**1. What problem does serverless computing aim to solve compared to traditional microservice deployment on Kubernetes? Give one example where serverless is clearly better, and one where it may not be.**

(1)**Serverless computing aims to solve the problem of operational overhead and inefficient resource utilization.** Traditional microservices on Kubernetes require developers to manage the underlying infrastructure (like pods, scaling rules, and service meshes), and they often run continuously, consuming resources even when idle. Serverless abstracts away the infrastructure, providing automatic, event-driven scaling that can scale down to zero when not in use, leading to significant cost savings and developer productivity.

(2)**Clearly Better Example:** An image processing API that is triggered by user uploads. It experiences sporadic, unpredictable traffic. With serverless (e.g., Knative), it scales to zero when idle (costing nothing) and automatically scales up instantly to handle bursts of uploads. This is more cost-effective than running a continuously active Kubernetes deployment.

**Where It May Not Be Better:** A long-running WebSocket connection for a real-time chat application. Serverless functions have execution timeouts and are not designed for persistent connections. The constant cold starts from scaling from zero would introduce unacceptable latency, making a traditional always-on microservice more suitable.

**2. What are the advantages of using a service mesh (like Istio) for managing microservices communication instead of relying only on Kubernetes networking?**

**Fine-grained Traffic Management:** Enables intelligent routing (e.g., canary releases, A/B testing based on HTTP headers), retries, timeouts, and circuit breaking, which are not possible with basic Kubernetes Services.

**Enhanced Observability:** Provides detailed metrics, logs, and traces for all service-to-service communication, offering deep insights into performance and dependencies without changing application code.

**Stronger Security:** Offers automatic mutual TLS (mTLS) to encrypt all traffic between services, and provides policy-based access control.

**3. Explain what a sidecar proxy (such as Envoy in Istio) does. Why is it needed in a service mesh?**

(1)A sidecar proxy is a companion container deployed alongside the main application container in a pod. It intercepts all incoming and outgoing network traffic for the application. In Istio, the Envoy proxy is the sidecar.

**Its main functions are to:**

**Intercept Traffic:** Capture all network communication to and from the application.

**Implement Policies:** Enforce traffic rules (routing, retries), security (mTLS), and collect telemetry data as defined by the service mesh control plane.

(2)**It is needed to decouple application business logic from networking concerns.** Without it, each microservice would need to implement complex logic for routing, security, and observability, leading to code bloat and inconsistency. The sidecar pattern centralizes this cross-cutting functionality, making it easier to manage and update across all services.

1. **What kind of traffic management features does Istio provide? Give two examples of how they can be useful in production systems.**

(1)Istio provides powerful traffic management features including:

**Canary / Weighted Traffic Shifting**

**Request Routing based on HTTP headers, URIs, etc.**

**Fault Injection (e.g., delay, abort)**

**Retries with configurable policies**

**Timeouts**

**Circuit Breaking**

**Mirroring**

(2)Two useful examples:

**Canary Releases:** You can route 1% of user traffic to a new version of a service (the "canary") while 99% goes to the stable version. This allows you to validate the new version with real users with minimal risk. If the canary shows errors, you can instantly route traffic away from it.

**Resilience with Retries and Circuit Breaking:** If a service becomes slow or starts failing, Istio can automatically retry failed requests on a different service instance. If the failure rate becomes too high, the circuit breaker "opens," halting requests to that service to prevent cascading failures and allow it to recover.

**5. Explain how Knative Serving enables autoscaling for an application. What triggers scaling up and scaling down?**

(1**)Knative Serving enables autoscaling through its Knative Pod Autoscaler (KPA).** It scales the number of application pods based on incoming request traffic (concurrency).

(2)**Scaling Up (To-N):** The primary trigger is request concurrency. When the number of concurrent requests per pod exceeds a predefined threshold (default is 100), Knative will create new pods to handle the increased load. It can scale up very aggressively to handle sudden traffic spikes.

**Scaling Down (To-Zero):** The trigger is the absence of traffic. If a pod has zero requests for a stable window (default 60 seconds), Knative will terminate it. This is the "scale to zero" feature that defines the serverless experience and optimizes cost.

**6. What is the role of Knative Eventing, and how does it support event-driven architectures?**

(1)**The role of Knative Eventing is to enable event production and consumption between decoupled application components.** It supports event-driven architectures by providing a set of APIs for:

Routing Events: Delivering events from sources (like Kafka, GCP Pub/Sub, or HTTP) to interested consumers (your serverless functions or microservices) using constructs like Brokers and Triggers.

Managing Event Flow: It allows for filtering, fan-out (sending one event to multiple services), and ensuring reliable delivery.

(2)It supports event-driven architectures by allowing developers to easily create applications that react to events from various systems without needing to manage the underlying messaging infrastructure. Your service simply declares what events it is interested in, and Knative Eventing handles the delivery.

**7. How does Knative leverage Kubernetes primitives to provide a serverless experience? Discuss which components of Kubernetes (e.g., Deployments, Services, Horizontal Pod Autoscaler) are abstracted away and how this abstraction benefits developers.**

(1)Knative builds upon and abstracts key Kubernetes primitives to create a higher-level, developer-friendly API for serverless workloads.

(2)Abstracted Kubernetes Components:

**Deployment & ReplicaSet:** Knative's Service resource manages the lifecycle of your application, automatically creating and managing the underlying Kubernetes Deployments and ReplicaSets.

**Service & Ingress:** Knative automatically creates Kubernetes Services and Ingress resources (or Istio VirtualServices) to handle network routing and traffic splitting for revisions.

**Horizontal Pod Autoscaler (HPA):** Knative replaces the standard HPA with its own more advanced Knative Pod Autoscaler (KPA) which supports scale-to-zero and request-driven scaling.

(3)Benefits of Abstraction:

Simplified Developer Experience: Developers define what they want to run (their container image) using a simple Knative Service YAML, not how it should be deployed and managed. They don't need to be Kubernetes experts.

Focus on Code: It allows developers to focus purely on writing business logic without worrying about the operational complexities of deployments, service meshes, and scaling policies.

**8. In KServe, what is the main function of an InferenceService, and how does it simplify deploying ML models?**

(1)The main function of an InferenceService in KServe is to provide a unified and high-level API for deploying and serving ML models at scale. It is a custom resource that encapsulates the entire serving stack for a model.

(2)It simplifies ML model deployment by:

**Abstraction:** It abstracts away the underlying complexities of Knative, Istio, and Kubernetes. You don't need to create separate Knative Services, Istio VirtualServices, etc.

**Standardization:** It provides a single, consistent manifest to define everything: the model artifact, the serving runtime (e.g., TensorFlow, PyTorch, Triton), resource requirements, scaling, and canary rollout strategies.

**Serverless Benefits:** It automatically leverages Knative for auto-scaling (to zero and to-N), providing cost-efficiency and handling traffic spikes.

**9. In a production ML workflow using KServe, describe how data moves from an incoming HTTP request to a model prediction response. Which layers (Knative, Istio, KServe, Kubernetes) handle which responsibilities, and where could latency bottlenecks occur?**

(1)Data Flow:

**Istio/Istio Ingress Gateway:** The request first hits the Istio Ingress Gateway. It handles L4/L7 load balancing, TLS termination, and routes the request to the correct KServe InferenceService based on the host/path.

**Knative Serving:** The request is received by the Knative Activator (if the service is scaled to zero) or the active pod. Knative manages the request queue and autoscaling. If scaled to zero, the "cold start" latency is incurred here as a new pod is spun up.

**KServe InferenceService:** Inside the pod, the KServe model server (e.g., KServe's own server, Triton, etc.) loads the model and executes the inference logic on the request data.

**Kubernetes:** Provides the fundamental orchestration layer—scheduling the pod, managing resources (CPU, GPU, memory), and providing the network namespace.

(2)Potential Latency Bottlenecks:

**Cold Start (Knative Layer):** The most significant bottleneck when scaling from zero. The delay includes pod scheduling, container startup, and model loading.

**Model Loading (KServe Layer):** For large models, loading them from a network disk into memory can take time.

**Inference Execution (KServe Layer):** The actual computation on the model, especially for large, complex models or on under-provisioned hardware (e.g., insufficient CPU/GPU).

**Network Latency (Istio/Kubernetes Layer):** Although usually minimal, network hops between the gateway, activator, and pods can add latency.

**10. How can Istio’s traffic routing capabilities (e.g., weighted routing, retries, circuit breaking) be used to support canary deployments or A/B testing in Knative or KServe environments?**

**In Knative/KServe, each new configuration creates a new "Revision." Istio's traffic routing (via VirtualService) can be used to split traffic between these revisions.**

(1)**Canary Deployment:** You can direct 90% of traffic to the stable revision and 10% to the new canary revision. You can then gradually increase the traffic to the canary (50/50, 100%) while monitoring its health and metrics.

**A/B Testing:** You can route traffic based on HTTP headers (e.g., user-group: beta-testers) to send a specific user segment to the new revision (B) while the rest go to the current one (A). This allows for feature testing with a controlled audience.

(2)**Pros compared to Manual Rollout** (e.g., using native Kubernetes deployment strategies):

**Precision & Flexibility:** Fine-grained, percentage-based traffic control (e.g., 1%, 5%) and intelligent routing based on request properties, which is difficult with manual replica count adjustments.

**Safety & Observability:** Easy to instantly shift traffic away from a faulty version. Combined with Istio's metrics, it provides clear insight into the performance of each version.

**Decoupling:** Traffic splitting is decoupled from the deployment itself, allowing for more complex rollout strategies.

**Cons:**

**Complexity:** Introduces additional complexity by requiring knowledge and management of Istio's VirtualService and DestinationRule resources.

**Overhead:** Adds another layer (the service mesh) to the stack, which has a small resource footprint and operational cost.