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 erases the need to forecast capacity, size clusters, or keep any replica warm—you ship code and the platform starts, scales, and stops containers for you while charging only for actual CPU-memory seconds, which is perfect for sporadic, latency-tolerant work; on the flip side it surrenders control over the runtime, introduces cold-start jitter, and caps execution time, so a real-time gaming shard that must hold WebSockets for thirty minutes and answer in <20 ms is still cheaper and faster on ordinary Kubernetes pods you tune yourself.

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

A mesh gives every pod a local Envoy that enforces mutual TLS, collects golden metrics, traces every hop, retries with backoff, splits traffic by version, and applies policy without you writing any of that logic into the microservice or touching kube-proxy, so you gain uniform observability, zero-trust security, and fine-grained traffic control that plain Services and Ingresses simply do not offer.

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

The sidecar intercepts every egress and ingress packet, handles service discovery, load-balancing, retries, circuit-breaking, mTLS hand-shake, and emits telemetry, letting the application remain purely business code; this separation is needed because it lets operators change routing or security rules cluster-wide without rebuilding or even restarting the application container.

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

Istio can shift 5% of live traffic to a canary build while mirroring another 100% to a shadow environment for regression tests, and it can trip a circuit-breaker after three consecutive 5xx responses within ten seconds, protecting the checkout service from a cascading failure during a database stall.

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

Knative installs a custom autoscaler that watches HTTP request concurrency per Revision; when in-flight requests
exceed the target concurrency it adds pods in seconds, and after a configurable window (default 60s) of no traffic it drives the Deployment to zero, waking it up again the moment a new request hits the activator component.

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

Eventing provides sources (Kafka, S3, GitHub, etc.), a broker/Channel layer that decouples producers from consumers, and triggers that filter and route CloudEvents to any Knative Service or Kubernetes address, so you can stitch serverless functions together with reliable delivery, retries, and dead-letter queues without writing custom glue code.

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.

Knative replaces your hand-written Deployment, Service, Ingress, and HPA with a single "Service" custom resource that, under the hood, still creates those objects but hides them—developers declare only the container image and concurrency target, and the platform generates and owns the Kubernetes boilerplate, eliminating YAML drift and letting teams ship features instead of tuning pods.

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

An InferenceService wraps the model artifact URI, resource requirements, and scaling hints into one YAML, and KServe turns that into a ready-made TensorFlow-Serving, TorchServe, or Triton container with predictable REST/gRPC endpoints, automatic scale-to-zero, A/B revision support, and built-in explainability hooks, so data scientists deploy models without building Dockerfiles or writing Flask servers.

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?

The request hits the Istio Gateway (TLS termination, routing rules), is forwarded to the Knative activator which may wake a cold KServe revision, the KServe agent downloads the model weights into the TFServing/Triton container running in a Kubernetes pod, inference runs on GPU/CPU, and the response retraces the same path; stalls can arise at TLS handshake, scale-from-zero latency, model weight download over object storage, or queuing inside the autoscaler before enough pods are ready.

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

By declaring a 90/10 weight between two InferenceService revisions Istio gradually steers live traffic while collecting golden signals, and automatic retries with circuit-breakers abort the shift if error rate spikes, giving safer, faster, metrics-driven rollouts than the traditional “edit Deployment image, watch Grafana, manually kubectl scale” approach, though it demands learning CRDs like VirtualService and can hide mis-configurations behind declarative YAML until runtime.