In-Class Assessment 4

Sun Yixuan 202383930020

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.

The core problem addressed by serverless computing is eliminating infrastructure management overhead, allowing developers to focus solely on code rather than cluster operations, node provisioning, or container orchestration. In traditional microservice deployments on Kubernetes, teams must manually handle tasks like node scaling, pod scheduling, resource allocation (CPU/memory limits), and load balancer configuration—all of which add operational complexity.

- Example where serverless excels: Low-frequency batch processing tasks (e.g., daily log analysis or monthly invoice generation). With serverless, instances are automatically spun up when the task triggers and destroyed immediately after completion (even scaling to zero during idle periods). This eliminates the need to maintain idle pods (as required in Kubernetes), reducing costs by up to 90%. In contrast, Kubernetes would require keeping at least one pod running continuously to await task triggers, wasting resources.
- Example where serverless is less suitable: Low-latency real-time systems (e.g., online game servers or stock trading platforms). Serverless suffers from cold starts—initializing runtimes and loading dependencies can introduce delays of hundreds of milliseconds to seconds, which is unacceptable for latency-sensitive use cases. Kubernetes, by contrast, allows pre-provisioning pods (setting minReplicas=5) and using guaranteed QoS (Quality of Service) to reserve resources, ensuring consistent sub-millisecond latency.

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

Kubernetes networking only provides basic inter-pod communication (via Services for DNS resolution and Layer 4 load balancing). A service mesh builds on this by adding fine-grained control at Layer 7 (the application layer) for microservice communication, with key advantages including:

- Dynamic traffic control: Kubernetes Services only support simple load balancing

methods like round-robin or random. Istio, however, enables routing based on HTTP headers (e.g., User-Agent), traffic weights, or paths (e.g., /v1/* vs. /v2/*), which is critical for gradual rollouts.

- End-to-end observability: Kubernetes only offers pod/node-level monitoring. Istio uses sidecar proxies to collect request-level metrics (latency, error rates, traffic volume) and integrates with tools like Jaeger or Zipkin for distributed tracing. This makes it easy to identify bottlenecks in cross-service calls—for example, pinpointing that "90% of requests from the payment service to the inventory service have latencies exceeding 500ms."
- Code-free security: Kubernetes requires manual configuration of PodSecurityPolicies or NetworkPolicies. Istio automatically enables mTLS encryption for inter-service communication and uses AuthorizationPolicies to control access (e.g., "Service A is allowed to call the POST endpoint of Service B"). No changes to application code are needed.
- Fault injection and resilience testing: Istio can simulate failures like latency (e.g., adding a 1-second delay to Service A) or errors (e.g., returning 503 errors for 50% of requests). This lets teams test system fault tolerance—a capability Kubernetes lacks.

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

A sidecar proxy is a lightweight network proxy deployed in the same pod as a microservice. All inbound traffic (requests received by the service) and outbound traffic (requests sent by the service to other services) is intercepted and routed through this proxy.

Key Functions of a Sidecar Proxy

1. Traffic interception and forwarding: It hijacks the pod's network traffic using iptables rules and forwards it according to service mesh policies—for example, routing requests to /api/v2 to the v2 version of a service.
2. Protocol handling: It parses Layer 7 protocols like HTTP/2 and gRPC to enable path- or method-based routing. For TCP traffic, it provides Layer 4 load balancing.
3. Security enhancement: It acts as a termination point for mTLS, encrypting and decrypting traffic and validating client certificates to ensure the confidentiality of

inter-service communication.

4. Resilience mechanisms: It implements retries (e.g., retrying 5xx errors 3 times), timeout controls (e.g., limiting waits for downstream service calls to 1 second), and circuit breaking (stopping calls to a failing service for 30 seconds after 10 consecutive errors).
5. Data collection: It records request metrics (latency, status codes) and sends them to Prometheus, and generates tracing data for Jaeger.

Why It's Needed in a Service Mesh

The core goal of a service mesh is to decouple network logic from business code. If each service had to implement routing, encryption, retries, and other network functions on its own, it would lead to code redundancy, inconsistencies across languages (e.g., different retry logic for Java and Python services), and difficulty in updating—changing network rules would require redeploying all services. The sidecar proxy centralizes these network functions, letting services focus solely on business logic.

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

Istio provides rich traffic management capabilities through Custom Resource Definitions (CRDs) like VirtualService and DestinationRule. Below are key features and their production use cases:

1. Weighted Routing

This feature routes a specified percentage of traffic to different service versions—for example, sending 80% of traffic to v1 and 20% to v2.

Production Use Case: Canary deployments. For instance, an e-commerce platform rolling out v2 of its payment service might first route 10% of user traffic to v2. It then monitors v2's error rate and latency; if no issues arise, it gradually increases the traffic percentage to 100%, minimizing the risk of full-scale outages.

2. Retries and Timeouts

Istio lets you configure retry policies (e.g., retrying GET requests 5 times with 100ms intervals) and timeout limits (e.g., waiting a maximum of 2 seconds for calls to the inventory service).

Production Use Case: Handling transient failures. If the inventory service occasionally

returns 503 errors due to network flakiness, retries can automatically recover the request. Timeout controls prevent upstream services from being blocked indefinitely by unresponsive downstream services, avoiding cascading failures.

3. Traffic Mirroring

This feature forwards a copy of production traffic to a test version of a service without affecting the main traffic's response.

Production Use Case: Safe testing. For example, a team can mirror real order traffic to a new recommendation algorithm service. This lets them validate if the new algorithm's outputs align with expectations—all without impacting end users.

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

Knative Serving enables intelligent autoscaling primarily based on request concurrency (by default), using either the Knative Pod Autoscaler (KPA) or the Kubernetes Horizontal Pod Autoscaler (HPA). Its design focuses on elasticity and resource efficiency.

Core Autoscaling Metrics

By default, Knative uses request concurrency (the number of concurrent requests a pod can handle) as the scaling metric. The target concurrency per pod is configurable via autoscaling.knative.dev/target (default: 70% utilization of a pod's maximum concurrency). For example, if a pod's maximum concurrency is 100, scaling is triggered when actual concurrency reaches 70. Knative also supports custom metrics (e.g., CPU usage, message queue length) by integrating with Prometheus adapters.

What Triggers Scaling Up

Scaling up is triggered when the number of incoming requests exceeds the total capacity of existing pods. Knative calculates the required number of pods using the formula: ceil(current concurrency / target concurrency). For example, if there are 2 existing pods with a target concurrency of 70, and current concurrency hits 200, Knative will compute ceil(200/70) = 3 new pods, bringing the total to 5.

What Triggers Scaling Down

Scaling down is triggered when concurrency remains below the target threshold for a cool-down period (default: 60 seconds). Knative gradually reduces the number of pods to avoid sudden capacity drops. Its most distinctive feature is scale-to-zero: if no requests are received for a configurable period (default: 30 seconds), all pods are

terminated, freeing up resources entirely.

Optimization Mechanisms

- Proactive Scaling: Knative analyzes traffic trends to create pods in advance, reducing the impact of cold starts.
- Step Control: It limits extreme pod fluctuations—for example, allowing a maximum 10x increase during scaling up and a 50% decrease during scaling down.

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

Knative Eventing is a platform layer for building event-driven architectures (EDAs). Its core purpose is to decouple event producers (e.g., databases, APIs, cloud storage) from consumers (e.g., serverless functions, microservices), enabling services to respond asynchronously to events (e.g., "process a file after it's uploaded" or "notify the inventory service when an order is created").

Core Components and Workflow

1. Event Sources: These connect external systems to the event mesh and generate events. Examples include KafkaSource (listens to Kafka topics), S3Source (monitors S3 bucket file changes), and APIServerSource (captures Kubernetes API events). All events are converted to the CloudEvents specification, which standardizes fields like type (event category), source (event origin), and data (event payload).
2. Brokers: These act as event hubs—they receive events from sources, store them (using in-memory storage by default, or persistent storage like Redis/Kafka for production), and make them available for consumers to subscribe to.
3. Triggers: These define filtering rules to route events from brokers to specific consumers. For example, a trigger might specify: "Send all events of type com.example.order.created to the order-processing service."
4. Subscribers: These are the services that process events, typically Knative Services (serverless services) or other HTTP endpoints.

How It Supports Event-Driven Architectures

- Loose Coupling: Producers do not need to know about consumers. For example,

an order system does not need to hardcode calls to the inventory service—instead, it emits an "order created" event, and the inventory service subscribes to that event via a trigger. Adding a new consumer (e.g., a analytics service) only requires creating a new trigger, no changes to the producer.

- Fan-Out Capabilities: A single event can be routed to multiple consumers via multiple triggers. For example, an "order created" event can simultaneously trigger inventory deduction, email notifications, and sales analytics.
- Persistence: When used with persistent brokers (e.g., Kafka), Knative Eventing ensures events are not lost, even if consumers are temporarily unavailable. It also supports retries for failed event deliveries.

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 is built on Kubernetes and abstracts underlying Kubernetes primitives to simplify serverless development. Below is how key primitives are abstracted, and the benefits for developers:

1. Kubernetes Deployments/ReplicaSets → Knative Configuration/Revision

Kubernetes Deployments and ReplicaSets manage pod lifecycles and scaling. Knative replaces these with two resources:

- Configuration: Defines the service template, including the container image, environment variables, and resource limits.
- Revision: An immutable snapshot of the Configuration—every time the Configuration is updated (e.g., a new image version), a new Revision is automatically created. Knative manages pod creation and updates based on the active Revision.

Developer Benefit: Developers no longer need to write complex Deployment YAML files or manage rolling update strategies. Version control is automated—for example, rolling back to a previous service version only requires switching the active Revision, no manual pod deletion or redeployment.

2. Kubernetes Services/Ingress → Knative Route

Kubernetes Services handle internal service discovery, while Ingress manages external

traffic routing. Knative replaces these with a single Route resource:

- The Route automatically creates a Kubernetes Service and an Ingress (or integrates with Istio for advanced routing). It handles DNS resolution (assigning a domain like my-service.default.example.com), TLS configuration, and traffic routing to the active Revision.

Developer Benefit: Developers do not need to manually configure Ingress rules or load balancers. Exposing a service to the internet only requires defining a Route, which abstracts all network setup.

3. Kubernetes HPA → Knative Pod Autoscaler (KPA)

The Kubernetes HPA scales pods based on metrics like CPU or memory usage, but requires manual configuration of scaleTargetRef and metric thresholds. Knative's KPA simplifies this by:

- Using request concurrency as the default scaling metric (more relevant for serverless workloads than CPU/memory).
- Supporting scale-to-zero out of the box.
- Eliminating the need for manual scaleTargetRef setup—KPA automatically targets the active Revision.

Developer Benefit: Developers do not need to learn complex HPA configurations. They only need to set a single parameter (e.g., autoscaling.knative.dev/target=100 for 100 concurrent requests per pod) to control scaling. Services automatically scale to zero when idle, reducing resource costs.

4. Kubernetes ConfigMaps/Secrets → Direct Integration

Knative retains Kubernetes ConfigMaps and Secrets for configuration management but simplifies their use:

- The Configuration resource lets developers inject ConfigMaps or Secrets as environment variables or volume mounts via envFrom, without manually associating them with Deployments.

Developer Benefit: Configuration changes are more intuitive—updating a ConfigMap automatically propagates to the service via a new Revision, no manual Deployment edits.

8. In KServe, what is the main function of an InferenceService, and how does it

simplify deploying ML models?

KServe's InferenceService is a core Custom Resource Definition (CRD) designed specifically for deploying machine learning (ML) models. It encapsulates the entire lifecycle of model serving—from model loading to scaling and monitoring—abstracting infrastructure complexity for ML teams.

Main Functions of InferenceService

1. Multi-Framework Support: It natively supports popular ML frameworks like TensorFlow, PyTorch, ONNX, and XGBoost. For each framework, it automatically selects the appropriate model server (e.g., TorchServe for PyTorch, Triton Inference Server for ONNX). Developers do not need to manually deploy or configure these servers.
2. Model Version Management: It supports multiple model versions, including a default version (for production traffic) and a canary version (for testing). Traffic can be split between versions (e.g., 95% to default, 5% to canary) using Istio, which is integrated with KServe.
3. Autoscaling Integration: It leverages Knative Serving's autoscaling capabilities to scale model pods based on request volume. This includes scale-to-zero, ensuring resources are not wasted when the model is idle.
4. Inference Optimization: It includes built-in optimizations to improve inference performance, such as model quantization (reducing model size for faster inference), dynamic batching (grouping requests to maximize GPU utilization), and GPU sharing (letting multiple models share a single GPU).
5. Monitoring and Logging: It automatically exposes inference-related metrics (e.g., latency, throughput, error rates) for integration with Prometheus. It also logs request and response data, making it easy to debug issues like failed predictions.

How It Simplifies ML Model Deployment

Traditional ML model deployment requires multiple manual steps: writing a serving interface (e.g., a Flask API to handle prediction requests), building a container image, configuring Kubernetes Deployments and Services, setting up scaling with HPA, and integrating monitoring.

With InferenceService, developers only need to define a simple YAML file specifying the model's location (e.g., a GCS or S3 path), framework, and resources (e.g., GPU count).

KServe then automatically handles model downloading, server deployment, traffic routing, scaling, and monitoring. ML teams can focus on model development rather than infrastructure setup.

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?

In a production ML workflow using KServe, data flows from an incoming HTTP request to a model prediction response through multiple layers (Istio, Knative, KServe, Kubernetes). Below is a detailed breakdown of the flow and potential latency bottlenecks:

Step 1: Client Sends an HTTP Request

The client sends a prediction request (e.g., POST /v1/models/mnist:predict with an image payload) to the KServe endpoint.

- Responsible Layer: Istio Gateway

The Istio Gateway receives the external request, terminates TLS (decrypting HTTPS traffic), and routes the request to the KServe Ingress Gateway based on the request's Host header or path.

- Potential Bottleneck: TLS handshake latency (especially for first-time connections) or insufficient Gateway resources (e.g., high CPU usage causing request queuing).

Step 2: Traffic Routes to the Target Service

After passing through the Istio Gateway, the request is routed to the appropriate InferenceService.

- Responsible Layers: Istio VirtualService + Knative Serving
    - The Istio VirtualService uses KServe's configuration to route the request to the InferenceService's associated Knative Service.
    - Knative Serving checks the status of the InferenceService's active Revision:
        - If no pods are running (due to scale-to-zero), Knative triggers the KPA to create new pods.

    - If pods are already running, Knative forwards the request to one of the pods via a Kubernetes Service.

- Potential Bottleneck: Cold starts are a critical issue here. If the service has scaled to zero, initializing new pods involves pulling the container image, starting the model server, and loading the model into memory (or GPU). For large models (e.g., 10GB+ transformer models), this can take tens of seconds. Additionally, complex Knative routing rules may introduce minor delays in request matching.

Step 3: Model Inference Processing

Once the request reaches the pod, it undergoes model inference.

- Responsible Layers: KServe Agent + Model Server
    - The KServe Agent, running alongside the model server in the pod, validates the request format, fetches the model from storage (e.g., GCS, S3, or a local volume), and loads it into the model server.
    - The model server (e.g., Triton, TorchServe) parses the input data (e.g., decoding an image tensor), executes the inference computation (e.g., running a CNN for image classification), and generates a prediction response.
- Potential Bottlenecks:
    - Model loading: Fetching large model files from remote storage (e.g., S3) can introduce latency, especially with slow network connections. Even with local caching, initial loads or updates to the model trigger this delay.
    - GPU/CPU contention: If multiple requests arrive simultaneously, the model server may queue inference jobs, leading to increased latency—this is common with GPU-bound workloads where computation is resource-intensive.
    - Preprocessing overhead: Converting raw input data (e.g., raw images) into the format expected by the model (e.g., normalized tensors) can add significant latency if not optimized.

Step 4: Response Returns to the Client

The prediction result travels back through the same layers to the client.

- Responsible Layers: Kubernetes Networking + Knative Service + Istio Gateway

    - The response exits the pod, is routed through the Kubernetes Service and Knative's network layer, and then through the Istio Gateway (which may re-encrypt the traffic for HTTPS).

- Potential Bottleneck: Large response payloads (e.g., returning detailed prediction probabilities or high-resolution output images) can slow down network transmission, especially over bandwidth-constrained links.

10. How can Istio 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—such as weighted routing, header-based matching, and fault injection—integrate seamlessly with Knative and KServe to enable controlled canary deployments and A/B testing. Below is how this works, along with pros and cons compared to manual rollout strategies.

How It Works

1. Canary Deployments in Knative:
    - Deploy two versions (Revisions) of a Knative Service: a stable v1 and a new v2.
    - Use an Istio VirtualService to split traffic between them—for example, routing 90% to v1 and 10% to v2 using weighted rules.
    - Monitor metrics like error rates and latency for v2; if stable, incrementally increase its traffic share (e.g., 20%, 50%, 100%).
2. A/B Testing in KServe:
    - Deploy an InferenceService with two model versions:
      a default (production) model and a canary (test) model.
    - Configure Istio to route traffic based on custom headers (e.g., X-Test-Group: beta) to direct specific user segments to the canary model, while all other traffic goes to default.
    - Compare prediction accuracy or latency between the two models to evaluate the canary version.

Pros vs. Manual Rollout Strategies

- Granular Traffic Control: Istio supports fine-grained routing (e.g., 5% traffic to the new version, or routing only premium users to the test model), whereas manual strategies (e.g., scaling Kubernetes Deployments) only split traffic by pod count (e.g., 1 out of 10 pods for the new version), which is less precise.
- Rapid Rollbacks: If issues arise, Istio lets you instantly redirect 100% of traffic back to the stable version by updating the VirtualService—no need to terminate pods or redeploy, which saves time during outages. Manual rollbacks require scaling down the new Deployment and scaling up the old one, which is slower and riskier.
- Integration with Serverless: Istio works seamlessly with Knative's scale-to-zero and Revision management. For example, if the canary version receives little traffic, Knative scales its pods down, and Istio automatically adjusts routing without manual intervention. Manual strategies struggle to coordinate scaling with traffic splits.
- Enhanced Observability: Istio's metrics (e.g., per-version latency, error rates) integrate with tools like Grafana, making it easy to compare performance between versions. Manual rollouts require custom monitoring setups, which are error-prone.

Cons vs. Manual Rollout Strategies

- Increased Complexity: Configuring
  Istio's VirtualService and DestinationRule requires learning its syntax and understanding service mesh concepts. Misconfigurations (e.g., incorrect weight distributions) can disrupt traffic entirely, whereas manual rollouts (e.g., updating a Deployment's image) are simpler.
- Latency Overhead: Istio's sidecar proxies process every request, adding microsecond-level latency (10–50µs per request). In high-throughput systems (e.g., 10k+ requests/second), this accumulates and can impact overall performance—manual rollouts avoid this overhead.
- Debugging Challenges: Traffic flows through multiple layers (Istio, Knative, KServe), making it harder to trace issues. For example, a failed request could stem from an Istio routing rule, a Knative scaling issue, or a model error—diagnosing this requires expertise across all components. Manual rollouts have fewer moving parts, simplifying debugging.