White Paper

# Routing in the Cloud

## Connecting the Dots in a Microservices Architecture

træfiklabs

# Table of Contents

# Introduction

Stating that change is the only constant is no mere platitude when it comes to advancements in technology. Savvy enterprise organizations understand that outperforming competitors requires harnessing these changes in the form of continuous innovation initiatives. Software development teams lie at the heart of the resulting digital transformation activities, and improving the velocity at which they're able to deliver value amplifies overall organizational success. In that regard, engineering teams today have an unprecedented opportunity to increase their business impact due to the unique convergence of recent technologies.

It's only been about a decade since cloud computing launched into the market, and veteran developers still recall being highly dependent upon IT teams to provision compute, storage, and networking resources as part of their software development life cycle (SDLC). Due to the need for manual human intervention, the time to deploy applications from development to a production environment in these legacy models easily spanned weeks or months. By providing developers the ability to provision resources on-demand through self-service portals and APIs, cloud computing accelerated processes that could previously take months to mere minutes.

The rapid adoption of cloud computing in the enterprise is evidenced by the proliferation of public cloud options and solutions to implement private cloud environments on-premises. Early application deployment models in the cloud heavily relied upon virtual machines as the common abstraction model within which software components could be deployed. Today, there is tremendous interest in containerization technologies that virtualize the operating system to isolate multiple, granular application components within a single host (virtual or physical). The combination of a container strategy along with the self-service cloud consumption model leads naturally to microservices, the next stage in the evolution of cloud-native application architectures. While promising developers the power they need to deliver against modern enterprise demands around speed-to-market and agility, the microservices paradigm also poses new challenges.

## With great power comes great routing responsibility

By empowering development teams with the combined benefits of cloud computing and container technologies, microservices are perfectly positioned as the future of enterprise application architecture. However, it would be naive for any engineering leader to believe there's a silver bullet that can address the myriad complexity concerns which accompany modern applications. They realize it's incumbent upon them to understand the inherent tradeoffs across alternative approaches as adopting an emerging paradigm naturally brings with it the need to address new challenges. This paper provides an overview of the challenges related to adopting microservices including:

- Fundamental connectivity challenges
- DevOps challenges
- Production challenges

Articulating these challenges begs the obvious question: How can software developers and architects effectively address them to attain success with microservices? The answer, which will be explored in greater detail in the context of specific challenges, consists of employing cloud-native routing technologies that empower developers to ameliorate issues arising from microservices while still reaping the ensuing benefits.

# Modernizing enterprise applications with microservices

Experienced software architects and developers are rightfully skeptical whenever there's a claimed "better" approach and are likely to ask: Why, exactly, should microservices be pursued in lieu of traditional application architectures? The answer has to do both with the inherent limitations of monolithic architectures as well as the benefits microservices can deliver in today's landscape.

## Limitations of the monolithic application architecture

### Codebase complexity

Monolithic architectures may seem viable in early development stages when the overall codebase is relatively small and well structured. Over time, however, the need to support a large number of evolving features and an ability to integrate with other systems can easily result in spaghetti code that is difficult to maintain. Developers may subsequently avoid making changes for fear of an unintended regression or employ extensive, time-intensive testing before any changes are pushed to production. For these reasons, monolithic architectures can result in application codebases that have poor quality and are not amenable to rapid iteration cycles.

### Scalability

When applications are packaged in a single artifact and then deployed to a dedicated host, options for scalability are limited to two approaches. Developers can attempt to scale-up the underlying hardware resources and, for example, provision more powerful CPUs, additional memory, greater I/O bandwidth, etc. However, not only can this result in the need for more expensive hardware, but the capacity is unlikely to scale linearly due to the complicated relationship between resource inputs and performance. The other option is to scale-out by provisioning additional instances with a load balancer to distribute traffic. In this scenario, however, developers are encumbered with the need to manage complexities such as high-availability hardware load balancer configurations.

### Resilience

Despite the best intentions of developers and QA engineers, software will always have bugs and other deficiencies such as security vulnerabilities or resource leaks. Unfortunately, with a monolithic application architecture a single issue impacts the entire workload, including all instances in a scale-out deployment. The fact that these implementations lead to complex and incomprehensible codebases only exacerbates this problem and severely limits the applicability of monolithic approaches beyond trivial applications..

## Benefits of a microservices architecture

### Complexity management

The most immediate benefit of a microservices architecture is it allows developers to decompose complex applications into a set of services, each of which has limited functionality and scope. In contrast to a monolithic architecture that often devolves into unwieldy code, microservices naturally compartmentalize themselves into maintainable codebases that are easily approachable for new developers joining a project. Code changes, whether for feature updates or bug fixes, incur a reduced risk and allow teams to maintain a high velocity for the overall application.

## Polyglot friendliness

With a monolithic architecture, developers are typically locked into the choice of language and framework that the original team selected for the application. Given the rapidly evolving ecosystem of libraries and the unpredictable nature of application requirements over prolonged periods, this leads to a guaranteed suboptimal implementation over time. In contrast, microservices provide development teams the flexibility to select the best language and framework for each functional component.

## Scalability

Similar to a monolithic application deployment, microservices can be scaled-out as needed for additional capacity. However, a microservices architecture also allows developers to scale applications through functional decomposition. This may occur through a forward-design process where new functionality is instantiated through a corresponding new service in the architecture, or as a process of refactoring when an existing microservice is deemed to be exceeding its scope or desired complexity.

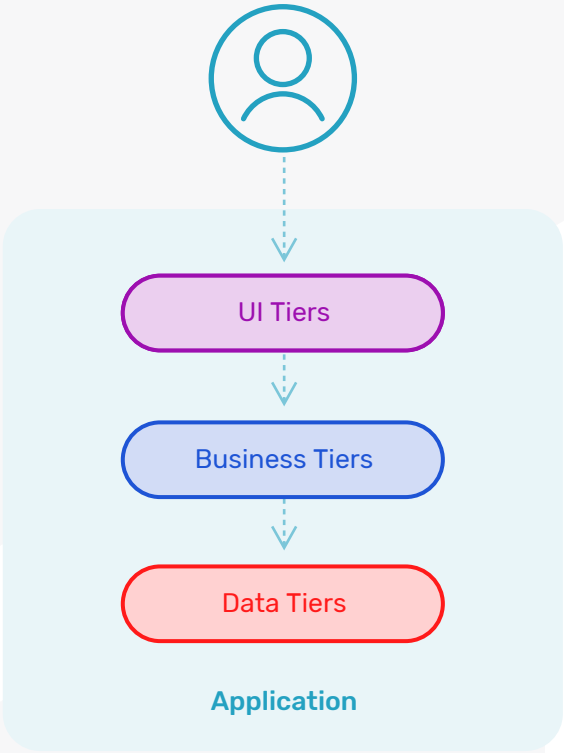
## Fault isolation

In stark contrast to monolithic implementations, when an application composed of microservices experiences a bug in production, the impact can be isolated to the specific service versus the entire application. In addition to limiting the impact from a software defect, when faults do occur it's easier for developers to identify the root cause since the investigation can be constrained to the code for the corresponding microservice.

# Routing: The key to addressing challenges with microservices

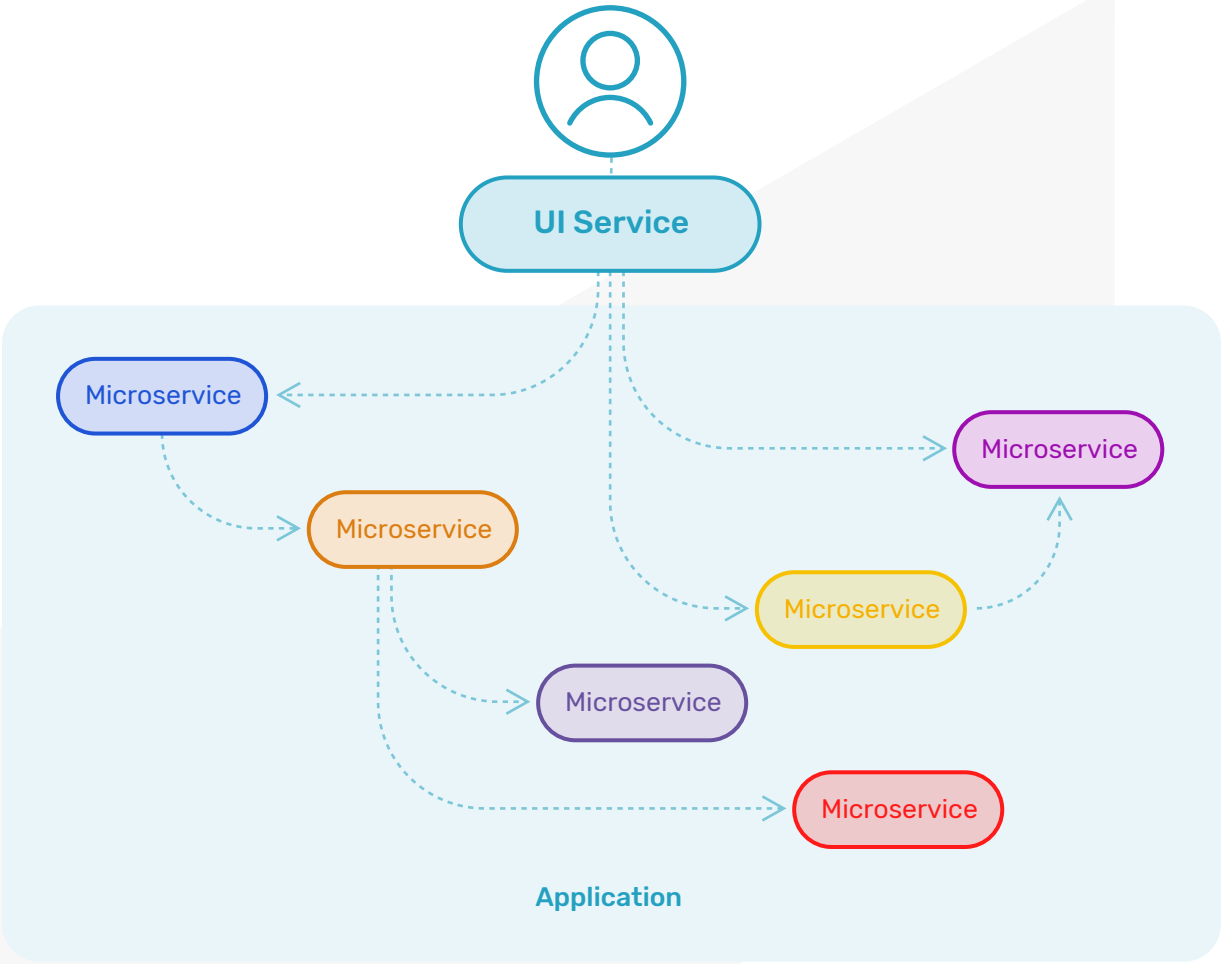
By virtue of adopting a microservices architecture, software development teams can effectively limit the scope of each independent service. However, the approach isn't a panacea that magically removes complexity from the system altogether. With any software system, there's a tendency towards conservation of overall complexity. Hence, the simplification benefits delivered by microservices must result in additional complexity of a different form.

## Dynamics: The source of complexity with microservices




*3-tier monolithic architecture*

With traditional monolithic applications, the deployment architectures are both relatively simple and static. For example, a typical three-tier system for a web application is comprised of front-end, application server (the monolith), and database tiers. Each tier can be scaled-out, and communication between tiers occurs using known, static load balancer endpoints. Once deployed, upgrades consist of atomic changes to the monolith artifact and are relatively infrequent.




*Microservices architecture*

In a microservices architecture, instead of a stable and small number of tiers, the application deployment has a large number of components in the form of containers. For a typical enterprise application, the difference can often be multiple orders of magnitude where instead of a three tiered monolith one can often find tens or hundreds of small microservices in a typical application. The resulting management problem necessitates the adoption of sophisticated orchestration platforms such as Kubernetes or Mesos to help developers automate and streamline deployment and lifecycle operations.

The added layer of abstraction from orchestrators provides benefits, but also creates new dynamics due to the additional layer of indirection. Specifically, orchestrators encapsulate their own logic for where containers should be placed, when they need to be destroyed / recreated, etc. The practical implication being the containers that underlie a microservice deployment can come and go at unpredictable intervals with corresponding changes to their IP address. Moreover, since lifecycle operations are typically performed at the granularity of microservices (versus the overarching application), service upgrades are asynchronous relative to each other. The combination of these dynamics creates multiple challenges that developers must overcome when adopting a microservices architecture.

# Challenges with microservices

## Fundamental connectivity challenges

With a monolithic application model, there are a limited number of logical network connection points and traffic patterns in the system. External users connect to the front-end tier (referred to as north-south traffic) through a load balancer that distributes requests across the application servers. Similarly, there may be some intermediary to route traffic between the application servers and database tier (referred to as east-west traffic) based upon the specific high-availability configuration adopted. Microservices create complexities for both of these traffic patterns that dictate an alternative and more sophisticated response.

In the case of north-south traffic, inbound requests are likely to be served by a variety of different microservices, and the mapping between requests and target services can vary over time. There is an even greater contrast between monolithic and microservices architectures when it comes to east-west traffic patterns due to the fact that microservices invoke each other. Not only are these dependencies unpredictable, but they can evolve and change over time. Indeed, this is simply an emergent property of the agility benefits that microservices provide to developers. However, due to the lack of stable IP addresses that services can depend on for interdependencies, there's an inherent connectivity challenge for both of these traffic patterns.

## DevOps challenges

Cloud computing and DevOps are intimately tied together in today's enterprise software development lexicon. It follows that any architectural patterns that teams adopt must align with DevOps practices around continuous integration and continuous deployment (CI/CD). The microservices approach creates a need to incorporate more sophisticated mechanisms for CI/CD due to both the growth in the number of components deployed as part of a cluster and the various interdependencies that must be accounted for and managed.

## Production challenges

Enterprise development teams understand the challenges surrounding deploying and operating an application at scale in production. The myriad aspects including monitoring, tracing, security, and fault-tolerance can be intimidating even in the context of a monolithic application architecture. These issues only grow superlinearly in complexity and importance with microservices.

# Addressing challenges with routing

The ability to functionally decompose a complex application into constituent microservices provides a clear benefit for developers and architects. The reduced complexity at the service implementation level, however, resurfaces in the form of increased dynamics and network communications. What were previously function calls translate into request-response operations that must be routed correctly and efficiently in the cluster. It's by virtue of the elevated and central role that routing plays within a microservices architecture that it becomes a point of control that developers can use to address the highlighted challenges.

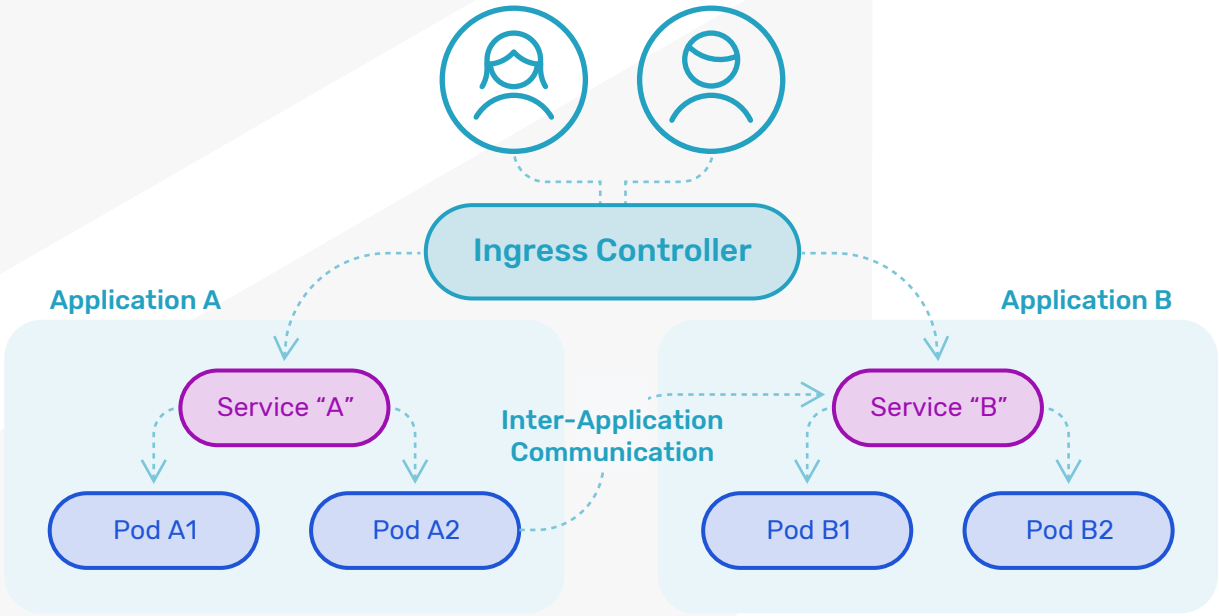
# Software-based routing with microservices

IT and operations teams historically relied upon hardware devices to realize network and routing functionality in their data centers. With advancements in commodity computing performance, software based solutions are now not only viable from a performance perspective, but preferable due to their inherent flexibility over hardware implementations. With microservices, software based routing should be employed to address various connectivity challenges.

## Edge connectivity: Reverse proxies and load balancers

In a traditional application architecture, north-south requests from users are received at the edge of the network by an intermediary load balancer which bridges external and private networks. It then distributes requests amongst a pool of backing nodes. The information regarding the set of backend nodes to target is typically configured through a manual operator step. This can often be a source of human introduced errors in the system when IP addresses get shared through spreadsheets and other ad-hoc mechanisms. Nonetheless, with a relatively static system where backend nodes have stable IPs, the approach can be viable in practice.

Microservices, of course, change the game entirely. Upon creation, containers are typically assigned IP addresses which are not only inaccessible outside of the node but can change at any time as containers get recreated due to node failures, resource reallocations, etc. Orchestrators like Kubernetes provide the capability to define logical services and can help manage the underlying host configurations to proxy and route requests, but a traditional load balancer (hardware or software) is completely oblivious to these mechanisms. The approach of an operator manually translating a set of mappings into a configuration is no longer viable in this environment.




*Ingress controller architecture*

The solution to this challenge entails employing an intelligent software router that can react in real-time and adjust to the state of the system as containers are lifecycled by orchestrators. Specifically, orchestrators provide interfaces that can be queried by software to monitor for changes in state. While a legacy load balancer treats these systems as a blackbox and relies upon an out-of-band mechanism to inject updated configurations, a cloud-native software routing solution can leverage this type of interface to track the set of backing containers for a microservice. For example, many software router technologies are capable of integrating with Kubernetes as ingress controllers. Once deployed, developers can specify an Ingress resource in the cluster which defines connectivity requirements to backing services, but abstracts the specifics of IP addresses, etc. The ingress controller honors these requests by monitoring the cluster and routing inbound requests to containers based upon where they are deployed by Kubernetes, alleviating the developer or operator from having to manually configure load balancers over time.

Microservices also demand capabilities beyond basic layer 4 load balancing. As a basic example, developers may want to expose a single logical service to users through a URL that is backed by multiple underlying microservices. A software based reverse proxy allows developers to specify mappings based on path prefixes so that https://<domain>/service1 and https://<domain>/service2 get routed to separate, operator designated microservices based upon the current state of the cluster.

## Service discovery

A software load balancer and reverse proxy can effectively address north-south traffic at the edge, but what about internal east-west traffic? With microservices, each independent service may depend upon one or more other services to realize its functionality. Moreover, as services evolve and are updated, interdependencies can change. Things are complicated even further by the fact that at any time multiple versions of a service may be deployed in a cluster. The challenge that must be overcome, then, is enabling services to locate each other in this highly dynamic setting.

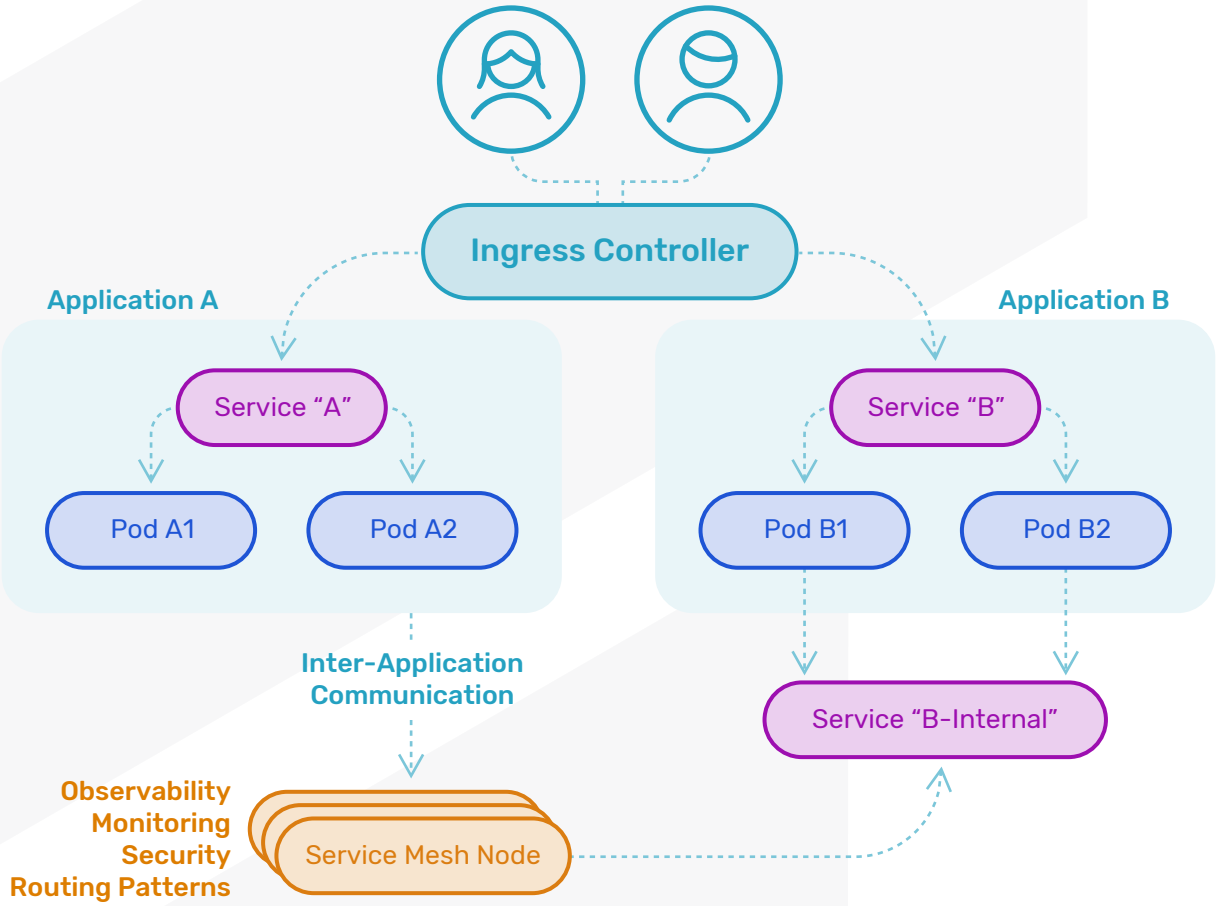
Legacy methods akin to tracking service locations manually via a spreadsheet or similar mechanism are unrealistic with microservices. An effective and robust solution can be instantiated by combining two components that fit naturally with a dynamic model: 1) A logically centralized service registry; 2) A reverse proxy that can route traffic based upon information in the registry.

A service registry allows microservices to push information to a centralized datastore that other components can then use for discovery purposes. The basic concept is that instead of a static definition that specifies where services are deployed and how they can be addressed, a dynamic registry is proactively updated at runtime generating an accurate view of the current state of the world. Service registries such as Consul and etcd are often employed for this purpose with microservices. A software router can then be used to receive and route requests by relying upon the service registries as a data provider. Developers can thereby avoid the need to implement custom discovery logic within each microservice, and traffic patterns can dynamically adjust as needed to accommodate changes in application dependencies or deployment states.

## Service mesh

While discovery is a key component of addressing issues related to east-west traffic, it doesn't fully address the scope of challenges that come along with a microservices approach. The discovery problem allows services to correctly route requests to other services based upon the current state of the cluster. However, with any network request clients must also manage concerns such as load distribution across multiple microservice instances, retry logic, and security.

To better motivate the need for a new approach, we can consider the load balancing aspect of the problem. Every microservice is scaled-out using multiple containers spanning failure domains, and requests need to be distributed amongst them to prevent any one instance from becoming overloaded. In a traditional monolithic architecture, a load balancer would be incorporated into the east-west traffic between tiers. To avoid a single point of failure, the load balancer would need to be configured for high-availability as well. Adopting the same model with a microservices architecture would be untenable since every service would require a corresponding HA load balancer.




*Service Mesh architecture*

An alternative approach for load balancing with microservices could be to implement logic within the code that would distribute requests amongst discovered service instances. However, this results in duplication and violates the principle of separation of concerns. A better model consists of interjecting an intermediary to avoid the need for microservices to directly talk to each other and at the same time that scales horizontally without any single points of failure. The service mesh model accomplishes this exactly. Many service mesh implementations employ a "sidecar" proxy that accompanies each microservice application container. Others, such as Maesh, are less invasive and rely upon a per-node proxy endpoint to enable opt-in models for the mesh. In either case, they allow microservice implementations to focus on business logic while the service mesh takes on the responsibility of managing communication concerns when outbound requests are intercepted by a proxy.

# DevOps and routing with microservices

Application codebases are continuously evolving to address new feature requirements and code deficiencies. Microservices are appealing precisely because they create an opportunity to increase the velocity of improvements and value creation. At the same time, every code change constitutes a potential risk for regression or other unexpected failure that must be mitigated. Software routing technologies can be used as part of DevOps to implement risk-mitigation strategies across the various development and deployment stages of microservices.

## Traffic shadowing

Every enterprise software engineering team employs various types of testing to ensure that implementations align with requirements and detect regressions. While unit and functional tests are important, integration and end-to-end testing are necessary to identify, diagnose, and remediate issues where components interface with each other. With a microservices architecture this type of testing is even more critical due to the inherent increase in interdependencies and potential for interface drift over time. While synthetic tests are a good starting point that provide rudimentary coverage, they don't provide a representative sample set of permutations which can arise in production. Developers can address this shortcoming by employing traffic shadowing.

Traffic shadowing consists of duplicating incoming production traffic so that requests are simultaneously received by active, production instances of a service as well as a test deployment. The immediate benefit is that developers can achieve test coverage using representative traffic from users, but at the same time the test service does not impact the production request / response flow in any way. For example, DevOps can mirror a specific percent of requests to a test service, allowing them to easily perform integration testing either as part of an automated CI/CD pipeline or manual testing and performance benchmarking.

## Canary deployments

Whereas traffic shadowing allows software to be tested in isolation using production traffic, canary deployments are an approach DevOps can use to evaluate a deployment in production while minimizing the accompanying risk. The so called "canary" deployment consists of a candidate version of a service. By routing a subset of incoming requests to the canary service which can be gradually increased, a determination can be made with regards to whether the release is ready for widespread rollout.

Implementing a canary deployment requires integrating with the edge routing component that processes inbound traffic from users. For example, in a Kubernetes environment configured with an ingress controller, it's possible for the Ingress resource configuration to support denoting multiple services that should receive requests along with a designated traffic percentage. Engineers can use this type of mechanism to begin with a small value such as 1%, and then gradually increase the load to the canary deployment as they gain confidence in the quality of the changes. Conversely, traffic can be quickly reallocated to the existing production deployment if the canary proves to have issues that need to be fixed before trying another canary deployment.

## Blue-green deployments

While canaries can be used to vett a release in a production environment, blue-green deployment strategies are a method used to release an updated version of software in a repeatable manner. They can be used as part of DevOps to implement automated rollouts in a CI/CD pipeline.

In a blue-green strategy, the new version of a service is deployed in parallel to the current instance serving production traffic. An automation server can use interfaces exposed by an orchestrator, for example, to fulfill this step of the process. The pipeline can then run automated smoke tests to ensure the service instance is behaving as expected as a sanity check before switching traffic from the old service instance over to the new. The final switchover can then be performed by invoking the software routing component of the system, which then automatically manages the transition so all traffic flows to a new blue service in lieu of the old green without further intervention from the user.

## A/B testing

At a much finer level of granularity, developers and product designers may want to test out a specific application feature in production. An example here could be to determine the impact of multiple user interface options on a specific workflow. From a DevOps standpoint, the underlying build is the same, and instead, there's some form of configuration which determines whether a given feature is activated or not. While there are various ways to approach A/B testing, a software routing based method can be implemented using the same traffic distribution method as canary testing. In this case, different instances of the service are deployed using the same build but with varying configuration attributes specified through the orchestrator or other mechanism. Developers can then distribute traffic amongst the feature groups according to the desired relative percentage.

# Production routing considerations

Managing microservices in production can be difficult for even the most experienced teams. In particular, the increased reliance on network communications can create or exacerbate operational considerations relative to traditional architectures. As it pertains to software-based routing components in a microservices architecture, there are opportunities to leverage corresponding capabilities to remediate production challenges as well as considerations that should be accounted for when selecting networking technologies.

## Tracing and monitoring

Complex software systems are difficult to fully understand and impossible to predict. Despite intensive testing efforts beforehand, anomalies are bound to arise in production. Therefore, it's critical that engineering teams have visibility into what is happening in the cluster. Tracing and monitoring are mechanisms that developers can employ to obtain these runtime insights.

With microservices, each service may be developed and owned by separate teams. Moreover, once a microservice is deployed and available, other teams may decide to rely on it without any explicit coordination. Therefore, not only is it unlikely for a single common source of truth capturing interdependencies to exist, even if one were to be created by engineers it would quickly become obsolete. A better approach is to adopt tracing tools that can dynamically identify where service invocations come from and the various call flows that ensue.

Understanding the resource consumption of services allows operators to analyze the scalability of subcomponents under load and can also enable developers to identify performance bottlenecks that need to be addressed. Monitoring tools can track metrics including CPU, network, and I/O utilization at scale providing datasets that engineers can utilize for these purposes.

All inter-service communications flow through software routing elements, and they can therefore provide an opportunity to easily collect monitoring and tracing data for microservices. As part of selecting appropriate routing solutions, developers should ensure that routing software integrates easily with the monitoring and tracing backends adopted by their teams.

## Planning for communication failures

Simply by virtue of adding more runtime elements in the form of services and network requests, a microservices architecture will increase the potential for communication failures to occur in the system. A request sent to a microservice may fail either due to the fault of the corresponding container / host, a network partition in the cluster, or a transient failure of the entire service. The application must be resilient to these types of failures to avoid negatively impacting users.

In the case of an instance failure, the load balancing mechanism can stop forwarding requests to the unhealthy instance and consolidate traffic to the remaining healthy pool. Once the instance is detected as being available again, it can be reincorporated into the load balancing set.

There may be times that an entire microservice is unavailable. When this occurs, client services should fail gracefully instead of attempting to submit retry requests that result in the consumption of additional resources and cascading failures. Circuit breaking is a common protection mechanism for this type of failure where requests to unavailable services are stopped (e.g. “opening” the circuit) and an error returned (e.g. in the form of a 503 response) allowing application logic to implement an appropriate fallback behavior.

For both of these communication failure mitigation strategies, a possible implementation strategy can be to insert the corresponding logic at each client. However, this would be both cumbersome and error prone. Instead, the responsibility for realizing health check and circuit breaking capabilities should fall to the appropriate software routing elements so development teams can focus on how the application should behave under fault scenarios instead of the network layer logic.

## Addressing unexpected load spikes

Load demands in a production environment can vary dramatically, and anomalous spikes can occur for multiple reasons. A benign cause for load spikes might be a transient and unpredictable increase in user demand. Alternatively, an increase in request volume may be part of an orchestrated distributed denial-of-service (DDOS) attack. In either case, application deployments should handle these occurrences gracefully and not fail altogether from being overloaded.

Rate limiting is a simple mechanism that can be employed to protect against broader impacts from load spikes. For example, when adopting an edge router, the software can enforce operator specified request rates to front-end services. Incorporating the protection mechanism in the routing software in this manner provides a critical safety valve to avoid unintended downtime in production deployments.

## Securing communications and certificate management

Security is always a key concern for both developers and IT when managing production environments. Two specific areas where networking and routing implementations intersect with security include enforcing network communication permissions (e.g. which services are allowed to talk to each other) and implementing encryption in flight.

Microservices are deployed into a shared computing environment where, often by default, every service is capable of accessing all of the others. While this provides a high degree of flexibility, it may not be desirable from a security point of view. For microservices that access and expose sensitive data it is often preferable to strictly limit access to client

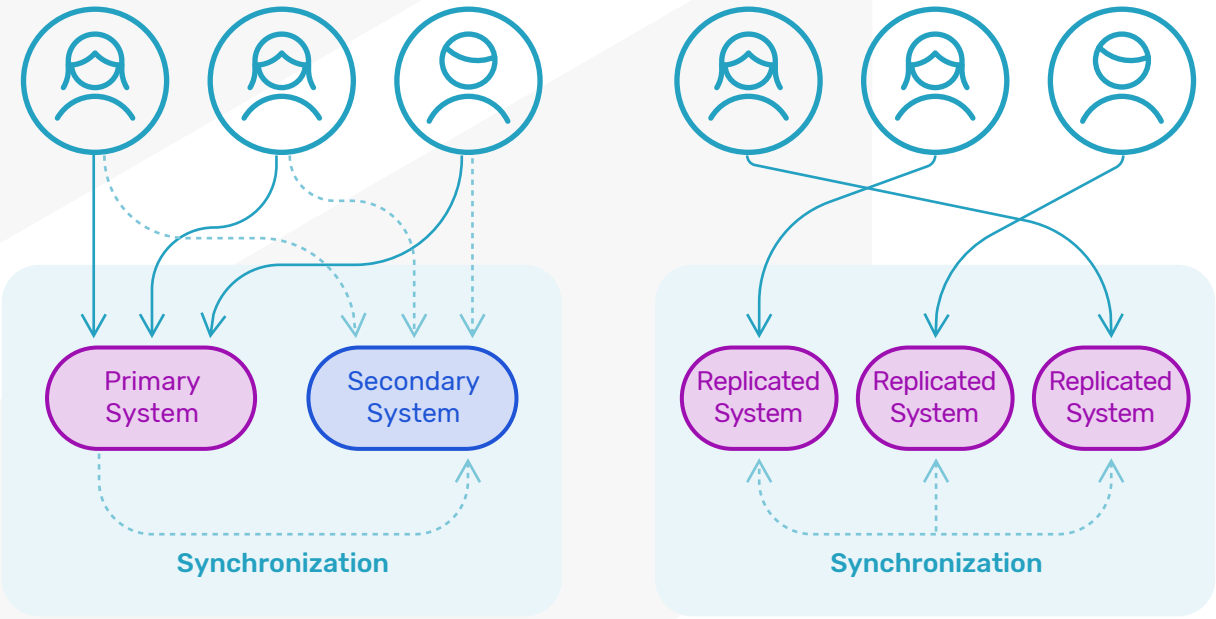
services that have an approved business need. Routing technologies can provide a means to implement these policies through network segmentation mechanisms appropriate for microservices. In a service mesh, for example, segmentation policies can be pushed and enforced at the edge of the network through the corresponding proxies.

While a general best practice, in many circumstances organizations must ensure all data is encrypted in the network for regulatory and compliance reasons. For east-west traffic, service meshes can help implement secure TLS encryption for all inter-service communications without requiring additional work for developers. In the north-south data path, the edge router can perform encryption given an appropriate certificate. More sophisticated implementations provide advanced management capabilities to remove human intervention altogether through integration with services like Let's Encrypt to automate lifecycle management of trusted certificates.

## High availability

As a general rule, single points of failure that can cause application downtime should be avoided at all costs in a production environment. To meet this requirement with microservices, engineers must evaluate the adopted software routing technologies for high availability (HA). In the traditional hardware network device model, components such as load balancers typically support a hot-cold configuration for HA whereby a dedicated, idle device is provisioned for failover if the primary stops functioning. With software routing implementations, high availability can be achieved without the expense and inefficiency of hot-cold strategies.

Resilient network routing software for the cloud should be designed for high availability. The enterprise edition of Traefik, TraefikEE, supports a highly-available architecture composed of a horizontally scalable data plane and a fault tolerant control plane. The control plane employes multiple instances that persist the state of the system using Raft as the underlying consensus protocol. Instances can be added to the data plane as needed for capacity and resilience, and the control plane can tolerate one or more failures (depending upon the configured quorum for consensus) without user facing impact of the routing service. While just one example, TraefikEE exemplifies the HA capabilities that engineers should architect for across software routing technologies in any production microservices deployment.




*Hot-cold architecture vs. Multiple replicated instances architecture*

## Multi-cloud and heterogeneous environments

Within large enterprise organizations, it's a common need for engineering teams to support deployments across multiple, heterogeneous cloud environments. Moreover, each environment may adopt a different orchestration technology for containers. For example, a public cloud environment may use Kubernetes while the production environment on-premises relies on Docker Swarm. An inherent benefit of container based microservices implementations is that they can be deployed easily across these heterogeneous boundaries. However, developers should take care to consider whether the software routing solutions they select are just as portable.

A key benefit of solutions that can support multiple orchestration technologies is that they allow engineering teams to deploy the same solution across environments while maintaining a common model for the routing layer. While this continuity is desirable in general to prevent developers from having to become familiar with multiple technologies that accomplish the same goal, it's a particularly valuable benefit in the context of microservices due to the elevated role of networking and routing in the architecture.

# Summary

Enterprise software teams are accustomed to continuously adapting to changes in the technological landscape in order to better meet the needs of their organizations. The combination of cloud computing adoption trends and the recent growth of containerization technologies has lead to a shift in mindsets regarding how enterprise applications should be architected. In order to address business demands related to velocity and agility, it's increasingly clear to software leaders that traditional monolithic designs should be eschewed in favor of a microservices architecture. To complete this transition successfully, teams must be prepared to address multiple challenges using software routing including:

- Fundamental connectivity challenges: Microservices necessitate more sophisticated traffic patterns that software routing technologies can uniquely support.
- DevOps challenges: Deployment patterns with microservices create requirements that software routing solutions help address.
- Production challenges: Meeting operational needs translates into specific feature and resilience capabilities for software routing implementations.

Technologies such as TraefikEE have been designed from the ground-up to deliver software routing in a microservices architecture with a cloud-native mindset. By adopting these solutions as part of their microservices journey, engineering leaders can be confident that their teams are well equipped to reap the benefits of this new paradigm.