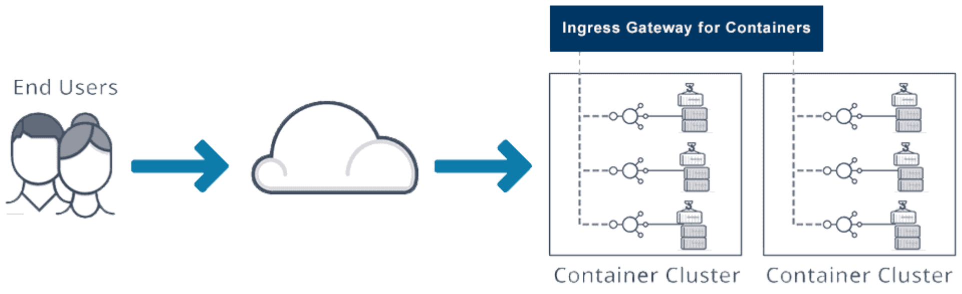
AVI

Microservices And Containers

Microservices & Containers

What are Microservices and Containers?

Microservices is an architectural design for building a distributed application. **Microservices** break an application into independent, loosely-coupled, individually deployable services. This microservices architecture allows for each service to scale or update using the deployment of service proxies without disrupting other services in the application and enables the rapid, frequent and reliable delivery of large, complex applications. Microservices get their name because each important function of an application operates as an independent service proxy. This architecture allows for each service to scale or update without disrupting other services in the application so that applications can be continuously delivered to end users. A microservices framework creates a massively scalable and distributed system, which avoids the bottlenecks of a central database and improves business capabilities, such as enabling continuous delivery/deployment applications and modernizing the technology stack.



A microservices framework including microservices and containers creates a massively scalable and distributed system, which avoids the bottlenecks of a central database. It also enables continuous integration / continuous delivery (CI/CD) pipelines for applications and modernizing the technology stack.

Companies like Amazon and Netflix have re-architected monolithic applications to microservices applications, setting a new standard for container technology.

What are Containers?

Containers are a lightweight, efficient and standard way for applications to move between environments and run independently. Everything needed (except for the shared operating system on the server) to run the application is packaged inside the container object: code, run time, system tools, libraries and dependencies.

Microservice Benefits

The biggest benefit is simplicity. Applications are easier to build, optimize and maintain when they're split into a set of smaller parts. Managing the code also becomes more efficient because each microservice is composed of different code in different programming languages, databases and software ecosystems. More microservice benefits include:

Independence — Small teams of developers can work more nimbly than large teams.

Resilience — An application will still function if part of it goes down because microservices allow for spinning up a replacement.

Scalability — Meeting demand is easier when microservices only have to scale the necessary components, which requires fewer resources.

Lifecycle automation — The individual components of microservices can more easily fit into continuous delivery pipelines when monoliths bring complexities.

Types of Containers

Stateless Microservices

Don't save or store data. Stateless microservices handle requests and return responses. Any data required for the request is lost when the request is complete. Stateless containers may use limited storage, but anything stored is lost when the container restarts.

Stateful Microservices

Requires storage to run. Stateful microservices directly read from and write to data saved in a database. Storage persists when the container restarts. However, stateful microservices don't usually share databases with other microservices.

Monolithic Architecture versus Microservices Architecture

Applications were traditionally built as monolithic pieces of software. Monolithic applications have long life cycles, are updated infrequently and changes usually affect the entire application. Adding new features requires reconfiguring and updating the entire stack — from communications to security. This costly and cumbersome process delays time-to-market and updates in application development.

Microservices architecture was designed to remedy this problem. All services are created individually and deployed separately. This allows for autoscaling based on specific business needs. Containers and microservices require more flexible and elastic load balancing due to the highly transient nature of container workloads and the rapid scaling needs without affecting other parts of the application.

Monolithic Architecture

Application is a single, integrated software instance

Application instance resides on a single server or VM

Updates to an application feature require reconfiguration of entire app

Network services can be hardware based and configured specifically for the server

Microservices Architecture

Application is broken into modular components

Application can be distributed across the clouds and datacenter

Adding new features only requires those individual microservice to be updated

Network services must be software-defined and run as a fabric for each microservice to connect to

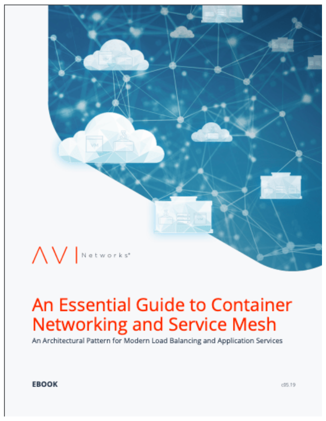
Why Microservices Architecture Needs Container Ingress

Applications require a set of services from their infrastructure—load balancing, traffic management, routing, health monitoring, security policies, service and user authentication, and protection against intrusion and DDoS attacks. These services are often implemented as discrete appliances. Providing an application with these services required logging into each appliance to provision and configure the service.

This process was possible when managing dozens of monolithic applications, but as these monoliths become modernized into microservices based applications it isn't practical to provision hundreds or thousands of containers in the same way. Observability, scalability, and high availability can no longer be provided by discrete appliances.

The advent of cloud-native applications and containers created a need for a service mesh to deliver vital application services, such as load balancing. The service mesh handles east-west services within the datacenter, with container ingress handling north-south into and out of the datacenter. By contrast, trying to place and configure a physical hardware appliance load balancer at each location and every server is overly challenging and expensive. And require businesses need to deploy microservices to keep up with application demands and multi-cloud environments.

A solution to this problem is Kubernetes ingress — a new way to deliver service-to-service communication through APIs that cannot be provided by appliances.


AVI Networks
An Essential Guide to Container Networking and Service Mesh
An Architectural Pattern for Modern Load Balancing and Application Services
EBOOK

An Essential Guide to
Container Networking and Service Mesh

Learn how both modern applications in container clusters as well as traditional applications in on-prem data centers and clouds can benefit from the granular application services made possible by a service mesh – understand what it is, why it matters and how to deploy large container clusters in production.

GET THE WHITEPAPER

Microservices Architecture Definition

Here are some high-level microservices best practices:

Fully Commit to Microservices: Trying to turn a nicely designed monolithic architecture with tightly coupled modules into microservices will likely cost more money especially if you have to breakdown an application to retro fit the design. Start with a purpose-built deployment architecture from the ground up.

Assemble an Integrated Team: Designing effectively will require architects, developers, domain experts and business leaders to collaborate on defining the Bounded Context and Core Domain and Ubiquitous Language, Subdomains, Context Maps. Developers and architects then break down the Core Domain into autonomous services: Entity, Value Object, Aggregate, Aggregate Root.

Dedicated Databases: Although shared databases provide some pragmatic advantages, for more sustainable, scalable and long term software development every microservice should have a dedicated database (private tables).

Deployment Automation: When deciding how to deploy it is important to implement a "build and release" automation structure to reduce lead time and make releases quicker.

Phase the Migration to Microservices: Monolithic architectures often involve a complex weave of repositories, deployment, monitoring, and other complex tasks so break down the migration into phases to avoid errors and gaps.

Bake in the Splitting System: One of the key best practices is to inspect the current monolithic structure to understand the components causing problems and transform this part into a microservice. Define the interactions and processes between different pieces as you split them into microservices until you have enough pieces in place to make the final switchover.

Observability: Utilize solutions that simplifies observability issues inherent with continuously monitoring so many individual services and pulls logs and application performance metrics into a centralized hub.

Service Mesh and Container Ingress: Using a service mesh for microservices deployments enables efficient handling of service discovery, traffic management, security authentication, and authorization for container-based applications no matter the size or geographic distribution of servers. Combined with a Container Ingress that provides North-South traffic management, including local and global server load balancing (GSLB), web application firewall (WAF) and performance monitoring, across multi-cluster, multi-region, and multi-cloud environments unlocks advanced container and microservices orchestration.

Deployment of Microservices

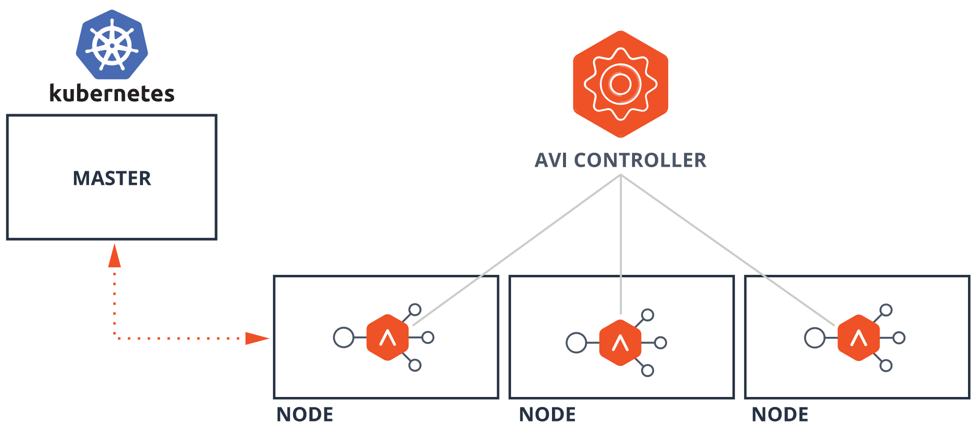
In a modern microservices architecture, the deployment of microservices plays an important role in the effectiveness and reliability of an application infrastructure. The following components of microservices guidelines should be considered in the deployment strategy:

- Ability to spin up/down independently of other microservices.
- Scalability at each microservices level.
- Failure in one microservice must not affect any of the other services.

Docker is a standard way to deploy microservices using the following steps:

- Package the microservice as a container image.
- Deploy each service instance as a container.
- Scale based on changing the number of container instances.

Kubernetes provides the software to build and deploy reliable and scalable distributed systems. Large-scale deployments rely on Kubernetes to manage a cluster of containers as a single system. It also lets enterprises run containers across multiple hosts while providing service discovery and replication control. Red Hat OpenShift is a commercial offering based on Kubernetes for enterprises.



Running a small amount or building hundreds of microservices and running thousands of instances presents different challenges. Instances should be able to increase when users increase and decrease when users decline. Open source helps with building microservices that run intelligently, give a clear view of the service instances that are running or going down, and manage complexity.

This article from opensource.com looks at some of the key terminologies in the microservices ecosystem and some of the open source software to build out a microservices architecture.

"Microservices architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API," according to authors Martin Fowler and James Levis in their article Microservices. "These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

Microservices Best Practices

Designing Microservice Architectures is challenging due to the vast amount of possible, custom application delivery solutions they can create which can also lead to complex issues that arise if they are not designed, deployed and managed correctly. Companies that are transforming from a monolithic architecture to a more modern microservice architecture, need to be cautious not to rush the design of microservices deployment which can end up doing more harm than good. Costs can quickly pile up if you are dealing w unexpected bugs, slow system updates, insufficient development teams or relying on hybrid hardware and software components.

Container Ingress Traffic Management

First of all, the key functionality of container ingress is traffic management, which includes routing the traffic from external sources into the cluster through an ingress gateway or out of the cluster through an egress gateway. This is called north-south traffic management.

Container ingress traffic management capabilities include:

- Ingress gateway with integrated IPAM/DNS, deny list/accept list and rate limiting
- L4-7 load balancing with SSL/TLS offload
- Automated service discovery and application map

Microservices Security

The next logical step of an application lifecycle is to secure the application, especially in the case of thousands of microservices. The connectivity is dynamic, and each service-to-service communication needs to be encrypted, authenticated and authorized.

Microservices security capabilities include:

- Zero trust security model and encryption
- Distributed WAF for application security
- SSO for enterprise-grade authentication and authorization

Microservices Observability

Observability in microservices is important because most enterprises replace monolithic applications incrementally. As microservices are introduced, there can be many different applications that need to communicate with each other and interact with the monolithic applications that remain. Microservices observability is key for understanding the complicated architecture and root-causing problems when failures happen. It allows for health checks with a broad view of application interactions.

Microservices observability capabilities include:

Real-time application and container performance monitoring with tracing

Big data driven connection log analytics

Machine learning-based insights and app health analytics

Ready to See Avi in Action?

SCHEDULE A DEMO

Why Avi

- What We Do
- Platform Overview
- Platform Architecture

Solutions

Modern Load Balancing
- Upgrade from F5
- SDN: Cisco ACI
- Cisco ACE Migration

VMware Ecosystem
- VMware NSX
- VMWare Horizon
- VMware vSphere

Public / Private Cloud
- Microsoft Azure
- Amazon Web Services
- Google Cloud Platform
- OpenStack

Container Ingress
- Kubernetes

Products

Avi Vantage
- Software Load Balancer
- Intelligent WAF
- Container Ingress

Avi SaaS
- Overview

Customers

- Technology
- Financial
- E-Commerce
- Media
- Other

Partners

Resources

Technology Partners

- Cisco
- Red Hat
- Amazon Web Services
- Microsoft Azure

Resource Center

- Content Library
- Webinars
- Blog

Technical Help

- Knowledge Base
- Professional Services
- Support
- Community

Education

- Glossary
- Workshops

Avi 101

- Load Balancing
- Web Application Firewall
- Microservices & Containers
- Application Delivery Controller

Company

- Events
- Contact Us
- Privacy Policy
- California Privacy Rights

Privacy Policy | California Privacy Rights

AVI