KUBERMATIC
KubeLB

# Cloud Native Multi Tenant Load Balancing

## Summary

Throughout the years, application architecture has evolved from client-server to service-oriented to cloud-native microservices-based architectures. This evolution significantly impacts the application development methods and approaches to scalability, security, and, most importantly, application delivery. Unfortunately, the industry's current state of Application Delivery Controller (ADC) solutions, also known as load balancers, falls short.

The demand for services has evolved, and what once required just one service now necessitates multiple distinct services. In this new era of service explosion, it will be challenging for IT teams to effectively handle the lifecycle management of applications unless the modern-day ADC is “microservices aware” and has the appropriate automation, enabled via APIs, automation, and orchestration frameworks, to provision, configure and manage every microservice.

This paper describes how application architectures have evolved and how KubeLB's distributed microservices can dramatically reduce the operational impact of microservices-based application architectures.

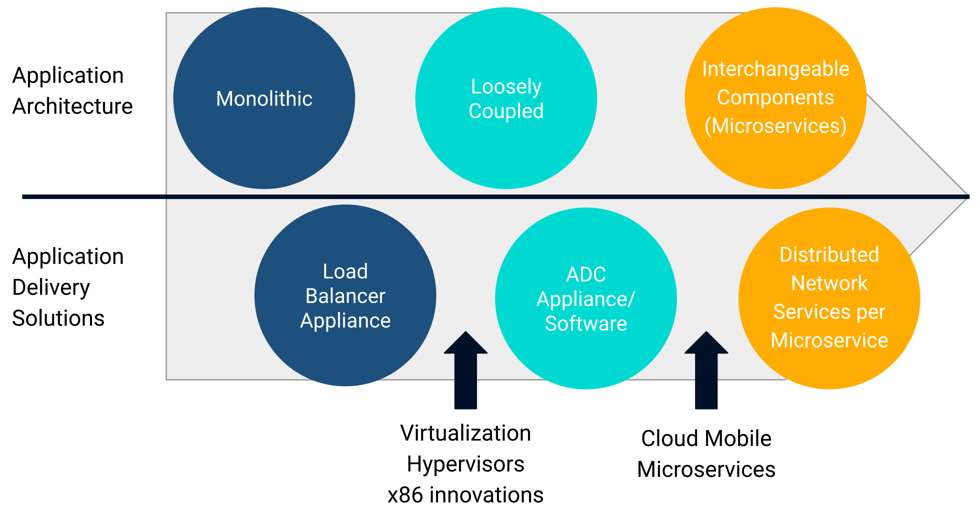

Application Architecture
Monolithic
Loosely Coupled
Interchangeable Components (Microservices)
Application Delivery Solutions
Load Balancer Appliance
ADC Appliance/ Software
Distributed Network Services per Microservice
Virtualization Hypervisors x86 innovations
Cloud Mobile Microservices

*Figure 1: Evolution of Application Architectures vs Application Delivery Solutions*

## The Monolithic Architecture

From the inception of web application development, the predominant enterprise application architecture involved bundling all the application's server-side components into a single unit. Numerous enterprise Java applications, for instance, often consist of a solitary WAR or EAR file.

Consider developing an online store that accepts customer orders, verifies inventory and available credit, and handles shipments. This application comprises various components, including the StoreFront UI, responsible for the user interface and services dedicated to managing the product catalog, processing orders, and handling customer accounts. These services utilize a shared domain model, including Product, Order, and Customer. Despite its logically modular design, the application is deployed as a monolith. A single WAR file runs on a web container like Tomcat in Java.

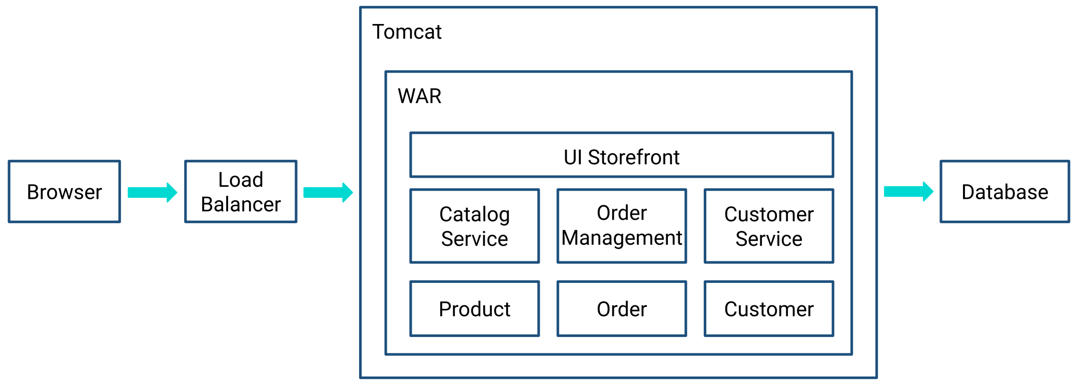


*Figure 2: Traditional monolithic application architecture*

This so-called monolithic architecture has several benefits. Developing monolithic applications is simple because IDEs and other development tools are oriented toward creating a single application. Testing and deploying them is also straightforward since everything coexists within a single application.

This approach works well for relatively small applications. Nevertheless, when dealing with complex applications, the monolithic architecture becomes cumbersome. It also poses challenges in experimenting with and incorporating new technologies. For instance, attempting a new infrastructure framework often requires rewriting the entire application, which becomes risky and impractical. Consequently, the technology choices made at the project's outset tend to become fixed. Additionally, scaling individual portions of the application is challenging, if not impossible, as all the application code operates within the server process on the server. To deploy new changes to one application component, you have to build and deploy the entire monolith, which can be complex, risky, and time-consuming, require the coordination of many developers, and result in long testing cycles.

In cases where one service demands significant memory resources and another is CPU intensive, provisioning the server necessitates allocating sufficient memory and CPU capacity to accommodate the baseline load for each service. This expense can escalate, especially if each server requires substantial amounts of CPU and RAM, and the situation is further aggravated when load balancing is employed to scale the application horizontally. In other words, the monolithic architecture can’t be scaled to support large, long-lived applications. A huge monolithic application can quickly become a delicate house of cards where a fault in one minor part of the application can bring the whole system down.

## The cloud-native microservices-based Architecture

The cloud-native microservices-based architecture is designed to address the issues seen by the monolithic architecture. The services outlined in the monolithic application architecture are disassembled into distinct services and deployed independently on separate hosts. Each microservice is dedicated to a specific business function, solely encompassing the operations essential to that particular business function. This architecture utilizes tools like Kubernetes, ArgoCD, flux, and other cloud-native-related projects.

## This architecture has several benefits

1. Each service is relatively small, with a more understandable codebase for developers. This enhances developer productivity since we are now only focusing on a subset of the application; IDE is more efficient, and running and testing the application is less time-consuming.

2. Since each service is isolable and not dependent on other services, developers can work on a specific service in isolation without being dependent on different teams or silos within an organization. This makes continuous development, testing, and deployment easy and greatly attractive.

3. The biggest advantage of this architecture that affects all different silos within an organization; developers, operations, and management, is the ability to configure scaling, underlying hardware, and resource requirements(CPU, GPU, or memory intensive) per service. This can greatly enhance the throughput of these applications while reducing the cost incurred.

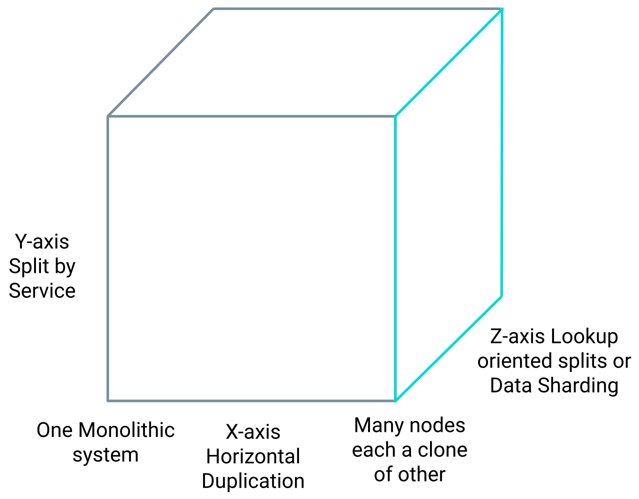


*Figure 3: Application scaling along X, Y & Z axes*

## Scaling Microservices along X, Y & Z axes

The most common representation of scaling an application is the "Scale Cube," a three-dimensional scalability model popularized by the book "The Art of Scalability." In this model of scalability, the three dimensions are X-axis, Y-axis, and Z-axis:

**#1** X-axis scaling: Horizontal scaling involves running multiple copies of the entire application behind a load balancer. Each instance of the application is referred to as a horizontal slice. This is the most common form of scaling for web applications.

**#2** Y-axis scaling: Vertical scaling involves splitting the application into different parts, each running on a separate machine. This is the most common form of scaling for microservices.

**#3** Z-axis scaling: Also known as functional decomposition, involves splitting the application into different parts, with each part running on a separate machine. This is the most common form of scaling for microservices.

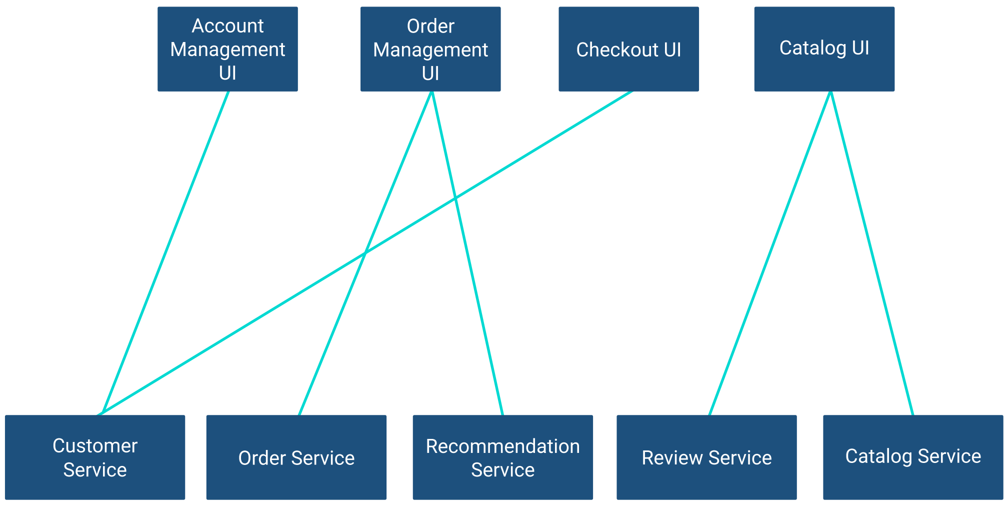


*Figure 4: Microservices based application architecture*

## Rise of Containerization and Orchestration Tools

As microservices became more prevalent, tools like Docker, Docker Swarm, and Kubernetes emerged to simplify deployment and scaling. Docker revolutionized containerization, enabling consistent environments across development and production. Docker Swarm built on this by providing native clustering and orchestration, easing X and Y axis scaling for more straightforward applications.

Kubernetes advanced these capabilities, offering robust orchestration, management, and scalability for complex, distributed microservices architectures. Its powerful features, such as automated scaling, self-healing, and rolling updates, addressed many challenges associated with deploying and managing microservices at scale, making it the de facto go-to platform for deploying and managing microservices in the industry.

## Challenges with Kubernetes

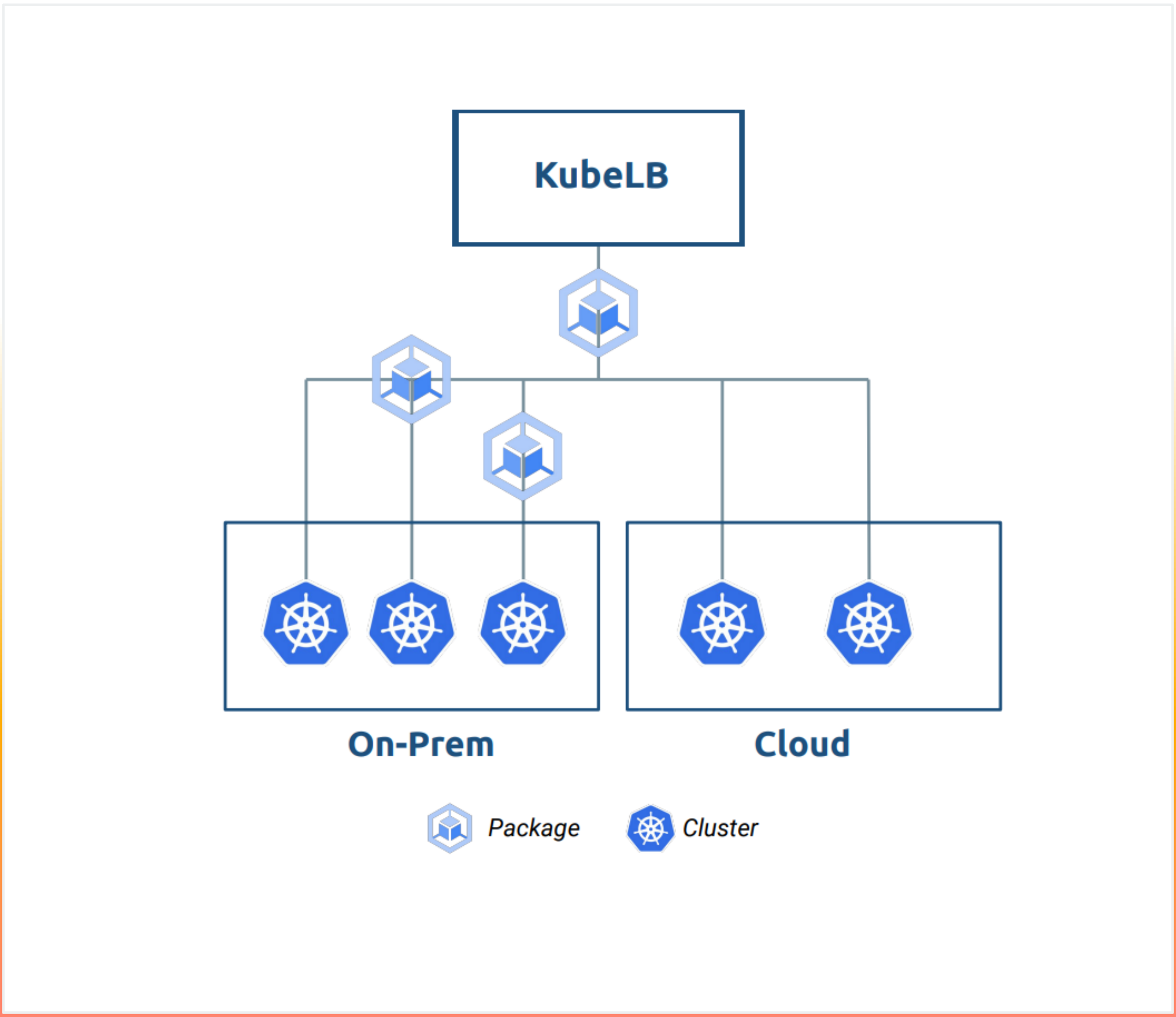
While Kubernetes significantly simplifies the orchestration and management of containerized applications, it introduces new challenges, particularly in multi-cluster and multi-tenant environments. Kubernetes provides interfaces for Layer 4 and Layer 7 load balancing in the form of Services and Ingresses. Still, they offer limited capabilities and depend on the cloud provider or ingress provider to implement them. This makes us dependent on the provider for advanced features like traffic splitting, traffic shaping, and observability. Managing network traffic, ensuring application security, and maintaining high performance across different clusters and clouds can be complex and resource-intensive. Traditional load balancers often struggle to keep up with the dynamic nature of microservices, leading to inefficiencies and increased operational overhead. These challenges highlight the need for a more advanced solution to effectively manage the data plane and optimize application delivery in cloud-native environments.

As applications grow in complexity and scale over time, some mission-critical applications may necessitate multi-cluster deployments. To address these challenges, enterprises began adopting multi-Kubernetes cluster architectures. Organizations achieved enhanced fault tolerance, improved performance, and better regulatory compliance by deploying multiple Kubernetes clusters across regions, availability zones, or cloud providers. This again introduced new challenges, such as managing various clusters, ensuring consistent policies, and providing seamless communication between applications hosted on multiple clusters. Kubernetes doesn't natively possess these capabilities and requires additional tools to address these challenges.

# Cloud Native Multi Tenant Load Balancing

## Introducing KubeLB

KubeLB is a software-based next-generation application delivery platform with integrated analytics, which provides secure, reliable, and scalable network services for cloud applications. At the heart of KubeLB is a revolutionary architecture based on SDN principles, separating the data plane from the control plane – an industry first for Application Delivery Controllers and Load Balancers. KubeLB enables seamless scaling of application delivery services within and across data centers and cloud locations while maintaining a single point of management and control. Load balancer operates as a service, so you can have multiple customers using the same software. It detects the customer environment and acts accordingly.

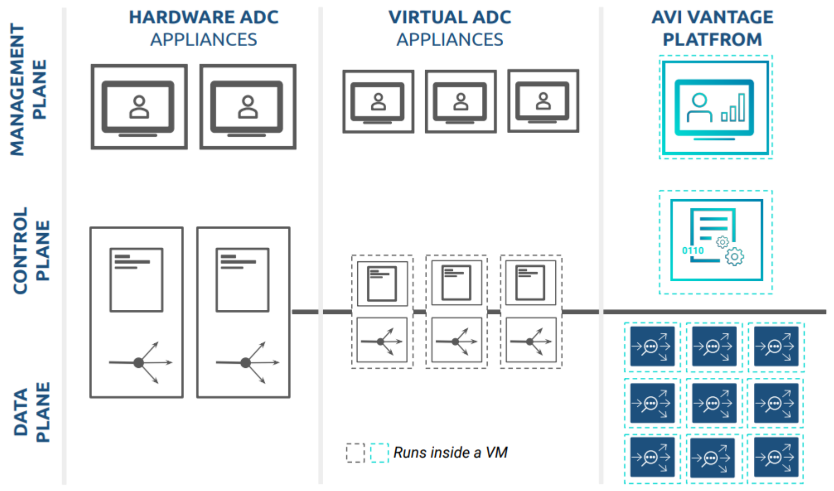


## Distributed Architecture

The distributed load balancers implemented by high-performance Cilium and Envoy provide comprehensive application delivery services such as load balancing, application acceleration, and application security. Cilium and Envoy can be co-located with applications within and across cloud locations and grouped for higher performance.

Using KubeLB's rich data, control, and management plane services, Cilium and Envoy can be placed close to the application's microservices and grouped for higher performance and faster client responses. The integrated data collectors gather end-to-end timing, metrics, and logs for each user-to-app transaction. These provide actionable insights about end-user experience, application performance, infrastructure utilization, and anomalous behavior, which can be used to better architect the application being served.

*Figure 6: Separation of data, control and management planes*



## Analytics-driven application delivery

As the demand for a particular microservice application grows, KubeLB's unique distributed architecture allows the KubeLB Service Engines to be scaled out automatically without human intervention.
KubeLB engines constantly monitor the traffic patterns of each microservice application. When a (customizable) threshold is met, the newly scaled-out Cilium and Envoy handle the increasing traffic load seamlessly.
Furthermore, the Inline Analytics engines can send a trigger based on ambient loads to scale up/down the backend microservices applications.
Finally, the distributed microservices architecture allows the Cilium and Envoy attached to each microservice to be scaled out independently of other microservices.

## Elastic Scale

The Elastic data plane of 'KubeLB' can dynamically scale out and scale in to meet the real-time requirements of microservice-based applications across 100s of tenants and 1000s of applications. Cilium and Envoy allow network services for each Microservice to be individually scaled out/in or up/down.

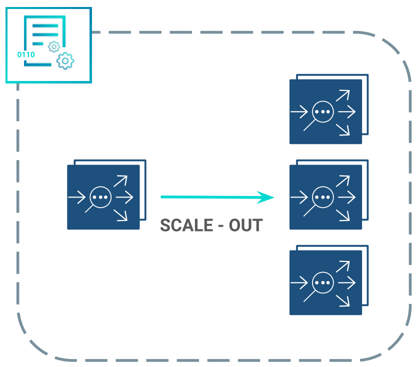


*Figure 7: As demand increases, KubeLB SE's are automatically created by KubeLB Controller*

## Application Affinity

Cilium and Envoy are placed close to microservices applications for best app performance and minimal traffic tromboning in the network. Whether microservices are inside a single physical server, in different servers but in a single data center, or even across different data centers, Cilium and Envoy automatically discover and locate themselves in the closest possible proximity to each microservice.

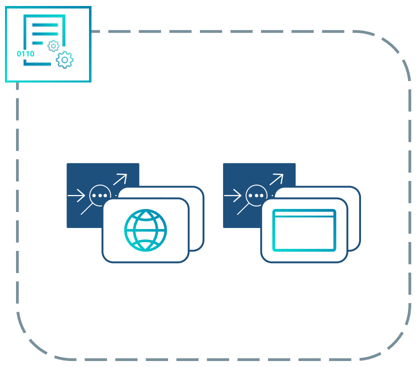


*Figure 8: KubeLB SE's can be collocated with each microservice within or across cloud locations*

# Cloud Native Multi Tenant Load Balancing

## Dataplane Isolation for Tenants and Applications

To avoid sharing appliances between critical applications, tenants and applications are allocated their own Service Engines for data plane isolation. This eliminates the 'noisy neighbor' problem wherein a rogue microservice or tenant could potentially impact the performance of an adjacent application. KubeLB's per-tenant, dedicated micro load balancers deliver true multi-tenant application services.

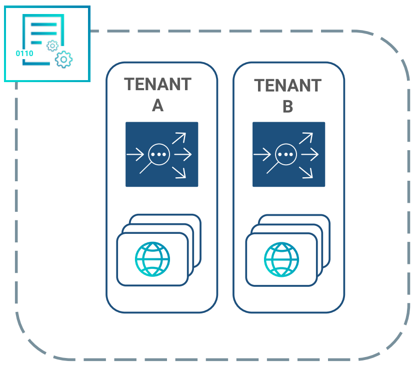


*Figure 9: KubeLB Service engines can be dedicated to microservices and/or tenants for true isolation*

## Programmability

All interactions with the KubeLB Controller are through native Kubernetes APIs, which enable native integration with Kubectl. DevOps automation tools like Crossplane, Terraform, or Ansible are also natively supported.

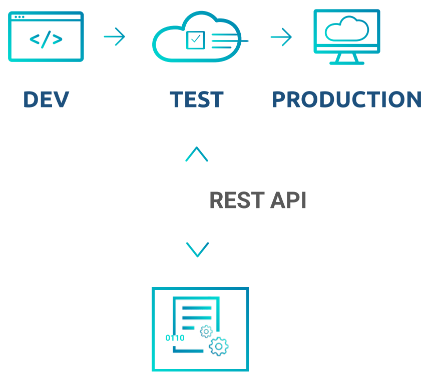


*Figure 10: KubeLB ADC has 100% features support by REST APIs*

## N-Way Active Redundancy

Using redundancy principles from web-scale datacenters, KubeLB provides N-Way Active-Active redundancy along with Active-Standby availability options.

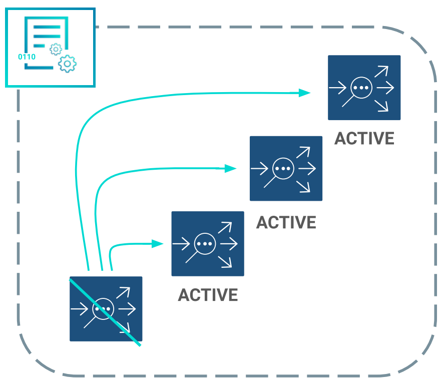


*Figure 11: KubeLB supports a N-way active redundancy model*

## Putting it all together:

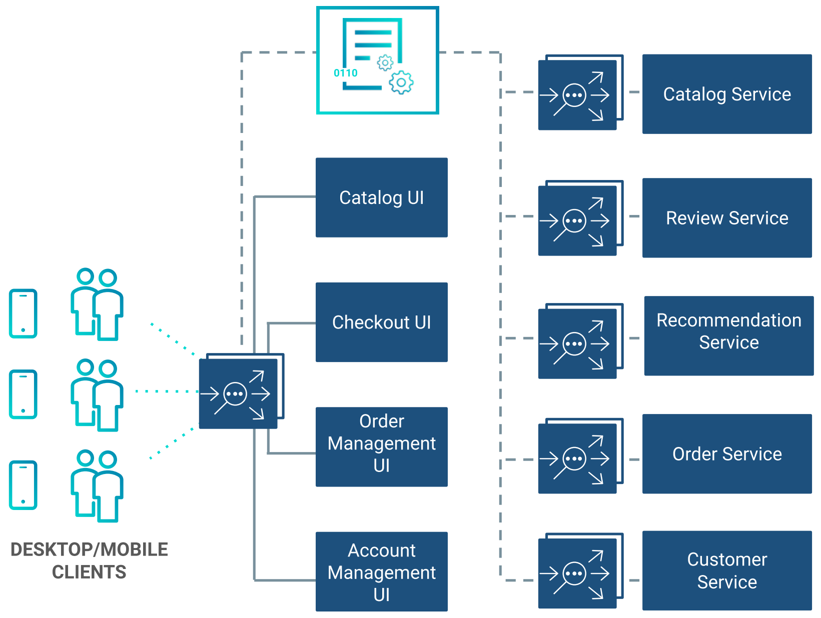


*Figure 12: KubeLB Service Engines attached at various microservices controlled and managed by KubeLB Controller*

Each group of Cilium and Envoy can be associated with a specific tenant, so in a multi-tenant environment, traffic for a particular application is isolated to that tenant's group of Cilium and Envoy.

KubeLB can manage multiple groups of Cilium and Envoys. Kubernetes's role-based access control mechanism ensures that users logged into a particular tenant can only view the details of that particular tenant.

## How Load Balancers/ADCs Need to Evolve – the Mantle for True Software-Defined Networks

Here are our views on the specific steps required to evolve application delivery for modern data center requirements:

# Cloud Native Multi Tenant Load Balancing

**Step No. 1**

In terms of architecture, there needs to be a complete and true separation of control and data plane within the ADC with the ability to distribute data plane resources dynamically across different hardware platforms and public/private clouds, exactly how application microservices can be distributed.

**Step No. 2**

ADCs must achieve the “application affinity” concept based on the app-world “processor affinity” concept, where resources are aligned/pinned for specific functions. This approach offers two significant advantages. First, the microservice will enhance the application response time by colocating the ADC resource alongside. Second, this tight alignment (affinity) enables ADC resources to achieve automatic microservices lifecycle management without manual intervention, significantly reducing management complexity.

**Step No. 3**

Achieving data plane independence (isolation) to enable multi-tenancy, especially in cloud environments. This aligns with how microservices can operate and be changed independently of each other without disrupting other microservices or the “no noisy neighbor” impact.

**Step No. 4**

Fulfilling the self-service programmability and efficiency promises of SDN. Most, if not all, ADC vendors today support REST (representational state transfer), the protocol of choice in hyperscale web services. However, only through a true control/data plane separation and the complete centralization of the control functions can the ADC achieve the real promise of SDN by enabling one-to-one communications between its controller and the application’s control elements through RESTful APIs.

# Cloud Native Multi Tenant Load Balancing

## Summary

The architecture of how applications are developed today has evolved from purpose-built, monolithic ("shrink-wrapped") code and products to a tightly federated collection of modular and reusable micro-services. It's as if app developers moved to using a common set of Lego blocks to build any number of web-based apps only limited by their imagination. The move to microservice-app development for networking teams means their existing assumptions around traffic patterns, load balancing scale, and service requirements are no longer valid. An increased level of network-wide intelligence and a new application delivery architecture that mirrors the microservice apps are required. KubeLB is an elastically scalable load balancer with a distributed data plane that can span, serve, and scale with apps across various on-premise and cloud locations. The distributed data plane empowers customers to obtain application affinity at the application microservice levels, thus significantly enhancing the overall application performance. In addition, the clean separation of planes also enables the creation of a unified, centralized control plane that significantly alleviates the operational complexity associated with integrating, operating, and managing each ADC appliance across locations individually.

In summary, KubeLB is a highly flexible, cost-focused, scalable, and efficient load balancer, particularly suited to multi-tenant service providers.

## Case-study

### How can a flexible Load Balancer defined by a software adjust over time to evolving changes in application architectures?

**Stage 1**

In the early stages, the company focuses predominantly on low complexity and overhead, leading to rapid software development and many features. The need to rush to market to deliver proof of concept features to customers implies developers typically do not have the luxury of designing the application for scalability, high availability, and redundancy. The application can typically be deployed on web and database servers. With KubeLB ADC, high availability and scalability can be quickly achieved by running the application on a pair of web servers behind a pair of load balancers. This also keeps the operation costs to a minimum.

## Stage 2

As the demand for the business grows, the company can scale quickly by adding more resources (X-axis scaling) behind KubeLB ADC. As the number of web servers grows, static content management can become a challenge but can be mitigated using caching engines on the KubeLB ADC. As the popularity of the company and its product grows, the need to scale and perform better leads to rearchitecting the application to break it into smaller applications along the lines of services/functions. Database partitions start to make sense, and partitions evolve along geographical locations, names, etc.

## Stage 2

With KubeLB’s future-proof security-focused design, the same ADC used on day 1 of the application deployment can still be used as the traffic to the application grows. Once the need to serve the application grows, so does the SLA needed to maintain it, and the application gets deployed in multiple locations. One instance of KubeLB can serve applications from a local data center and in the cloud. KubeLB’s centralized control and management interface makes managing all the load-balanced applications easy.

**Not sure about where to start? We'll meet you where you are and provide a customized solution to meet your specific needs.**

**Visit our Website**