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.**

Serverless computing aims to eliminate the operational burden of managing infrastructure, scaling, and server provisioning that developers face in traditional microservice deployments on Kubernetes. In Kubernetes, teams must define pod replicas, manage deployments, handle scaling rules, and monitor cluster utilization manually.

In contrast, serverless platforms like AWS Lambda or Knative automatically manage all of these aspects — provisioning resources only when needed and scaling them down to zero during idle times. This leads to better cost efficiency and developer productivity.

Example where serverless is better: Event-driven data processing tasks, such as automatically resizing images after an upload or processing IoT events. These workloads are unpredictable and benefit from serverless elasticity and cost-per-execution pricing.

Example where serverless may not be better: Long-running, stateful workloads, such as continuous video streaming or real-time trading systems. Frequent cold starts and state persistence make serverless architectures inefficient for these cases.

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

A service mesh like Istio provides a dedicated infrastructure layer for managing service-to-service communication within microservices. Unlike native Kubernetes networking, which primarily focuses on service discovery and load balancing, a service mesh introduces advanced capabilities such as:

Traffic management: Fine-grained control over request routing, retries, and timeouts.

Security: Mutual TLS (mTLS) for encrypted communication and identity-based authentication between services.

Observability: Distributed tracing, logging, and metrics collection without modifying application code.

These features help operators achieve consistent policies, reliable communication, and better visibility across large-scale microservice deployments.

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

A sidecar proxy, such as Envoy in Istio, is a lightweight network proxy that runs alongside each service instance (in the same pod). Its main purpose is to intercept and manage all inbound and outbound traffic for that service. This design offloads concerns like routing, encryption, retries, and metrics collection from the application code.

In a service mesh, sidecar proxies are essential because they provide a consistent way to enforce global policies. For example, they can ensure every request between services is encrypted with mTLS, apply rate limits, or collect telemetry data for monitoring. Without sidecars, these capabilities would need to be manually implemented in every service, increasing complexity and inconsistency.

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

Istio provides a comprehensive suite of traffic management features that allow fine control over how requests flow between microservices. Two of the most important features include:

Traffic splitting: Allows gradual rollout of new versions by splitting traffic between the old and new service versions (for example, 10%, 50%, then 100%). This is commonly used in canary deployments to reduce release risk.

Fault injection and retries: Lets developers simulate network failures or delays to test system resilience and automatically retry failed requests under controlled conditions.

In production systems, these features enhance reliability, allow safe experimentation, and reduce downtime during updates.

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

Knative Serving enables autoscaling through continuous monitoring of incoming HTTP request metrics such as concurrency and request rate. It uses an internal autoscaler component (KPA – Knative Pod Autoscaler) that dynamically adjusts the number of pods based on demand.

When traffic increases and request latency grows, Knative automatically scales up more pods to handle the load. When the system detects inactivity over a certain period, it scales the service down to zero, completely freeing resources. This behavior provides both high availability under load and cost savings during idle periods — a key aspect of serverless efficiency.

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

Knative Eventing provides a flexible event-driven framework that allows services to communicate asynchronously using events rather than direct requests. It defines key components such as Event Sources, Channels, Brokers, and Triggers, which together decouple event producers from consumers.

This model enables scalable and modular architectures where multiple services can react to the same event without being tightly integrated. For example, when a file is uploaded to cloud storage, one service could process metadata while another generates a thumbnail — all through event subscriptions. Knative Eventing makes it easy to build such loosely coupled, event-driven systems on top of Kubernetes.

1. **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.**

Knative leverages core Kubernetes primitives such as Deployments, Services, and the Horizontal Pod Autoscaler (HPA) but abstracts them behind higher-level constructs like Revisions and Routes.

Deployments and Services are used under the hood to manage application pods and expose them to the network.

HPA logic is extended by Knative’s own autoscaler (KPA) for request-based scaling.

Routes and Revisions enable version tracking and traffic management automatically.

This abstraction allows developers to deploy and manage serverless workloads without needing to understand Kubernetes internals. They can focus purely on writing code or containers while Knative automates scaling, routing, and version control, greatly improving development efficiency and reducing operational overhead.

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

In KServe, an InferenceService is the central abstraction that manages the complete lifecycle of a machine learning model serving deployment. It defines how a model is loaded, exposed, and scaled in a Kubernetes-native way.

An InferenceService standardizes the deployment process across frameworks (such as TensorFlow, PyTorch, XGBoost) and automatically handles model server provisioning, networking, autoscaling, and health monitoring. This allows data scientists to deploy models using simple YAML configurations without worrying about infrastructure details, ensuring consistency and repeatability across environments.

1. **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?**

In a production ML workflow using KServe, the flow from an HTTP request to a model prediction response typically involves several layers:

Istio manages ingress routing, load balancing, and enforces mTLS security between services.

Knative Serving handles autoscaling, request buffering, and revision routing for model versions.

KServe executes the actual model inference by invoking the model container or runtime (for example, TensorFlow Serving, Triton).

Kubernetes orchestrates pods, manages node resources, and provides persistent networking and storage.

Latency bottlenecks can occur at several points — cold starts in Knative (when scaling from zero), proxy overhead in Istio (due to traffic interception), or slow model loading in KServe (for large models). Optimizing caching and keeping warm replicas can help reduce these delays.

1. **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? Discuss the pros and cons compared to manual rollout strategies.**

Istio’s traffic routing capabilities, including weighted routing, retries, and circuit breaking, play a key role in enabling progressive delivery strategies such as canary deployments and A/B testing.

For example, during a canary release, Istio can direct only 5% of user traffic to a new version of a service while the remaining 95% goes to the stable version. If no errors occur, traffic weight can be gradually increased. Similarly, circuit breakers can prevent cascading failures by stopping traffic to unstable versions.

Pros: Safer rollouts, granular traffic control, real-time rollback, and automated monitoring integration.

Cons: Requires additional configuration complexity and may introduce slight latency due to proxy routing and telemetry collection.

Compared to manual rollout strategies, this approach minimizes human error, improves observability, and allows precise control over deployments, making it ideal for large-scale, production-grade environments.