enhance security with mTLS or limits outgoing traffic from certain pods to specific external services.



The **Ingress controller** monitors the individual pods of a **service**, guaranteeing intelligent routing and preventing requests from being "black-holed."





East-west (service-to-service) traffic is traffic moving among services within a Kubernetes cluster.

An **Ingress controller** cannot manage east-west traffic.

When your app and infrastructure reach a level of maturity where this traffic needs to be managed, you need a **service mesh**.

East-West traffic

A **service mesh** routes and secures east-west traffic.

It is used to implement:

- End-to-end encryption and mTLS
- Orchestration
- Management of service traffic
- Monitoring and visibility

When it's time to select an Ingress controller, it can be tempting to start with a feature list, but you might end up with an Ingress controller that has fantastic features but doesn't satisfy your business needs. Instead, make sure to explore two elements that impact how well the Ingress controller will work for your team and your apps: use cases (what problems it will solve) and resourcing (how you're going to "pay" for it). We cover these two topics in the remainder of this chapter.

WHAT PROBLEMS DO YOU WANT THE INGRESS CONTROLLER TO SOLVE?

The core use case for an Ingress controller is traffic management, so you probably want the Ingress controller to handle one or more of these common use cases:

- Load balancing (HTTP2, HTTP/HTTPS, SSL/TLS termination, TCP/UDP, WebSocket, gRPC)
- Traffic control (rate limiting, circuit breaking, active health checks)
- Traffic splitting (debug routing, A/B testing, canary deployments, blue-green deployments)

THERE'S NO REASON TO SETTLE FOR A "ONE-TRICK PONY" – MOST INGRESS CONTROLLERS CAN DO MORE THAN MANAGE TRAFFIC

But there's no reason to settle for a "one-trick pony" – most Ingress controllers can do more than manage traffic. By using the Ingress controller to solve multiple problems, not only do you reduce the size and complexity of your stack, but you can also offload non-functional requirements from the apps to the Ingress controller. Let's look at four non-traditional Ingress controller use cases that can help make your Kubernetes deployments more secure, agile, and scalable while making more efficient use of your resources.

Monitoring and Visibility

Lack of visibility into the Kubernetes cluster is one of the biggest challenges in production environments, contributing to difficulty with troubleshooting and resilience. Because Ingress controllers operate at the edge of your Kubernetes clusters and touch every bit of traffic, they're well situated to provide data that can help you troubleshoot (and even avoid) two common problems: slow apps and resource exhaustion in the Kubernetes cluster or platform. For an Ingress controller to improve visibility, it needs to:

- Provide metrics in real time so you can diagnose what's happening "right now"
- Be able to export metrics to popular visibility tools, like Prometheus and Grafana, that plot values over time to help you predict traffic surges and other trends

API Gateway

Unless you're looking to perform request-response manipulation in Kubernetes, it's very likely that the Ingress controller can double as your API gateway. Depending on its feature set, an Ingress controller might be able to provide core API gateway functions including TLS termination, client authentication, rate limiting, fine-grained access control, and request routing at Layers 4 through 7.

Authentication and Single Sign-On

Offloading authentication of login credentials from your Kubernetes services to your Ingress controller can solve two issues:

- Enable users to log into multiple apps with a single set of credentials by implementing single sign-on (SSO) with OpenID Connect (OIDC)
- Eliminate the need to build authentication functionality into each application, allowing your developers to focus on the business logic of their apps

Web Application Firewall Integration

It's not that an Ingress controller can serve as a web application firewall (WAF), but rather that the WAF can be consolidated with the Ingress controller. Although a WAF can be deployed at many places outside and within Kubernetes, for most organizations the most efficient and effective location is in the same pod as the Ingress controller. This use case is perfect when security policies are under the direction of SecOps or DevSecOps and when a fine-grained per-service or per-URI approach is needed. It means you can use the Kubernetes API to define policies and associate them with services. Further, the Ingress controller's role-based access control (RBAC) functionality can enable SecOps to enforce policies per listener while DevOps users set policies per Ingress resource.

HOW ARE YOU GOING TO RESOURCE THE INGRESS CONTROLLER?

PERHAPS YOU'VE HEARD THE PHRASE "FREE LIKE A PUPPY"

Every Ingress controller comes at a cost . . . even those that are free and open source (perhaps you've heard the phrase "free like a puppy"). Some costs can be assigned predictable dollar amounts as line items in your budget, while others depend on how much time your team has to spend dealing with the consequences of which Ingress controller you choose (increased complexity, lack of portability, and so on). Let's look at the primary costs to consider when budgeting for an Ingress controller: time and money.

Budgeting for Time Costs

Budgeting for headcount can be far more challenging than for fixed-cost line items, especially when you're trying to resource a project in an unfamiliar space. You need to ask questions like:

- Who will configure and administer the Ingress controller? Is it part of their full-time job (for example, as members of your Platform Ops team) or are they taking it on as an extra responsibility (as developers)?
- Can you make time for employees to take formal training? Or must the tool be simple enough to learn quickly and easily on the job?
- Are you prepared to have employees tinker with it whenever a new feature is needed, or spend extensive time troubleshooting when there's a problem? Or do you need them to deliver other business value?

A Note on Kubernetes Tool Ownership

We've observed a trend in the industry toward consolidating tools and ownership under the domain of NetOps teams. While this can go a long way towards simplifying your stack and improving security, we advocate for thoughtful duplication of tools based on team goals. It makes sense to have NetOps team manage perimeter tools (like external load balancers) because they focus on the broader platform, but DevOps teams care most about the services deployed close to the app code and need more agility and finer-grained control than they get from NetOps tools. Kubernetes tools, including the Ingress controller, have the best chance of success when they're selected by DevOps. That's not to say you must grant developers complete freedom of choice for tools! Some strict standardization of tooling within Kubernetes is still a best practice.

Budgeting for Capital Costs

When organizations first experiment with Kubernetes, they often don't budget much for tools or support. If you have the people resources to truly support an Ingress controller that needs more "hand holding," then no monetary budget is okay . . . at first. But as your Kubernetes investment increases, you may find yourself limited by the tool's features, or wanting to dedicate your team to other priorities. That's when the scale tips towards paying for an easier-to-use, more stable tool with enterprise features and support.

When you're ready to pay for an Ingress controller, be aware that the licensing model matters. Based on traditional pricing models outside of Kubernetes, pricing for Ingress controllers is often "per instance" or "per Ingress proxy." While there are use cases when this still makes sense in Kubernetes, licensing an Ingress controller on a "per cluster" basis instead means you pay based on application tenancy rather than "number of proxies".

Here are our recommendations for different scenarios:

- **New to Kubernetes? Choose per-cluster licensing.**
  When you're inexperienced with Kubernetes, it's very difficult to accurately predict the number of Ingress controller instances you need. If forced to choose a number of instances, you may underestimate – making it difficult to achieve your goals – or overestimate and waste money better spent on other parts of the Kubernetes project. Negotiating a relatively low fixed price for a "small cluster" increases your chance for success.
- **Scaling Kubernetes? Choose per-cluster licensing.**
  It's nearly impossible to predict how much and how often you'll need to scale Kubernetes resources up and down (bursting and collapsing). Per-instance pricing forces your team to impose arbitrary thresholds to stay within licensing caps. With per-cluster licensing, you don't have to track individual Ingress containers and, depending on the vendor (such as NGINX), bursting may be included at no additional cost.
- **Advanced or static deployments? Choose per-instance licensing.**
  If you're savvy enough with Kubernetes to know exactly how you're going to use the Ingress controller, and you plan to use a small number of Ingress proxies per cluster (typically fewer than three), and you don't need any bundled services that might come along with the tool, then per-instance pricing can be a great choice.

NEXT STEP: RISK TOLERANCE AND FUTURE-PROOFING

Now that you have a grasp on your requirements, the next step is to decide as a team what your tolerance is for the risks an Ingress controller might introduce, and to figure out how you'll need the Ingress controller to scale as you grow your Kubernetes deployment. We take up those topics in the next chapter.

6. How to Choose an Ingress Controller - Risks and Future-Proofing

Welcome to the second step of choosing a Ingress controller. At this point, you've identified your requirements – but it's not yet time to test products! In this chapter, we explain how the wrong Ingress controller (for your needs) can slow your release velocity and even cost you customers. As with any tool, Ingress controllers can introduce risks and impact future scalability. Let's look at how to eliminate choices that might cause more harm than good.

INGRESS CONTROLLER RISKS

TODAY'S CONSUMERS ARE FICKLE, AND ANYTHING THAT CAUSES A POOR DIGITAL EXPERIENCE MAY BE UNACCEPTABLE DESPITE A COMPELLING FEATURE SET

There are three specific risks you should consider when introducing a traffic management tool for Kubernetes: complexity, latency, and security. These issues are often intertwined; when one is present, it's likely you'll see the others. Each can be introduced by an Ingress controller and it's up to your organization to decide if the risk is tolerable. Today's consumers are fickle, and anything that causes a poor digital experience may be unacceptable despite a compelling feature set.

Ingress Controller Risks

New tools can introduce risks that might outweigh the rewards. Here are the top three risks that can be introduced by an Ingress controller that doesn't align to your needs.

01

Complexity

Does it Defeat the Purpose of a Microservices Architecture?

Complexity is one of the top challenges in using and deploying containers.[1]

The wrong Ingress controller can add even more complexity – limiting your ability to scale the deployment horizontally and negatively impacting app performance.

02

Latency

Does the Ingress Controller Slow Down Your Apps?

Organizations adopt Kubernetes for the ability to deploy new apps more quickly.[2]

But an Ingress controller that adds latency through errors, timeouts, and reloads can slow down your apps.

03

Security

Does the Ingress Controller Open the Door for Hackers?

More than half of organizations have delayed or slowed down application deployment into production due to container or Kubernetes security concerns.[3]

Watch out for Ingress controllers with slow CVE patching and beware of relying on support from public forums.

1. CNCF Survey 2020
2. 2021 Kubernetes Adoption Survey
3. Red Hat State of Kubernetes Security Report

Complexity

Does It Defeat the Purpose of a Microservices Architecture?

The best Kubernetes tools are those that meet the goals of microservices architecture: lightweight and simple in design. It's possible to develop a very feature-rich Ingress controller that sticks to these principles, but that's not always the norm. Some designers include too many functions or use convoluted scripting to tack on capabilities that aren't native to the underlying engine, resulting in an Ingress controller that's needlessly complex.

YOU CAN TYPICALLY SPOT AN OVERLY COMPLEX INGRESS CONTROLLER BY ITS SIZE: THE LARGER THE FOOTPRINT, THE MORE COMPLEX THE TOOL

And why does that matter? In Kubernetes, an overly complex tool can negatively impact app performance and limit your ability to scale your deployment horizontally. You can typically spot an overly complex Ingress controller by its size: the larger the footprint, the more complex the tool.

Latency

Does It Slow Down Your Apps?

Ingress controllers can add latency due to resource usage, errors, and timeouts. Look at latency added in both static and dynamic deployments and eliminate options that introduce unacceptable latency based on your internal requirements. For more details on how reconfigurations can impact app speed, see Performance Testing Three Different NGINX Ingress Controller Options in the appendix.

Security

Does It Open the Door for Hackers?

Common Vulnerabilities and Exposures (CVEs) are rampant on today's Internet, and the time it takes for your Ingress controller provider to furnish a CVE patch can be the difference between safety and a breach. Based on your organization's risk tolerance, you may want to eliminate solutions that take more than a few days (or at most weeks) to provide patches.

Beyond CVEs, some Ingress controllers expose you to another potential vulnerability. Consider this scenario: you work for an online retailer and need help troubleshooting the configuration of your open source Ingress controller. Commercial support isn't available, so you post the issue to a forum like Stack Overflow. Someone offers to help and wants to look for problems in the config and log files for the Ingress controller and other Kubernetes components. Feeling the pressure to get the problem resolved quickly, you share the files.

ONE OF THE TOP REASONS ORGANIZATIONS CHOOSE TO PAY FOR SUPPORT: IT GUARANTEES CONFIDENTIALITY

The "good Samaritan" helps you solve your problem, but six months later you discover a breach – credit card numbers have been stolen from your customer records. Oops. Turns out the files you shared included information that was used to infiltrate your app. This scenario illustrates one of the top reasons organizations choose to pay for support: it guarantees confidentiality.

A Note on OpenResty-Based Ingress Controllers

OpenResty is a web platform built on NGINX Open Source that incorporates LuaJIT, Lua scripts, and third-party NGINX modules to extend the functionality in NGINX Open Source. In turn, there are several Ingress controllers built on OpenResty, which we believe could potentially add two risks compared to our Ingress controllers based on NGINX Open Source and F5 NGINX Plus.

- **Timeouts** – As noted, OpenResty uses Lua scripting to implement advanced features like those in our commercial NGINX Plus-based Ingress Controller. One such feature is dynamic reconfiguration, which eliminates an NGINX Open Source requirement that reduces availability – namely, that the NGINX configuration must be reloaded when service endpoints change. To accomplish dynamic reconfiguration with OpenResty, the Lua handler chooses which upstream service to route the request to, thereby eliminating the need to reload the NGINX configuration.

  LUA MUST CONTINUOUSLY CHECK FOR CHANGES TO THE BACKENDS, WHICH CONSUMES RESOURCES

  However, Lua must continuously check for changes to the backends, which consumes resources. Incoming requests take longer to process, causing some of the requests to get stalled, which increases the likelihood of timeouts. As you scale to more users and services, the gap between the number of incoming requests per second and the number that Lua can handle widens exponentially. The consequence is latency, complexity, and higher costs.

  To find out how much latency Lua can add, see see Performance Testing Three Different NGINX Ingress Controller Options in the appendix.

- **CVE patching delays** – Compared to the Ingress Controllers from NGINX, patches for CVEs inevitably take longer to show up in Ingress controllers based on tools like OpenResty that are in turn based on NGINX Open Source. As we outline in detail in Mitigating Security Vulnerabilities Quickly and Easily with NGINX Plus on our blog, when a CVE in NGINX is discovered, we as the vendor are generally informed before the CVE is publicly disclosed. That enables us to release a patch for NGINX Open Source and NGINX Plus as soon as the CVE is announced.

  Technologies based on NGINX Open Source might not learn about the CVE until that point, and in our experience OpenResty patches lag behind ours by a significant amount – four months in one recent case. Patches for an Ingress controller based on OpenResty inevitably take yet more time, giving a bad actor ample opportunity to exploit the vulnerability.

FUTURE-PROOF YOUR INGRESS CONTROLLER

Even if you're just starting to dabble in Kubernetes, there's a good chance you aspire to put it into production someday. There are four main areas where your needs are likely to grow over time: infrastructure, security, support, and multi-tenancy.

Future-Proof Your Ingress Controller

There are four main areas where your needs are likely to grow over time.

01

Infrastructure

Will You Use Kubernetes in Hybrid- or Multi-Cloud Environments?

It's rare for an organization to be fully and permanently committed to one type of environment. Choose an infrastructure-agnostic Ingress controller from the start, allowing you to use the same tool across all your environments.

02

Security

How Will You Secure Kubernetes from the Inside?

Kubernetes apps are best protected when security – including authentication and authorization – is close to the apps. Centralizing security (authentication, authorization, DoS protection, web application firewall) at the point of Ingress makes a lot of sense from the standpoint of both cost and efficiency.

03

Support

How "On Your Own" Can You Afford to Be?

Using workarounds and waiting on community support is okay when when you're running small deployments but it's not sustainable when you move to production. Choose an Ingress controller that allows you to add support in the future – or has an inexpensive support tier that can be upgraded as you scale.

04

Multi-Tenancy

How Can Multiple Teams and Apps Share a Container Environment Safely and Securely?

When your services and teams grow in size and complexity, you'll probably turn to multi-tenancy to achieve maximum efficiency. Some Ingress controllers can help you carve up those clusters through a number of features and concepts: multiple ingresses, classes, namespaces, and scoped resources that support RBAC.

Infrastructure

Will You Use Kubernetes in Hybrid- or Multi-Cloud Environments?

It's rare for an organization to be fully and permanently committed to one type of environment. More commonly, organizations have a mix of on-premises and cloud, which can include private, public, hybrid-cloud, and multi-cloud. (For a deeper dive into how these environments differ, read What Is the Difference Between Multi-Cloud and Hybrid-Cloud?)

As we mentioned in chapter 5, it's tempting to choose tools that come default with each environment, but there are a host of problems specific to default Ingress controllers. We cover all the cons in chapter 7, but the issue that's most relevant to future-proofing is vendor lock-in – you can't use a cloud-specific Ingress controller across all your environments. Use of default cloud-specific tooling impacts your ability to scale because you're left with just two unappealing alternatives:

1. Try to make the existing cloud work for all your needs
2. Rewrite all your configurations – not just load balancing but security as well! – for the Ingress controller in each new environment

CHOOSE AN INFRASTRUCTURE-AGNOSTIC INGRESS CONTROLLER FROM THE START

While the first alternative often isn't viable for business reasons, the second is also tricky because it causes tool sprawl, opens new security vulnerabilities, and requires your employees to climb steep learning curves. The third, and most efficient, alternative is to choose an infrastructure-agnostic Ingress controller from the start, allowing you to use the same tool across all your environments.

When it comes to infrastructure, there's another element to consider: certifications. Let's use the example of Red Hat OpenShift Container Platform (OCP). If you're an OCP user, then you're probably already aware that the Red Hat Marketplace offers certified operators for OCP, including the NGINX Ingress Operator. Red Hat's certification standards mean you get peace of mind knowing that the tool works with your deployment and can even include joint support from Red Hat and the vendor. Lots of organizations have requirements to use certified tools for security and stability reasons, so even if you're only in testing right now, it pays to keep your company's requirements for production environments in mind.

Security

How Will You Secure Kubernetes from the Inside?

KUBERNETES APPS ARE BEST PROTECTED WHEN SECURITY IS CLOSE TO THE APPS

Gone are the days when perimeter security alone was enough to keep apps and customers safe. Kubernetes apps are best protected when security – including authentication and authorization – is close to the apps. And with organizations increasingly mandating end-to-end encryption and adopting a zero-trust network model within Kubernetes, a service mesh might be in your future.

AN INGRESS CONTROLLER THAT ALIGNS WITH YOUR SECURITY STRATEGY CAN PREVENT BIG HEADACHES THROUGHOUT YOUR APP DEVELOPMENT JOURNEY

What does all this have to do with your Ingress controller? A lot! Centralizing security (authentication, authorization, DoS protection, web application firewall) at the point of Ingress makes a lot of sense from the standpoint of both cost and efficiency. And while most service meshes can be integrated with most Ingress controllers, how they integrate matters a lot. An Ingress controller that aligns with your security strategy can prevent big headaches throughout your app development journey.

Read Secure Cloud Native Apps Without Losing Speed for more details on the risks of cloud-native app delivery and Deploying Application Services in Kubernetes, Part 2 to learn more about the factors that determine the best location for security tools.

Support

How "On Your Own" Can You Afford to Be?

When teams are just experimenting with Kubernetes, support – whether from the community or a company – often isn't the highest priority. This is okay if your teams have a lot of time to come up with their own solutions and workarounds, but it's not sustainable when you move to production. Even if you don't need support today, it can be wise to choose an Ingress controller that allows you to add support in the future – or has an inexpensive support tier that can be upgraded as you scale.

Multi-Tenancy

How Can Multiple Teams and Apps Share a Container Environment Safely and Securely?

In the beginning, there was one team and one app. . . Isn't that how every story starts? The story often continues with that one team successfully developing its one Kubernetes app, leading the organization to run more services on Kubernetes. And of course, more services = more teams = more complexity.

To achieve maximum efficiency, organizations adopt multi-tenancy and embrace a Kubernetes model that supports the access and isolation mandated by their business processes while also providing the sanity and controls their operators need. Some Ingress controllers can help you carve up those clusters through a number of features and concepts: multiple ingresses, classes, namespaces, and scoped resources that support role-based access control (RBAC).

Next Step: Narrow Down Options

Now that you've thought about your requirements, risk tolerance, and future-proofing, you have enough information to start narrowing down the very wide field of Ingress controllers. Breaking that field down by category can help you make quick work of this step. In chapter 7, we explore three different categories of Ingress controllers, including the pros and cons of each.

7. How to Choose an Ingress Controller – Open Source vs. Default vs. Commercial

Congratulations! After reading chapters 5 and 6, you're almost ready to select an Ingress controller. Let's recap where we've been so far:

- In chapter 5, we discussed how to identify your requirements, including performance, budget, use cases, architecture, and ownership.
- In chapter 6, we shared some risks that you might introduce by selecting the wrong Ingress controller for your needs, and outlined key areas where you can future-proof your selection.

Ingress controllers fall into three categories: open source, default, and commercial. Each has its use cases, and it's important to be clear on your short- and long-term needs before making a selection. In this chapter, we cover the pros and cons of each category.

OPEN SOURCE INGRESS CONTROLLERS

Many open source Ingress controllers are maintained by a community of users and volunteer developers, though some also have dedicated engineering teams. Two of the most popular open source Ingress controllers are based on NGINX – one is maintained by the Kubernetes community and the other is led by the core NGINX engineering team and open sourced. For further comparison of NGINX-based Ingress controllers, see chapter 8.

Open Source Ingress Controllers

Maintained by a community of users and volunteer developers, though some also have dedicated engineering teams.

Pros

Top reasons an open source Ingress controller could be right for you.

- ▲ No monetary investment (Free!)
- ▲ Community-driven
- ▲ High feature velocity

Ideal when . . .
You're just getting started in Kubernetes, in testing, or low-volume production.

Cons

Top reasons an open source Ingress controller could be wrong for you.

- ▼ Costs more of your time
- ▼ Risk of instability, insecurity, and unreliability
- ▼ Minimal or no support

Consider "default" or "commercial" options to mitigate these cons.

Open Source Ingress Controllers: Pros and Cons

- **Pros:**
  - **Free and community-driven** – Many people and organizations choose open source projects not only because of the unbeatable price (free!), but also because they prefer community-developed tech.
  - **Feature velocity** – These Ingress controllers are more likely to be on the cutting edge of feature innovation.

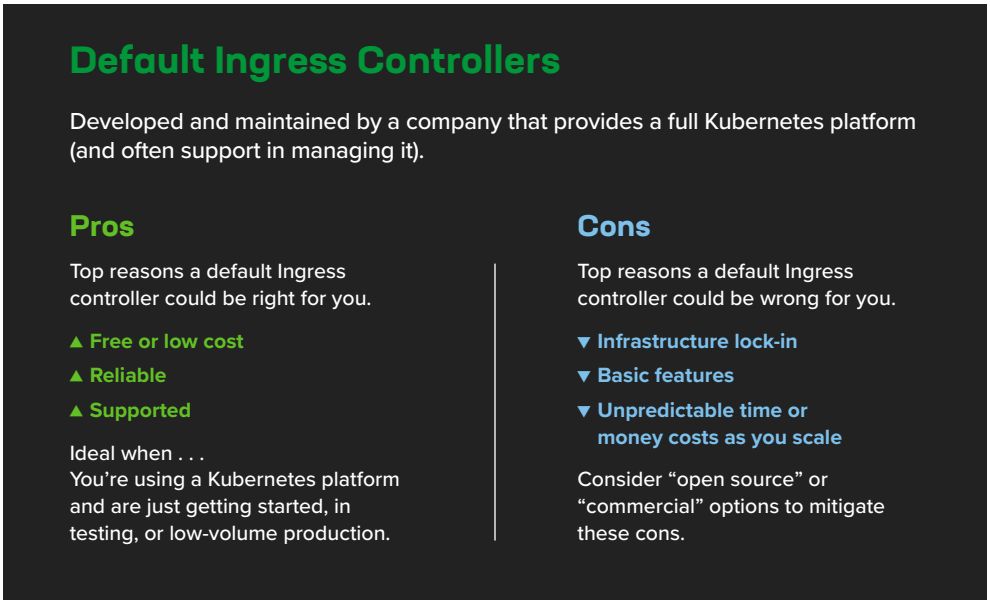
- **Cons** (shared with open source projects in general):
  - **Cost (time)** – They lack "out-of-the-box" tooling for easy setup and scalability, so you end up spending time on customizations and workarounds for your specific needs.
  - **Risky** – There can be issues with stability, security, and reliability (due to the emphasis on feature velocity and the volunteer nature of contributors). Patches for Common Vulnerabilities and Exposures (CVEs) may never come, or might arrive months after the CVE is publicly disclosed, giving hackers plenty of time to attack your Ingress controller.
  - **Minimal or no support** – Most are "self solve" . . . it's just you and the docs. If you run into problems you can't solve yourself, it can be difficult (or impossible) to get help – just about your only choice is to post your problem on community forums and hope other members of the community (a) bother to respond and (b) know of a solution.

IT'S JUST YOU AND THE DOCS

- **Summary:** When organizations first start experimenting with Kubernetes, they often choose an open source Ingress controller, for convenience or because the docs promise you can get up and running quickly for free. This can work beautifully when you're getting started, in testing, or running low-volume production.

DEFAULT INGRESS CONTROLLERS

Although many default Ingress controllers are based on open source technology, we categorize them separately because they're developed and maintained by a company that provides a full Kubernetes platform (and often support in managing it). Examples from this category include public cloud Ingress controllers, Rancher, and Red Hat OpenShift router.



Default Ingress Controllers: Pros and Cons

- **Pros:**
  - **Free or low cost** – The low price tag is a compelling reason to use these products. They're already integrated into the platform, a definite time-saver when you're first getting started.
  - **Reliable and supported** – Because they're maintained by a dedicated engineering team, they may be more reliable than community-maintained Ingress controllers. Commercial support is typically included or available at an extra cost.

- **Cons:**
  - **Infrastructure lock-in** – Default ingress controllers are not infrastructure-agnostic, so you can't take them or your configurations from cloud to cloud. That means you need different Ingress controllers for each deployment environment, which causes tool sprawl, increases the learning curve for your teams, and makes the Ingress controller more difficult to secure.
  - **Basic features** – They usually lack the advanced traffic management and security capabilities necessary for large-scale deployments.

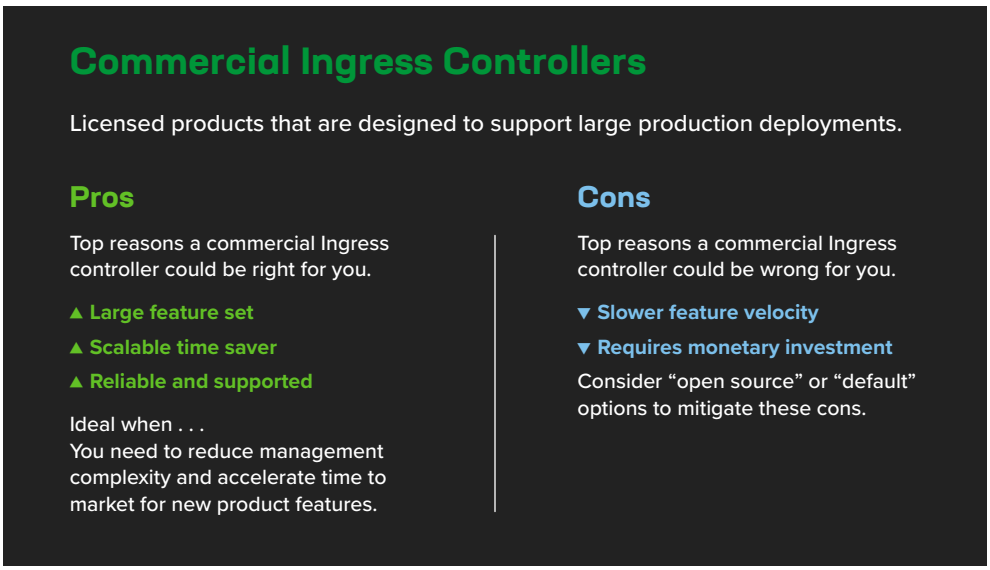
YOU CAN'T TAKE DEFAULT INGRESS CONTROLLERS OR YOUR CONFIGURATIONS FROM CLOUD TO CLOUD

- **Unpredictable costs (time and money)** – While initial costs are nil or low, they can increase dramatically and unpredictably as your application grows. This can take the form of time required to build functionality into your app that's missing from the Ingress controller's minimal feature set – and of course you have to regression-test that functionality every time you update the app. Another drawback of some default tools is huge jumps in your cloud bill as your app becomes more popular, due to throughput charges that seem innocuous at first.

- **Summary:** A default Ingress controller is a popular choice for teams that are newer to Kubernetes and using a managed platform such as Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), Microsoft Azure Kubernetes Service (AKS), Rancher, and Red Hat OpenShift Container Platform. As their apps mature and teams grow, organizations often choose to add an enterprise-grade Ingress controller to their stack, rather than replacing the default tool.

COMMERCIAL INGRESS CONTROLLERS

These Ingress controllers are licensed products that are designed to support large production deployments. One example is the NGINX Plus-based version of F5 NGINX Ingress Controller, which we discuss more in chapter 8.



Commercial Ingress Controllers: Pros and Cons

- **Pros:**
  - **Large feature set** – Commercial Ingress controllers include robust feature sets that support advanced traffic management and scalability for large deployments. There may be integrations with other production-grade products, such as a WAF or service mesh.
  - **Scalable** – Organizations often find these options to be time savers since they tend to have more "out-of-the-box" capabilities that don't require customization or workarounds. They can easily be added to automation pipelines to allow your infrastructure to grow as needed.
  - **Reliable and supported** – One of the main benefits of commercial products is that they're stable, which means extensively tested at each release, with regular software updates and security patches as needed. Full commercial support is typically available in various tiers, so you can often get confidential help within minutes or hours of encountering a critical problem.

- **Cons:**
  - **Slower development** – Because stability is important for commercial Ingress controllers, their feature velocity might lag a little bit behind their open source counterparts.
  - **Cost (money)** – The reality of commercial products is they cost money. For organizations that have more developer cycles than cash, the cost can be a deal breaker until that situation changes.

ONCE AN ORGANIZATION REACHES A HIGH DEGREE OF COMPLEXITY, A COMMERCIAL INGRESS CONTROLLER MAKES SENSE

- **Summary:** As organizations scale, the choice of Ingress controller becomes more important based on the complexity of their teams and apps. Once an organization reaches a high degree of complexity, a commercial Ingress controller makes sense because it can reduce management complexity and accelerate time to market for new product features.

NEXT STEP: EVALUATE THE OPTIONS

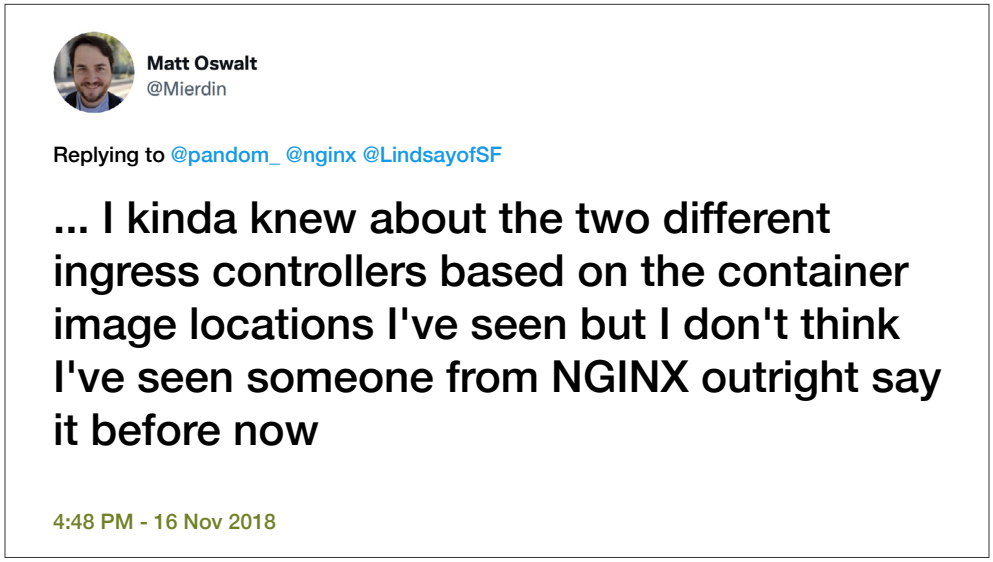
At this stage in your journey, you're ready to home in on some Ingress controllers to try by eliminating options that can't meet your needs. One great place to start your high-level feature comparison is learnk8s, which provides a free comparison table of the Ingress controllers they've evaluated.

As you're researching Ingress controllers, you'll likely notice that many options are based on NGINX. Read chapter 8 for an overview of the NGINX-based choices.

8. How to Choose an Ingress Controller - NGINX Ingress Controller Options

According to the Cloud Native Computing Foundation's (CNCF) Survey 2020, NGINX is the most commonly used data plane in Ingress controllers for Kubernetes – but did you know there's more than one "NGINX Ingress Controller"?

When this chapter was originally published in 2018 as a blog titled W*ait, Which NGINX Ingress Controller for Kubernetes Am I Using?*, it was prompted by a conversation with a community member about the existence of two popular Ingress controllers that use NGINX.



It's easy to see why there was (and still is) confusion. Both Ingress controllers are:

- Called "NGINX Ingress Controller"
- Open source
- Hosted on GitHub with very similar repo names
- The result of projects that started around the same time

And of course the biggest commonality is that they implement the same function.

NGINX VS. KUBERNETES COMMUNITY INGRESS CONTROLLER

For the sake of clarity, we differentiate the two versions like this:

- **Community version:** Found in kubernetes/ingress-nginx on GitHub, the community version is based on NGINX Open Source with docs on Kubernetes.io. It is maintained by the Kubernetes community with a commitment from F5 NGINX to help manage the project.

- **NGINX version:** Found in nginxinc/kubernetes-ingress on GitHub, NGINX Ingress Controller is developed and maintained by NGINX with docs on docs.nginx.com. It is available in two editions:
  - NGINX Open Source-based (free and open source option)
  - F5 NGINX Plus-based (commercial option)

There are also a number of other Ingress controllers based on NGINX, such as Kong, but fortunately their names are easily distinguished. If you're not sure which NGINX Ingress Controller you're using, check the container image of the running Ingress controller, then compare the Docker image name with the repos listed above.

NGINX Ingress Controller Goals and Priorities

A primary difference between NGINX Ingress Controller and the community Ingress controller (along with other Ingress controllers based on NGINX Open Source) are their development and deployment models, which are in turn based on differing goals and priorities.

OUR TOP PRIORITY FOR ALL NGINX PROJECTS AND PRODUCTS IS TO DELIVER A FAST, LIGHTWEIGHT TOOL WITH LONG-TERM STABILITY AND CONSISTENCY

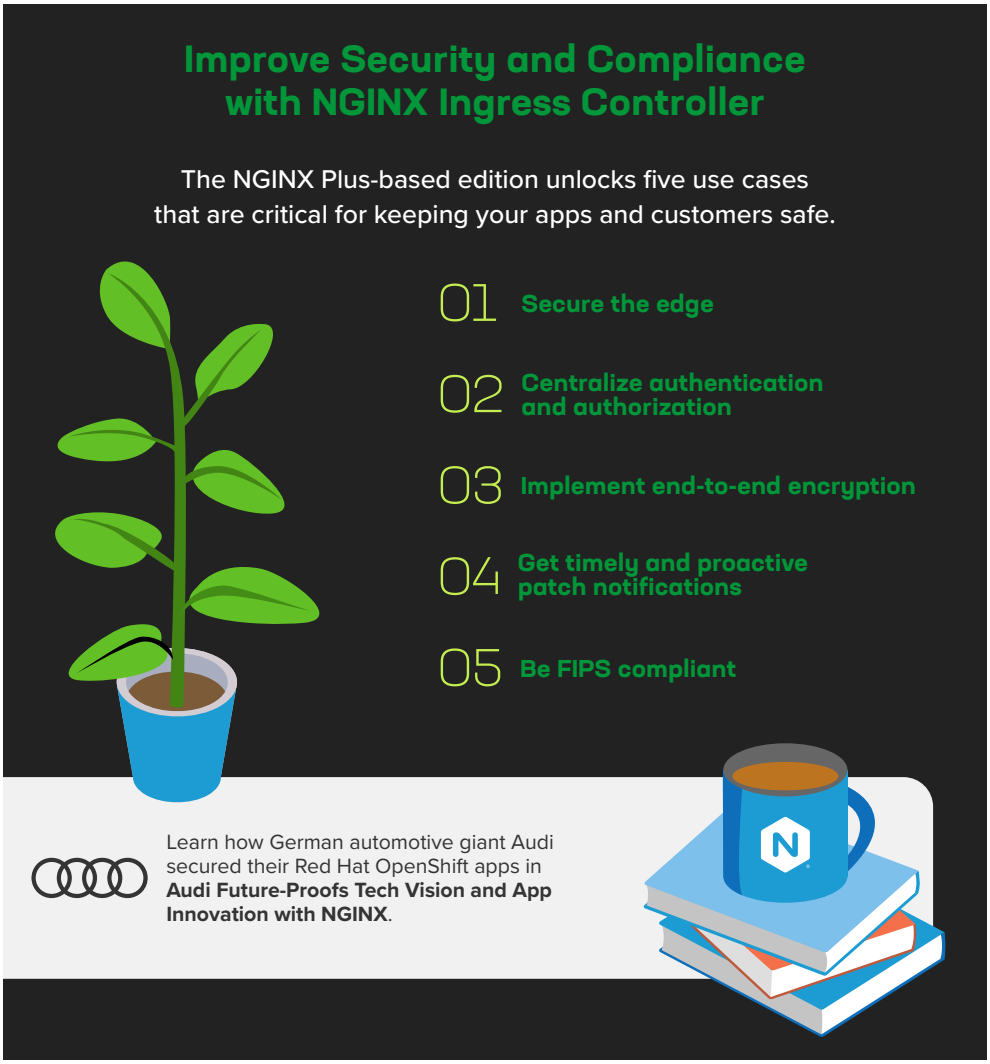
- **Development philosophy** – Our top priority for all NGINX projects and products is to deliver a fast, lightweight tool with long-term stability and consistency. We make every possible effort to avoid changes in behavior between releases, particularly any that break backward compatibility. We promise you won't see any unexpected surprises when you upgrade. We also believe in choice, so all our solutions can be deployed on any platform including bare metal, containers, VMs, and public, private, and hybrid clouds.
- **Integrated codebase** – NGINX Ingress Controller uses a 100% pure NGINX Open Source or NGINX Plus instance for load balancing, applying best-practice configuration using native NGINX capabilities alone. It doesn't rely on any third-party modules or Lua code that have not benefited from our interoperability testing. We don't assemble our Ingress controller from lots of third-party repos; we develop and maintain the load balancer (NGINX and NGINX Plus) and Ingress controller software (a Go application) ourselves. We are the single authority for all components of our Ingress controller.
- **Advanced traffic management** – One of the limitations of the standard Kubernetes Ingress resource is that you must use auxiliary features like annotations, ConfigMaps, and customer templates to customize it with advanced features. NGINX Ingress Resources provide a native, type-safe, and indented configuration style which simplifies implementation of Ingress load balancing capabilities, including TCP/UDP, circuit breaking, A/B testing, blue-green deployments, header manipulation, mTLS, and WAF.
- **Continual production readiness** – Every release is built and maintained to a supportable, production standard. You benefit from this "enterprise-grade" focus equally whether you're using the NGINX Open Source-based or NGINX Plus-based edition. NGINX Open Source users can get their questions answered on GitHub by our engineering team, while NGINX Plus subscribers get best-in-class support. Either way it's like having an NGINX developer on your DevOps team!

NGINX OPEN SOURCE VS. NGINX PLUS – WHY UPGRADE TO OUR COMMERCIAL EDITION?

And while we're here, let's review some of the key benefits you get from the NGINX Plus-based NGINX Ingress Controller. As we discussed in chapter 7, there are substantial differences between open source and commercial Ingress controllers. If you're planning for large Kubernetes deployments and complex apps in production, you'll find our commercial Ingress controller saves you time and money in some key areas.

Security and Compliance

One of the main reasons many organizations fail to deliver Kubernetes apps in production is the difficulty of keeping them secure and compliant. The NGINX Plus-based NGINX Ingress Controller unlocks five use cases that are critical for keeping your apps and customers safe (see chapter 4 for a deep dive on these use cases and more).

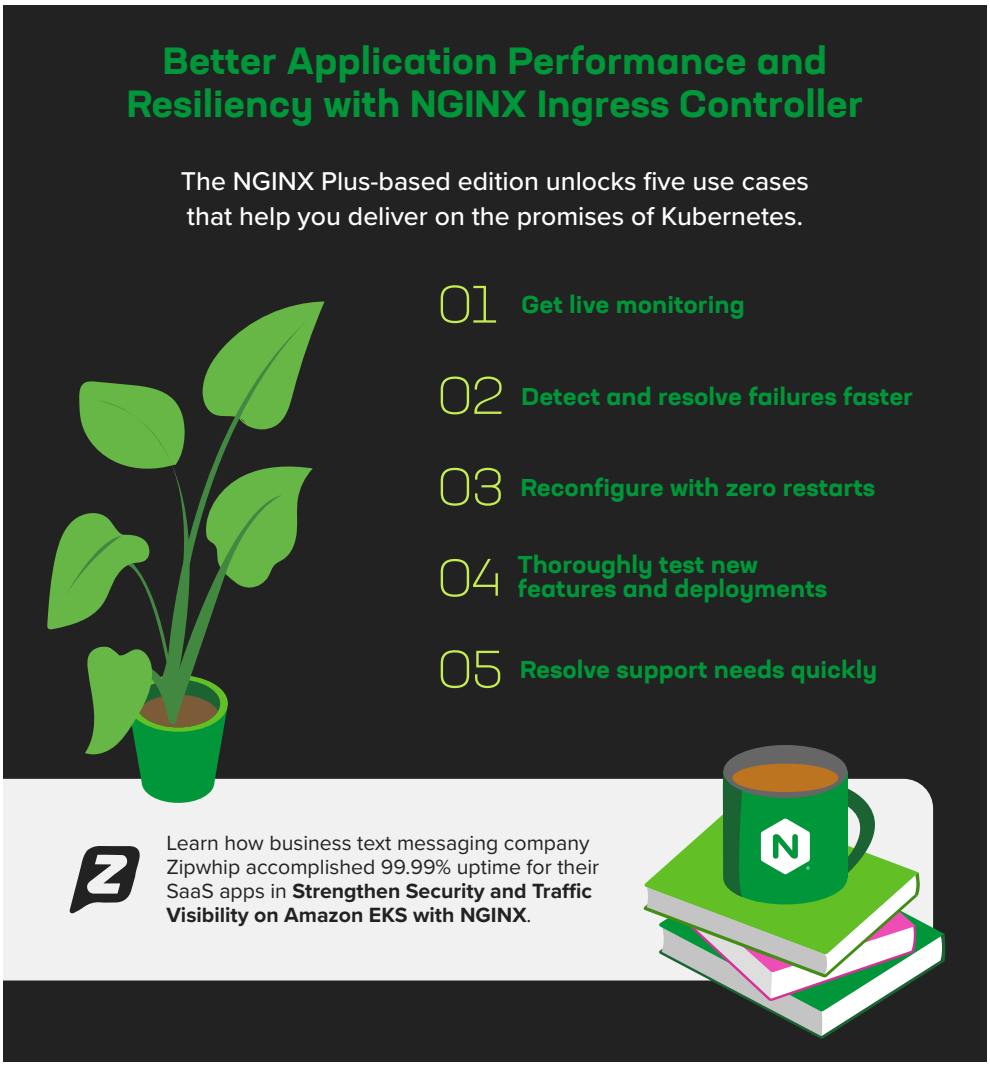


- **Secure the edge** – In a well-architected Kubernetes deployment, the Ingress controller is the only point of entry for data-plane traffic flowing to services running within Kubernetes, making it an ideal location for a WAF. F5 NGINX App Protect WAF integrates with NGINX Ingress Controller to protect your Kubernetes apps against the OWASP Top 10 and many other vulnerabilities, ensures PCI DSS compliance, and outperforms ModSecurity.
- **Centralize authentication and authorization** – You can implement an authentication and single sign-on (SSO) layer at the point of ingress with OpenID Connect (OIDC) – built on top of the OAuth 2.0 framework – and JSON Web Token (JWT) authentication.
- **Implement end-to-end encryption** – When you need to secure traffic between services, you're probably going to look for a service mesh. The always-free F5 NGINX Service Mesh integrates seamlessly with NGINX Ingress Controller, letting you control both ingress and egress mTLS traffic efficiently with less latency than other meshes.
- **Get timely and proactive patch notifications** – When CVEs are reported, subscribers are proactively informed and get patches quickly. They can apply the patches right away to reduce the risk of exploitation, rather than having to be on the lookout for updates in GitHub or waiting weeks (even months) for a patch to be released.
- **Be FIPS compliant** – You can enable FIPS mode to ensure clients talking to NGINX Plus are using a strong cipher with a trusted implementation.

Learn how German automotive giant Audi secured their Red Hat OpenShift apps in Audi Future-Proofs Tech Vision and App Innovation with NGINX

Application Performance and Resilience

Uptime and app speed are often key performance indicators (KPIs) for developers and Platform Ops teams. The NGINX Plus-based NGINX Ingress Controller unlocks five use cases that help you deliver on the promises of Kubernetes (many of which are covered in chapters 2 and 3).



- **Get live monitoring** – The NGINX Plus dashboard displays hundreds of key load and performance metrics so you can quickly troubleshoot the cause of slow (or down!) apps.
- **Detect and resolve failures faster** – Implement a circuit breaker with active health checks that proactively monitors the health of your TCP and UDP upstream servers.
- **Reconfigure with zero restarts** – Faster, non-disruptive reconfiguration ensures you can deliver applications with consistent performance and resource usage and lower latency than the open source alternatives.
- **Thoroughly test new features and deployments** – Make A/B testing and blue-green deployments easier to execute by leveraging the key-value store to change the percentages without the need for reloads.
- **Resolve support needs quickly** – Confidential, commercial support is essential for organizations that can't wait for the community to answer questions or can't risk exposure of sensitive data. NGINX support is available in multiple tiers to fit your needs, and covers assistance with installation, deployment, debugging, and error correction. You can even get help when something just doesn't seem "right".

Learn how business text messaging company Zipwhip accomplished 99.99% uptime for their SaaS apps in Strengthen Security and Traffic Visibility on Amazon EKS with NGINX

NEXT STEP: TRY NGINX INGRESS CONTROLLER

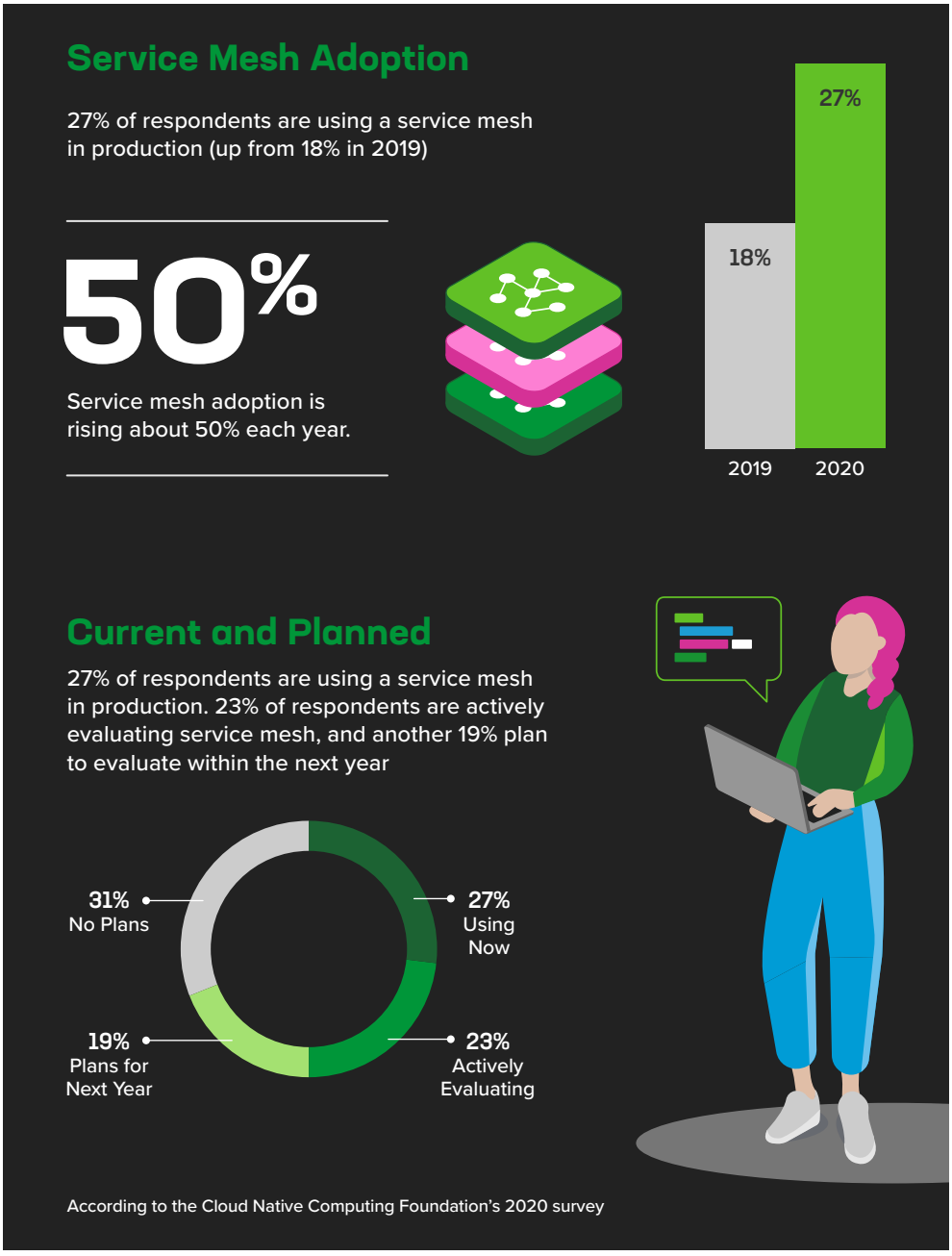
If you've decided that an open source Ingress controller is the right choice for your apps, then you can get started quickly at our GitHub repo.

For large production deployments, we hope you'll try our commercial Ingress controller based on NGINX Plus. It's available for a 30-day free trial that includes NGINX App Protect.

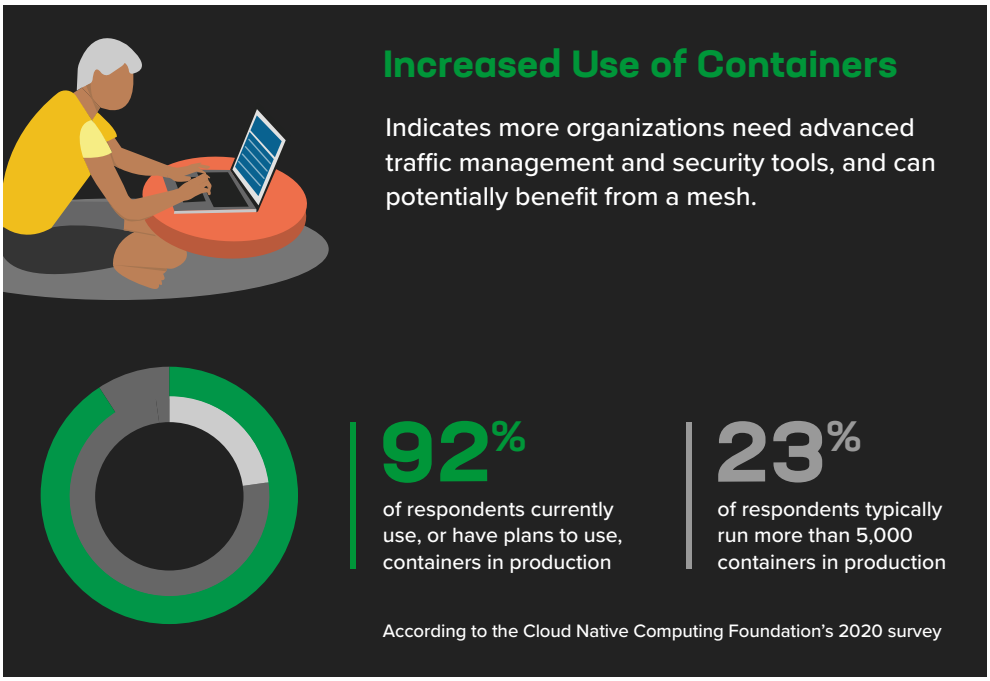
9. How to Choose a Service Mesh

In recent years, service mesh adoption has steadily moved from the bleeding edge to the mainstream as organizations deepen investment in microservices and containerized apps. The Cloud Native Computing Foundation's 2020 survey about use of cloud native technologies leads us to the following conclusions.

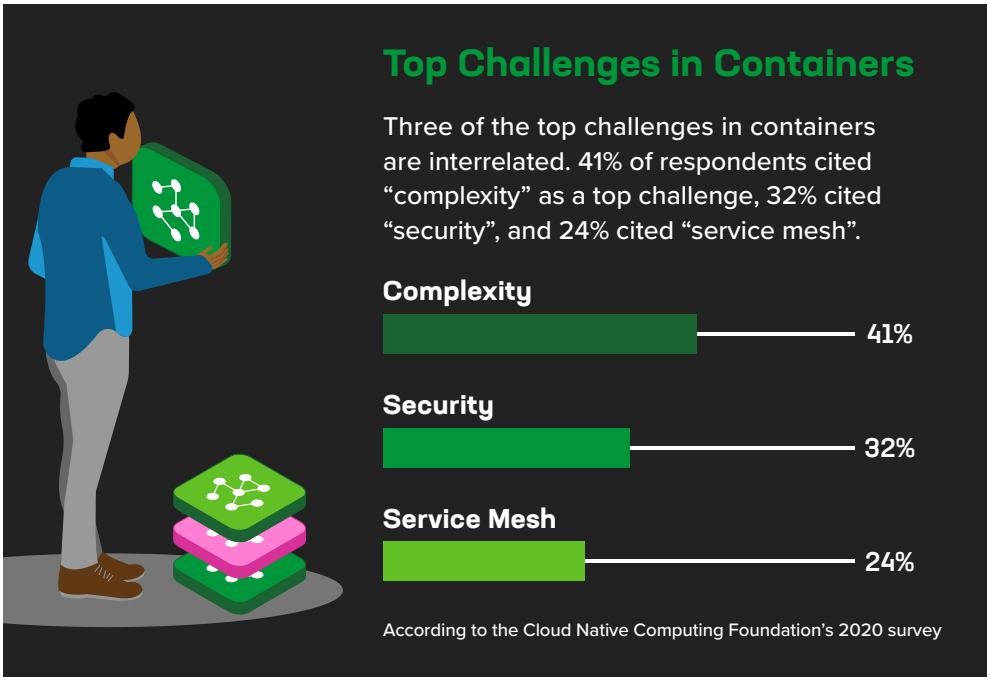
Takeaway #1: Service mesh adoption is rising rapidly.



Takeaway #2: Increased use of containers indicates more organizations need advanced traffic management and security tools, and can potentially benefit from a mesh.



Takeaway #3: Three of the top challenges in containers are interrelated.



ARE YOU READY FOR A SERVICE MESH?

IT'S NO LONGER A BINARY QUESTION OF "DO I HAVE TO USE A SERVICE MESH?"

At F5 NGINX, we think it's no longer a binary question of "Do I have to use a service mesh?" but rather "When will I be ready for a service mesh?" We believe that anyone deploying containers in production and using Kubernetes to orchestrate them has the potential to reach the level of app and infrastructure maturity where a service mesh adds value.

But as with any technology, implementing a service mesh before you need one just adds risk and expense that outweigh the possible benefits to your business. We use this six-point checklist with customers who are interested in adopting a service mesh, both to determine readiness and to gain an understanding of the modernization journey. The more statements are true for you, the more a service mesh will add value.

#1: You are fully invested in Kubernetes for your production environment.
Whether you've already moved production apps into Kubernetes or you're just starting to test app migration to container workloads, your long-term application management roadmap includes Kubernetes.

#2: You require a zero-trust production environment and need mutual TLS (mTLS) between services.
You either already rely on a zero-trust model for your production apps and need to maintain that level of security in your containerized environment, or you're using the migration as a forcing function to increase your service-level security.

#3: Your app is complex in both number and depth of services.
You have a large, distributed app. It has multiple API dependencies and most likely requires external dependencies.

#4: You have a mature, production CI/CD pipeline.
"Mature" depends on your organization. We apply the term to procedures that programmatically deploy Kubernetes infrastructure and apps, likely using tools including Git, Jenkins, Artifactory, or Selenium.

#5: You are deploying frequently to production – at least once per day.
This is where we find most people answer "no" – although they've moved apps into production Kubernetes, they aren't yet using Kubernetes for the dream goal of constant revisions.

#6: Your DevOps team is ready to rock and start using the right tools for ultra-modern app deployment!
Even if the service mesh is going to be owned by your NetOps team, administration is often handled within the cluster by DevOps teams and they need to be ready to handle the addition of a mesh to their stack.

Didn't answer "yes" to all six statements? That's ok! Keep reading to get an idea of where your journey will head once you're ready, including what you can do to prepare your team for a service mesh.

Are You Ready for a Service Mesh?

Use our 6-point checklist to determine your readiness - the more statements are true for you, the more a service mesh will add value.

01

You are fully invested in Kubernetes for your production environment.

02

You require a zero-trust production environment and need mutual TLS (mTLS) between services.

Your app is complex in both number and depth of services.

03

CI CD 04

You have a mature, production CI/CD pipeline.

05

You are deploying frequently to production – at least once per day.

Your DevOps team is ready to rock and start using the right tools for ultra-modern app deployment!

06

HOW TO CHOOSE THE SERVICE MESH THAT'S RIGHT FOR YOUR APPS

ISTIO ISN'T THE ONLY CHOICE, NOR IS IT THE *RIGHT* CHOICE FOR EVERYONE OR EVERY USE CASE

Once you've decided that you're ready for a service mesh, there's still a wide variety from which to choose. Much like Kubernetes has become the de facto container orchestration standard, Istio is often seen as the de facto service mesh standard. In fact, it's easy to think that Istio is the only choice, not just because of its prevalence but also because it aims to solve just about every problem in the Kubernetes networking world. At the risk of sounding self-serving, we're here to tell you that Istio isn't the only choice, nor is it the *right* choice for everyone or every use case. Istio adds a lot of complexity to your environments, can require an entire team of engineers to run it, and often ends up introducing more problems than it solves.

To get clear on which service mesh is best for your apps, we recommend a strategic planning session with your team and stakeholders. Here's a conversation guide to help you facilitate these discussions.

Step 1: Why are you looking for a service mesh?

In other words, what problems do you need the service mesh to solve? For example, your organization might mandate mTLS between services or you need end-to-end encryption, including both from the edge in (for ingress traffic) and from within the mesh (for egress traffic). Or perhaps you need enterprise-grade traffic management tools for your new Kubernetes services.

Step 2: How will you use the service mesh?

This depends on who you are.

- If you're a developer:
  - Do you plan to add security to a legacy app that is moving into Kubernetes?
  - Are you going to incorporate security as you refactor an app into a native Kubernetes app?
- If you're responsible for platforms and infrastructure:
  - Are you going to add the service mesh into your CI/CD pipeline so that it's automatically deployed and configured with every new cluster and available when a developer spins up a new instance?

Step 3: What factors influence your selection?

Does your service mesh need to be infrastructure-agnostic? Compatible with your visibility tools? Kubernetes-native? Easy to use? Do you see a future when you'll want to manage ingress and egress (north-south) traffic at the edge with the same tool as service-to-service (east-west) traffic within the mesh?

Step 4: Evaluate service mesh options

Once you've worked through these questions, you'll have a solid list of requirements to use as you evaluate options. There are two additional areas to assess during this process: the service mesh's data plane and the hidden costs that may come with the service mesh.

A SLOW-TO-RESPOND DATA PLANE WILL SLOW THE ENTIRE KUBERNETES ENGINE DOWN AND AFFECT SYSTEM PERFORMANCE

Data plane – The data plane directly influences customer perceptions of performance because a slow-to-respond data plane will slow the entire Kubernetes engine down and affect system performance. Use these questions to assess whether your service mesh candidate's data plane can support your needs.

- How many years has the data plane been around?
- What is the capacity of the data plane?
- Does the data plane have the integrations you need and want in the future?
- How does your data plane instrument and provide observability?
- Can the data plane dynamically recover from catastrophic failures?

Read more in Your Data Plane Is Not A Commodity.

ANY COSTS FOR DEPLOYING AND OPERATING SERVICE MESHES MAY NOT BE READILY APPARENT

Hidden costs – Any costs for deploying and operating service meshes may not be readily apparent and the bill can balloon once you are beyond rip-and-replace. Use these questions to get an accurate picture of any hidden costs that may come with your service mesh choice.

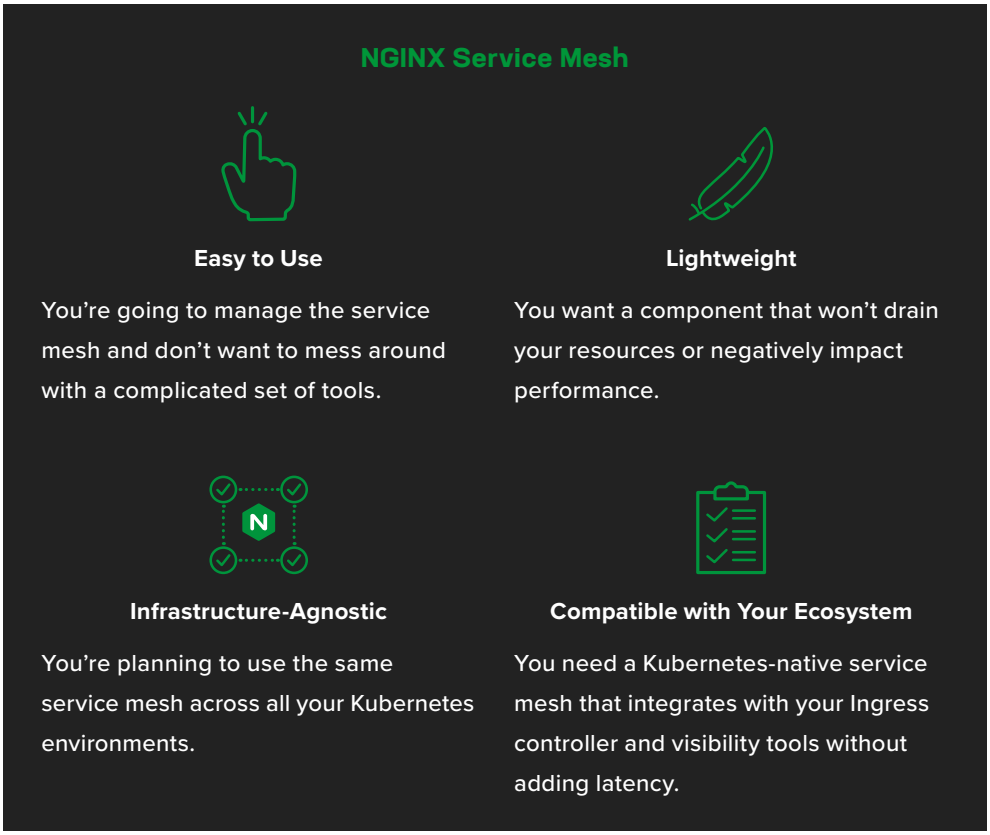
- How many container images does it take to run your control plane? And how large does each image have to be?
- What is the capacity of your Ingress controller for your service mesh?
- Can your sidecar keep up with your service demand?
- Will you be running multiple clusters or multi-tenancy?
- How many Kubernetes CustomResourceDefinitions (CRDs) does your service mesh require?
- Do you need dedicated staff to run the service mesh? If so, how many?
- Does your service mesh choice lock you into a specific choice of software or cloud?

Read more in The Hidden Costs of Service Meshes.

F5 NGINX SERVICE MESH

NGINX Service Mesh – introduced as a development release in 2020 – is free, optimized for developers, and the lightest, easiest way to implement mTLS and end-to-end encryption in Kubernetes for both east-west (service-to-service) traffic and north-south (ingress and egress) traffic. We built our own service mesh because we wanted to give you full control of the application data plane in the way that's least intrusive but still provides advanced flexibility and critical insights.

We think you'll like NGINX Service Mesh if you need a mesh that is:



About the Architecture

NGINX Service Mesh has two main components:

- **Control plane**
  We built a lightweight control plane that provides dynamic support and management of apps in partnership with the Kubernetes API server. It reacts to and updates apps as they scale and deploy so that each workload instance automatically stays protected and integrated with other app components – letting you "set it and forget it" and spend your time on valuable business solutions.
- **Data plane**
  The real star of NGINX Service Mesh is the fully integrated, high-performance data plane. Leveraging the power of F5 NGINX Plus to operate highly available and scalable containerized environments, our data plane brings a level of enterprise traffic management, performance, and scalability to the market that no other sidecars can offer. It provides the seamless and transparent load balancing, reverse proxy, traffic routing, identity, and encryption features needed for production-grade service mesh deployments. When paired with the NGINX Plus-based version of F5 NGINX Ingress Controller, it provides a unified data plane that can be managed with a single configuration.

OUR DATA PLANE BRINGS A LEVEL OF ENTERPRISE TRAFFIC MANAGEMENT, PERFORMANCE, AND SCALABILITY TO THE MARKET THAT NO OTHER SIDECARS CAN OFFER

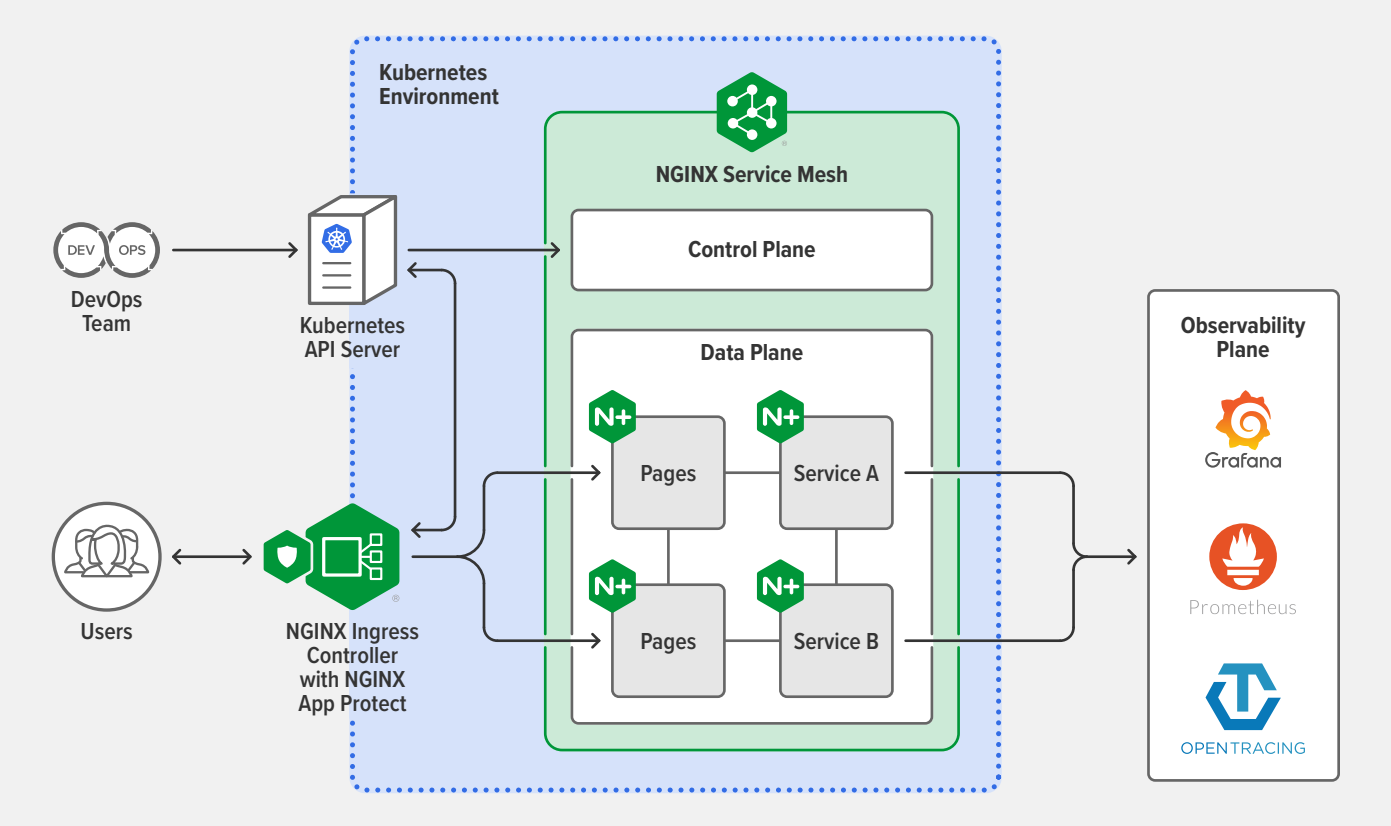


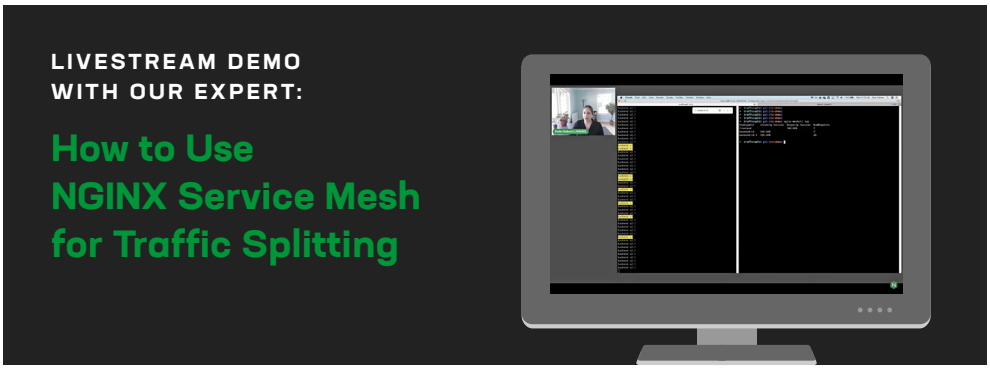
Figure 14: NGINX Service Mesh Architecture

NGINX SERVICE MESH BENEFITS

You can expect several benefits from NGINX Service Mesh:

- **Less complexity**
  NGINX Service Mesh is easy to use and infrastructure-agnostic. It implements the Service Mesh Interface (SMI) specification, which defines a standard interface for service meshes on Kubernetes, and provides SMI extensions that make it possible to roll out a new app version with minimal effort and interruption to production traffic. NGINX Service Mesh also integrates natively with NGINX Ingress Controller, creating a unified data plane where you can centralize and streamline configuration of ingress and egress (north-south) traffic management at the edge with service-to-service (east-west) reverse proxy sidecar traffic management. And unlike other meshes, NGINX Service Mesh doesn't need to inject a sidecar into NGINX Ingress Controller, so it doesn't add latency and complexity to your Kubernetes environments.
- **Improved resilience**
  With our intelligent management of container traffic, you can specify policies that limit traffic to newly deployed service instances and slowly increase it over time. Capabilities like rate limiting and circuit breakers give you full control over the traffic flowing through your services. You can leverage a robust range of traffic distribution models, including rate shaping, quality of service (QoS), service throttling, blue-green deployments, canary releases, circuit breaker pattern, A/B testing, and API gateway features.

  Check out chapter 2 on improving resilience in Kubernetes and watch this demo, How to Use NGINX Service Mesh for Traffic Splitting, by NGINX engineer Kate Osborn to learn more.



- **Fine-grained traffic insights**

  NGINX Service Mesh is instrumented for metrics collection and analysis using OpenTracing and Prometheus. The NGINX Plus API generates metrics from NGINX Service Mesh sidecars and NGINX Ingress Controller pods. With pre-built Grafana dashboards you can improve insight into app and API performance by visualizing metrics with detail down to the milisecond, day-over-day overlays, and traffic spikes.

  Learn more about ways to improve visibility in chapter 3 and check out this livestream: How to Improve Visibility in Kubernetes with Prometheus, Grafana, and NGINX.

- **Secure containerized apps**

  Extend mTLS encryption and Layer 7 protection all the way down to individual microservices and leverage access controls to define policies which describe the topology of your application – giving you granular control over which services are authorized to talk with each other. NGINX Service Mesh enables advanced security features including configuration gating and governance, and allowlist support for ingress-egress and service-to-service traffic. With the NGINX Plus-based version of NGINX Ingress Controller, you also get default blocking of north-south traffic to internal services, and edge firewalling with NGINX App Protect.

  Check out this demo on using access control to manage a zero-trust environment with end-to-end encryption, How to Use NGINX Service Mesh for Secure Access Control, by NGINX engineer Aidan Carson.

NOT READY FOR A SERVICE MESH?

If you aren't yet ready for a mesh, then you're probably new to Kubernetes or you're hitting blockers that are preventing you from large production deployments. This is a great time to work on employing a production-grade Ingress controller and built-in security to address common Kubernetes challenges around complexity, security, visibility, and scalability.

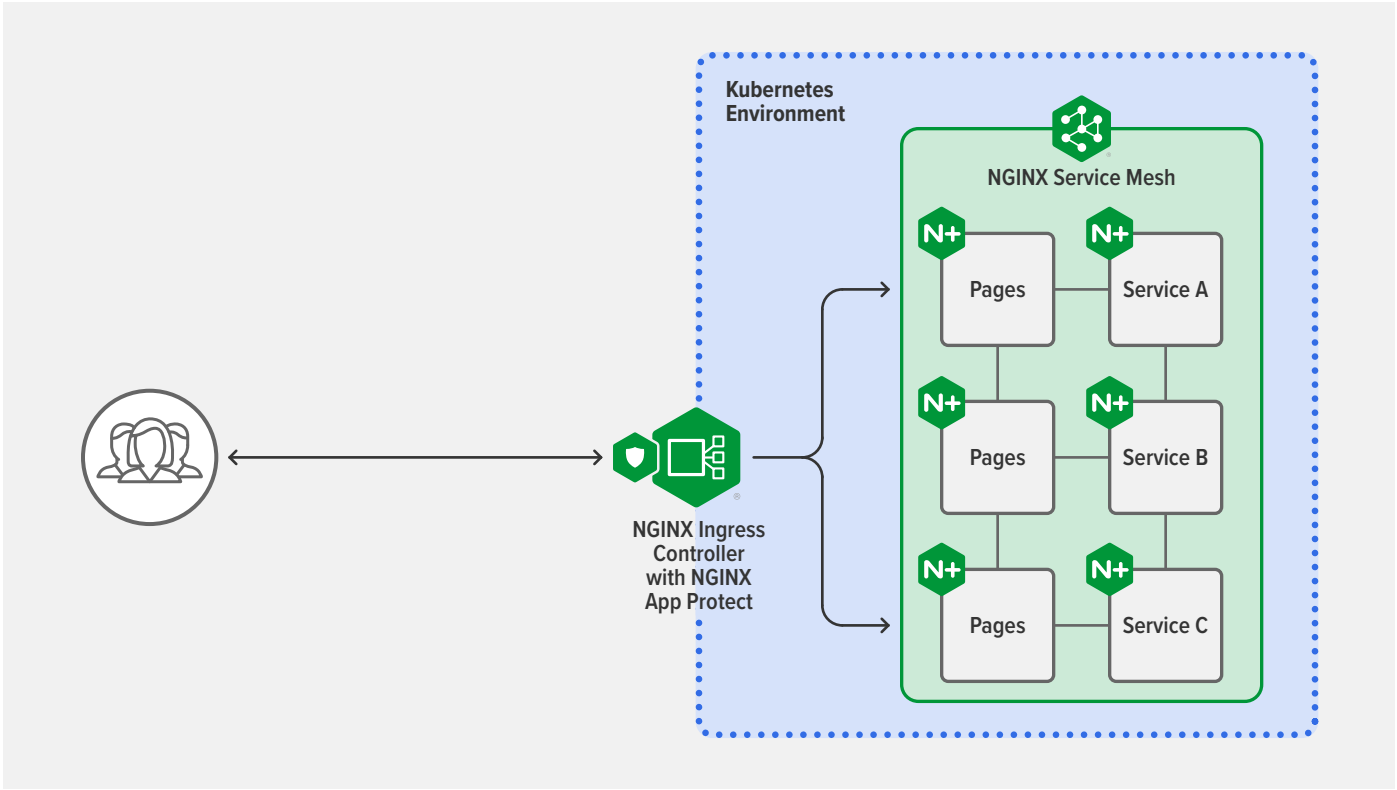


Figure 15: Production-Grade Kubernetes with NGINX

GET STARTED TODAY

Watch our webinar, Are You Service Mesh Ready? Moving from Consideration to Implementation, for a deeper dive into how to choose a service mesh!

NGINX Service Mesh is completely free and available for immediate download and can be deployed in less than 10 minutes. To get started, check out our docs and watch this short demo, Getting Started with NGINX Service Mesh, by Product Manager Alan Murphy. We'd love to hear how it goes for you, so please give us your feedback on GitHub.

If it turns out that Istio is the best fit for your needs, check out F5's Aspen Mesh. It's an enterprise-grade service mesh built on top of open source Istio. With a real-time traffic GUI, it's particularly great for service providers seeking to deliver 5G.

APPENDIX

Performance Testing Three Different NGINX Ingress Controller Options

The following test results were originally published in the blog Performance Testing NGINX Ingress Controllers in a Dynamic Kubernetes Cloud Environment. Read the blog for full details, including the testing methodology and specs.

A SLOW APP CAN QUICKLY LEAD TO LOSS OF CUSTOMERS

One of the most important factors in selecting a traffic management tool – especially for agile Kubernetes apps – is the amount of latency that the tool potentially adds. After all, a slow app can quickly lead to loss of customers. We compared the performance of three NGINX Ingress controllers in a realistic multi-cloud environment, measuring latency of client connections across the Internet:

- The NGINX Ingress Controller maintained by the Kubernetes community and based on NGINX Open Source. We refer to it here as the *community version*. We tested version 0.34.1 using an image pulled from the Google Container Registry.
- NGINX Open Source-based NGINX Ingress Controller version 1.8.0, maintained by NGINX. We refer to it here as the *NGINX Open Source version*.
- F5 NGINX Plus-based NGINX Ingress Controller version 1.8.0, maintained by NGINX. We refer to it here as the *NGINX Plus version*.

We generated a steady flow of client traffic to stress-test the Ingress controllers, and collected the following performance metrics:

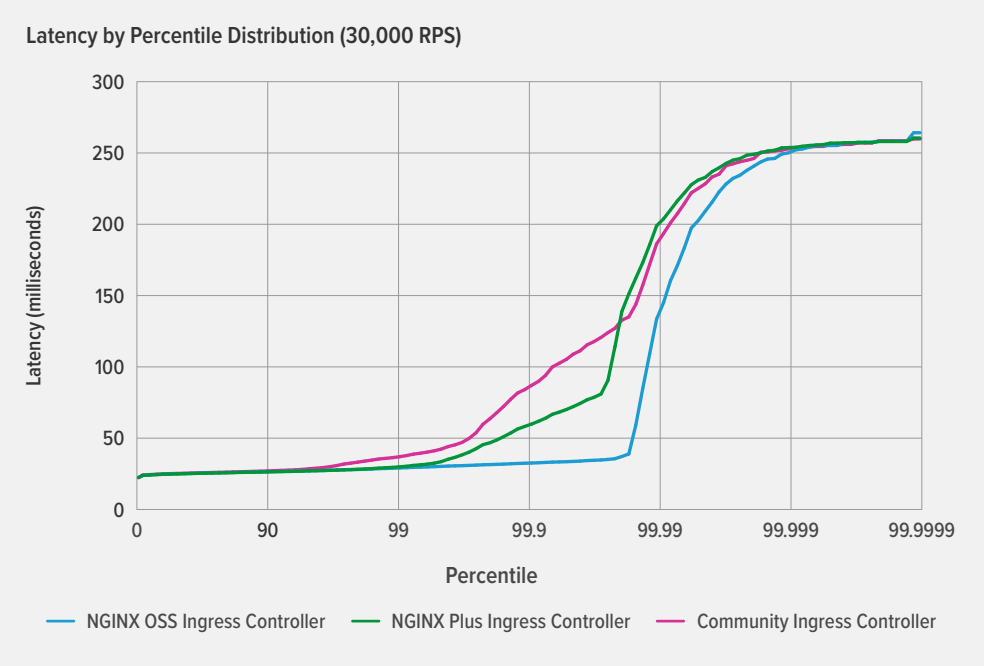
- **Latency** – The amount of time between the client generating a request and receiving the response. We report the latencies in a percentile distribution. For example, if there are 100 samples from latency tests, the value at the 99th percentile is the next-to-slowest latency of responses across the 100 test runs.
- **Connection timeouts** – TCP connections that are silently dropped or discarded because the Ingress controller fails to respond to requests within a certain time.
- **Read errors** – Attempts to read on a connection that fail because a socket from the Ingress controller is closed.
- **Connection errors** – TCP connections between the client and the Ingress controller that are not established.

We tested under two conditions. In the *static deployment*, the number of backend pod replicas remained constant at five throughout the test run. In the *dynamic deployment*, we used a script to periodically scale the number of replicas up to seven and back to five. As the following findings illustrate, we found that only the NGINX Plus version doesn't incur high latencies when the number of pod replicas scales up and down.

LATENCY RESULTS FOR THE STATIC DEPLOYMENT

As indicated in the graph, all three NGINX Ingress Controllers achieved similar performance with a static deployment of the backend application. This makes sense given that they are all based on NGINX Open Source and the static deployment doesn't require reconfiguration from the Ingress controller.

Figure 16: Latency Results for the Static Deployment



LATENCY RESULTS FOR THE DYNAMIC DEPLOYMENT

The graph shows the latency incurred by each NGINX Ingress Controller in a dynamic deployment where we periodically scaled the backend application from five replica pods up to seven and back.

IT'S CLEAR THAT ONLY THE NGINX PLUS VERSION PERFORMS WELL IN THIS ENVIRONMENT

It's clear that only the NGINX Plus version performs well in this environment, suffering virtually no latency all the way up to the 99.99th percentile. Both the community and NGINX Open Source versions experience noticeable latency at fairly low percentiles, though in a rather different pattern.

- Community version: Latency climbs gently but steadily to the 99th percentile, where it levels off at about 5000ms (5 seconds).
- NGINX Open Source version: Latency spikes dramatically to about 32 seconds by the 99th percentile, and again to 60 seconds by the 99.99th.

As we discuss further in Timeout and Error Results for the Dynamic Deployment, the latency experienced with the community and NGINX Open Source versions is caused by errors and timeouts that occur after the NGINX configuration is updated and reloaded in response to the changing endpoints for the backend application.

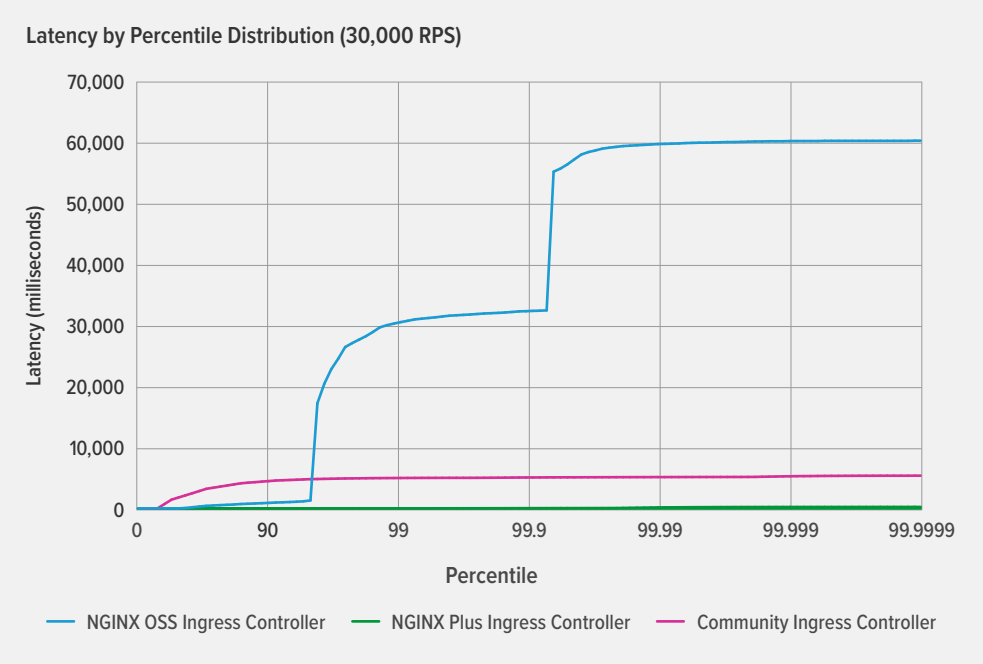
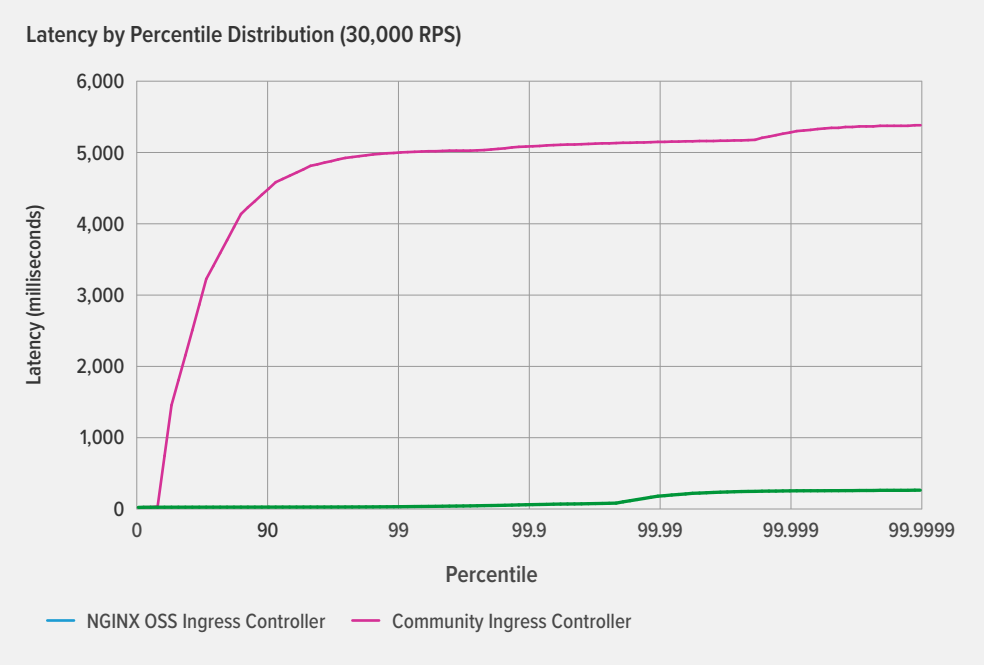


Figure 17: Latency Results for the Dynamic Deployment

Here's a finer-grained view of the results for the community and NGINX Plus versions in the same test condition as the previous graph. The NGINX Plus introduces *virtually no latency* until the the 99.99th percentile, where it starts climbing towards 254ms at the 99.9999th percentile. The latency pattern for the community version steadily grows to 5000ms latency at the 99th percentile, at which point latency levels off.

Figure 18: Finer-Grained View of Latency Results for the Dynamic Deployment



TIMEOUT AND ERROR RESULTS FOR THE DYNAMIC DEPLOYMENT

This table shows the cause of the latency results in greater detail:

	NGINX OPEN SOURCE	COMMUNITY	NGINX PLUS
Connection Errors	33,365	0	0
Connection Timeouts	309	8,809	0
Read Errors	4,650	0	0

With the NGINX Open Source version, the need to update and reload the NGINX configuration for every change to the backend application's endpoints causes many connection errors, connection timeouts, and read errors. Connection/socket errors occur during the brief time it takes NGINX to reload, when clients try to connect to a socket that is no longer allocated to the NGINX process. Connection timeouts occur when clients have established a connection to the Ingress controller, but the backend endpoint is no longer available. Both errors and timeouts severely impact latency, with spikes to 32 seconds at the 99th percentile and again to 60 seconds by the 99.99th.

With the community version, there were 8,809 connection timeouts due to the changes in endpoints as the backend application scaled up and down. The community Ingress controller uses Lua code to avoid configuration reloads when endpoints change. The results show that running a Lua handler inside NGINX to detect endpoint changes addresses some of the performance limitations of the NGINX Open Source version, which result from its requirement to reload the configuration after each change to the endpoints. Nevertheless, connection timeouts still occur and result in significant latency at higher percentiles. Also, as we discussed in chapter 6, the use of Lua in OpenResty-based Ingress controllers (including the community version) can introduce unexpected risks.

With the NGINX Plus version there were no errors or timeouts – the dynamic environment had virtually no effect on performance. This is because NGINX Plus uses the NGINX Plus API to dynamically update the NGINX configuration when endpoints change. As mentioned, the highest latency was 254ms at the 99.9999 percentile.

CONCLUSION

The performance results show that to completely eliminate timeouts and errors in a dynamic Kubernetes cloud environment, the Ingress controller must dynamically adjust to changes in backend endpoints without event handlers or configuration reloads. Based on the results, we can say that the NGINX Plus API is the optimal solution for reconfiguring NGINX in a dynamic environment. In our tests, only the NGINX Plus-based NGINX Ingress Controller achieved the flawless performance in highly dynamic Kubernetes environments that you need to keep your users satisfied.

GLOSSARY

A

A/B Testing: A process used to measure and compare user behavior for the purpose of determining the relative success of different product or app versions across the customer base. (Page 18)

Application Layer: See Layer 7.

Authentication and Authorization: Functions required to ensure only the "right" users and services can gain access to backends or application components:

- **Authentication**: Verification of identity to ensure that clients making requests are who they claim to be.
- **Authorization**: Verification of permission to access a resource or function. Accomplished through access tokens, such as Layer 7 attributes like session cookies, session IDs, group IDs, or token contents.
  (Page 35)

B

Blue-Green Deployment: A technique for reducing, or even eliminating, downtime for upgrades by keeping the old version (blue) in production while simultaneously deploying the new version (green) alongside in the same production environment.
(Page 19)

C

Canary Deployment: A safe and agile way to test the stability of a new feature or version. Most users stay on the stable version while a minority of users are moved to the new version. (Page 17)

Circuit Breaker: A way to prevent cascading failure by monitoring for service failures.
(Page 15)

Cloud-Native: An approach to application development and delivery that empowers organizations to build and run scalable applications in modern, dynamic environments (such as public, private, and hybrid clouds) with containers, service meshes, microservices, immutable infrastructure, and declarative APIs.

Conditional Routing: See Debug Routing.

Containers: A virtualization technology designed to create and support a portable form factor for applications, making it easy to deploy an application on a range of different platforms. A container packages up all the requirements for the application – the application code itself, dependencies such as libraries the application needs to run, and the run-time environment for the application and its dependencies – into a form factor which can be transported and run independently across platforms.

Critical Vulnerabilities and Exposures (CVEs): A database of publicly disclosed flaws "in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components".[1]
(Pages 30, 49, 62)

D

Debug Routing (sometimes called *Conditional Routing*): A technique in which an app is deployed publicly yet "hidden" except to users with permission to access it, based on Layer 7 attributes such as a session cookie, session ID, or group ID.
(Page 16)

Denial-of-Service (DoS) Attack: An attack in which a bad actor floods a website with requests (TCP/UDP or HTTP/HTTPS) with the goal of making the site crash.
(Page 32)

Distributed Denial-of-Service (DDoS) Attack: A version of a DoS attack in which multiple sources target the same network or service, making it more difficult to defend against than a standard DoS attack, due to the potentially large network of attackers.

E

East-West Traffic: Traffic moving among services within a Kubernetes cluster (also called *service-to-service traffic*).

Egress Traffic: Traffic exiting a Kubernetes cluster.

End-to-End Encryption (E2EE): The practice of keeping data fully encrypted during transit from the user to the app and back. E2EE requires SSL/TLS certificates and potentially mTLS, and is a key component of a zero-trust policy and environment.
(Page 37)

I

Ingress Controller: A specialized load balancer for Kubernetes (and other containerized) environments.
(Pages 8, 42)

Ingress Traffic: Traffic entering a Kubernetes cluster.

Insight: A deep understanding of a person or thing.
(Page 8)

K

Kubernetes: An open source container orchestration system, providing a complete platform for managing and scaling applications that are deployed in containers. Often abbreviated as *K8s* (pronounced "kates").

1. Source: CVE Program.

L

Layer 4: The intermediate *transport layer* in the OSI model, which deals with delivery of messages with no regard to the content of the messages.

Layer 7: The high level *application layer*, which deals with the actual content of each message.

M

Microservices: An approach to software architecture that builds a large, complex application from multiple small components which each perform a single function (such as authentication, notification, or payment processing), and also the term for the small components themselves. Each microservice is a distinct unit within the software development project, with its own codebase, infrastructure, and database. The microservices work together, communicating through web APIs or messaging queues to respond to incoming events..

Mutual TLS (mTLS): The practice of requiring authentication (via TLS certificate) for both the client and the host. What is mTLS? by F5 Labs provides an excellent primer on TLS and mTLS.
(Pages 36, 74)

N

North-South Traffic: Traffic entering and exiting a Kubernetes cluster (also called *ingress-egress traffic*).

R

Rate Limiting: The practice of restricting the number of requests a user can make in a given time period.
(Page 14)

S

Service Mesh: A traffic management tool for routing and securing service-to-service traffic.
(Pages 10, 65)

Sidecar: A separate container that runs alongside an application container in a Kubernetes pod. Typically, the sidecar is responsible for offloading functions required by all apps within a service mesh – SSL/TLS, mTLS, traffic routing, high availability, and so on – from the apps themselves, and implementing deployment testing patterns such as circuit breaker, canary, and blue-green.

Single Sign-On (SSO): The practice of requiring users to authenticate just once to access multiple apps and services. SSO technologies – including SAML, OAuth, and OIDC – make it easier to manage authentication and authorization.
(Page 35)

- **Simplified Authentication**: SSO eliminates the need for a user to have a unique ID token for each different resource or function.
- **Standardized Authorization**: SSO facilitates setting user's access rights based on their role, department, and level of seniority rather than having to separately configure rights for each user.

SSL (Secure Sockets Layer)/TLS (Transport Layer Security): A protocol for establishing authenticated and encrypted links between networked computers. (Although the SSL protocol was deprecated in 1999, it is still common to refer to these related technologies as "SSL" or "SSL/TLS.") An SSL certificate authenticates a website's identity and establishes an encrypted connection.
(Page 37)

T

Traffic Control (sometimes called *Traffic Routing* or *Traffic Shaping*): The act of controlling where traffic goes and how it gets there.
(Page 14)

Traffic Splitting (sometimes called *Traffic Testing*): A subcategory of traffic control in which incoming traffic is divided into streams directed to different versions of a backend app running simultaneously in an environment (usually the current production version and an updated version). The proportion of traffic directed to the updated version is usually increased by increments until that version receives all traffic.
(Page 14)

Transport Layer: See *Layer 4*.

V

Visibility: The state of being able to see or be seen.
(Pages 23, 74)

W

Web Application Firewall (WAF): A reverse proxy that detects and blocks sophisticated attacks against apps and APIs (including the OWASP Top 10 and other advanced threats) while letting "safe" traffic through.
(Pages 9, 32, 45, 62)

Z

Zero Trust: A security concept – used in high-security organizations but relevant to everyone – in which data must be secured at all stages of data storage and transport. This means that the organization has decided not to "trust" any users or devices by default, but rather to require that all traffic is thoroughly vetted. A zero-trust architecture typically includes a combination of authorization and mTLS, with a high probability that the organization implements end-to-end encryption.
(Pages 38, 67, 74)

NGINX®
Part of F5