

1) The Problem Serverless Computing Solves and Its Use Cases

Serverless computing aims to simplify application deployment by removing the need for developers to manage servers or infrastructure. In traditional microservice deployment on Kubernetes, teams must handle container orchestration, scaling, and resource allocation. Serverless platforms, such as AWS Lambda or Azure Functions, automatically handle these tasks, scaling functions up or down based on demand and charging only for actual execution time. This reduces operational overhead and cost for workloads with unpredictable traffic.

Serverless is clearly better for event-driven or intermittent workloads, such as image processing or user-triggered notifications, where resources are needed only briefly. However, it may not be ideal for long-running or latency-sensitive applications, such as real-time trading systems or continuous background services, because cold starts and limited runtime durations can cause delays and inefficiency. In short, serverless is excellent for dynamic scalability but less suitable for performance-critical systems.

2) Advantages of Using a Service Mesh (e.g., Istio) in Microservices

A service mesh like Istio enhances how microservices communicate by adding an intelligent layer that manages traffic, security, and monitoring. While Kubernetes networking provides basic service discovery and routing, it does not handle advanced communication features automatically. A service mesh introduces sidecar proxies that manage requests between services, enabling load balancing, traffic encryption (mTLS), retry policies, and circuit breaking without modifying application code.

It also improves observability by collecting detailed metrics, logs, and traces, allowing administrators to monitor performance and diagnose failures easily. Another advantage is fine-grained security control, since Istio enforces authentication and authorization policies across all services. Overall, a service mesh provides consistent, centralized management of service-to-service communication, making large-scale microservice architectures more reliable, secure, and maintainable than relying solely on Kubernetes networking.

3) What a Sidecar Proxy Does and Why It Is Needed in a Service Mesh

A sidecar proxy, such as Envoy in Istio, is a lightweight network proxy deployed alongside each microservice instance within a service mesh. It intercepts all inbound and outbound traffic for that service, managing communication, security, and observability without requiring changes to the application code. This design offloads complex networking tasks from developers, who can focus on business logic rather than communication management.

The sidecar proxy is essential in a service mesh because it provides a consistent and centralized way to control traffic between microservices. It handles encryption (mTLS), traffic routing, load balancing, and policy enforcement automatically. By standardizing these functions across all services, sidecar proxies enable fine-grained control, monitoring, and fault tolerance, ensuring that microservices remain secure, reliable, and easy to operate at scale. Essentially, the sidecar forms the data plane of the service mesh, while Istio's control plane manages its configuration.

4) Traffic Management Features Provided by Istio and Their Usefulness

Istio offers a wide range of traffic management features that help control how requests flow between microservices. These include intelligent routing, load balancing, traffic splitting, circuit breaking, and fault injection. Such features allow operators to shape traffic dynamically without modifying the application.

For example, traffic splitting enables gradual rollout of new versions through canary

deployments, where only a small percentage of user requests are sent to the new service version. This reduces the risk of full-scale failures during upgrades. Another useful feature is circuit breaking, which prevents cascading failures by stopping requests to an overloaded or failing service. It helps maintain system stability under heavy load or partial outages. Together, these capabilities make Istio a powerful tool for managing microservices traffic safely and efficiently in production environments.

5) How Knative Serving Enables Autoscaling and Its Triggers

Knative Serving provides automatic scaling for container-based applications deployed on Kubernetes. It monitors the incoming request traffic and dynamically adjusts the number of running instances (pods) based on demand. When traffic increases, Knative automatically scales up the number of pods to handle more requests. When demand drops, it scales down and can even scale to zero when there are no active requests, freeing cluster resources and reducing costs.

Autoscaling decisions are made using metrics such as requests per second (RPS) or concurrency per pod. Knative uses a component called the Autoscaler (KPA or HPA) to determine scaling behavior. For instance, if each pod is receiving more concurrent requests than the configured target, Knative will launch additional pods; when requests fall below that threshold, it will remove pods gradually. This makes Knative Serving ideal for serverless workloads, where traffic can be unpredictable or bursty.

6) The Role of Knative Eventing and Its Support for Event-Driven Architectures

Knative Eventing provides a framework for building event-driven applications on Kubernetes by managing how events are produced, delivered, and consumed. It allows services to react to events from various sources, such as cloud services, message queues, or other applications, without requiring tight coupling between components.

The key role of Knative Eventing is to enable asynchronous communication through components like Brokers, Triggers, and Channels. A Broker receives and distributes events, while Triggers define filtering rules to route specific events to subscribers. For example, a new file uploaded to cloud storage can automatically trigger a function to process it.

By standardizing event delivery and routing, Knative Eventing supports scalable, loosely coupled architectures that respond dynamically to real-world events. This model enhances flexibility, reusability, and resilience in modern cloud-native applications.

7) How Knative Leverages Kubernetes Primitives for a Serverless Experience

Knative builds on top of Kubernetes primitives, such as Deployments, Services, Ingress, and the Horizontal Pod Autoscaler (HPA), to provide a serverless experience while hiding much of their complexity from developers. Under the hood, Knative automatically creates and manages these Kubernetes resources to handle scaling, routing, and load balancing.

For example, when a developer deploys a Knative service, Knative generates the necessary Deployment and Service objects and sets up an autoscaler to adjust the number of pods based on incoming traffic. It also manages revisions of an application automatically, enabling version control and traffic splitting without manual configuration.

By abstracting these low-level Kubernetes components, Knative allows developers to focus solely on writing and deploying code instead of managing infrastructure. This abstraction provides faster deployment, easier updates, and automatic scaling to zero, making the developer experience more efficient and cloud-native without sacrificing flexibility or portability.

8) The Main Function of an InferenceService in KServe and Its Role in ML Deployment

In KServe, an InferenceService is the core abstraction used to deploy, manage, and serve

machine learning (ML) models on Kubernetes. It defines the complete lifecycle of an ML model from loading and scaling to versioning and serving predictions, using a simple declarative configuration.

Instead of manually creating containers, deployments, and networking rules, developers can define an InferenceService YAML specifying the model's storage location and serving runtime (e.g., TensorFlow, PyTorch, or SKLearn). KServe then automatically provisions the underlying Kubernetes resources and exposes a consistent API endpoint for inference requests.

This abstraction greatly simplifies ML deployment by standardizing the serving process across frameworks and automating complex tasks such as autoscaling, model rollout, and monitoring. It enables data scientists and engineers to deploy models quickly and reliably without deep knowledge of Kubernetes operations, making ML serving both efficient and production-ready.

9) Data Flow and Responsibilities in a KServe Production ML Workflow

In a production KServe workflow, data typically flows through several layers before producing a prediction. When an HTTP request arrives, Istio first handles traffic routing and security. It manages load balancing, TLS encryption, and retries to ensure the request reaches the correct service endpoint. The request is then passed to Knative Serving, which manages the serverless infrastructure, scaling pods up or down based on demand and routing traffic to the active KServe InferenceService.

Within KServe, the request is directed to the model server (e.g., TensorFlow Serving or TorchServe), which processes the input data and generates a prediction. Finally, Kubernetes orchestrates all underlying containers, networking, and resource allocation.

Potential latency bottlenecks can occur at several points: Istio's proxy layer (due to encryption and routing), cold starts in Knative (when scaling from zero), and model loading time in KServe, especially for large or complex ML models.

10) Using Istio Traffic Routing for Canary Deployments and A/B Testing in Knative or KServe

Istio provides powerful traffic routing capabilities that can be leveraged for canary deployments and A/B testing in Knative or KServe environments. With weighted routing, Istio can gradually shift a small percentage of traffic to a new model version or service revision, allowing safe testing under real-world conditions. Retries and circuit breaking add resilience by preventing overloads or cascading failures if the new version behaves unexpectedly.

This automated traffic management enables controlled rollouts with minimal downtime and quick rollback options if issues arise. It is particularly useful in ML workflows, where new models must be tested for performance and accuracy before full deployment.

However, compared to manual rollout strategies, Istio-based routing introduces added complexity and requires careful configuration to avoid misrouting or inconsistent metrics. Despite this, its automation, safety, and observability make it far more efficient and reliable for production-scale deployments.