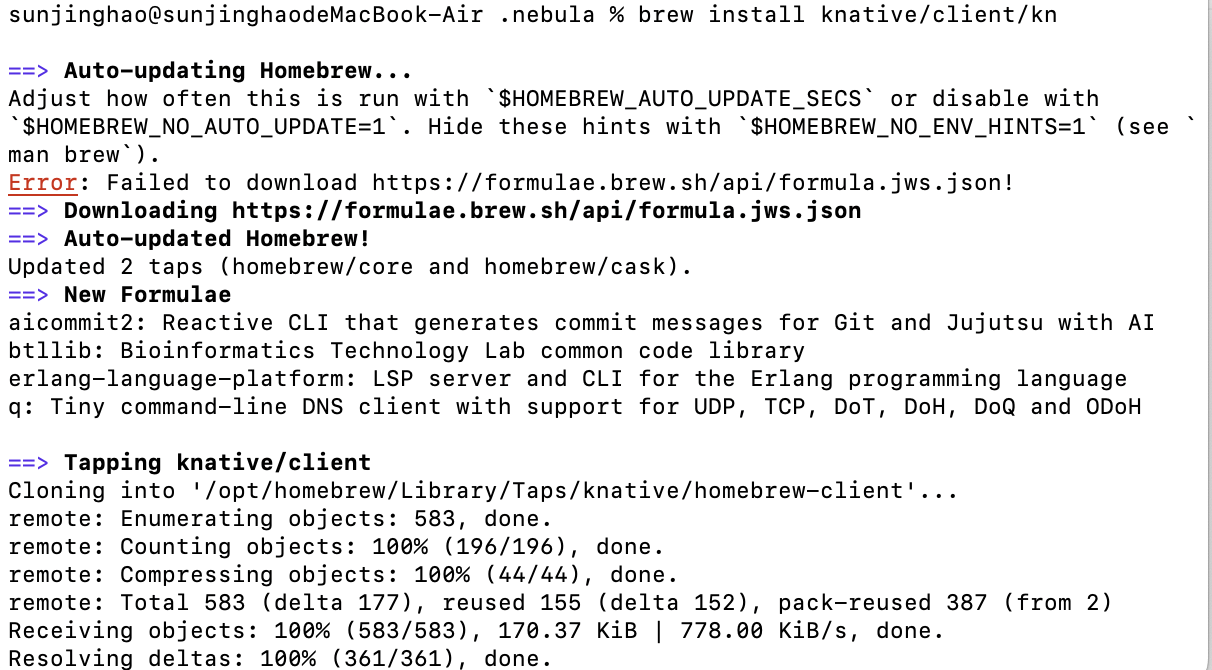
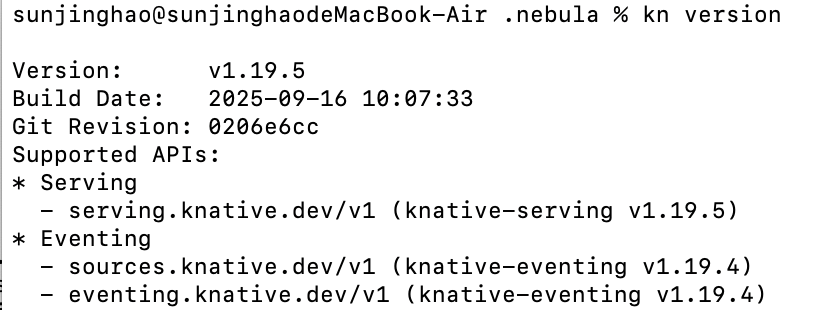
**1) What problem does serverless computing aim to solve compared to traditional microservice deployment on Kubernetes? One example where serverless is clearly better, one where it may not be.**

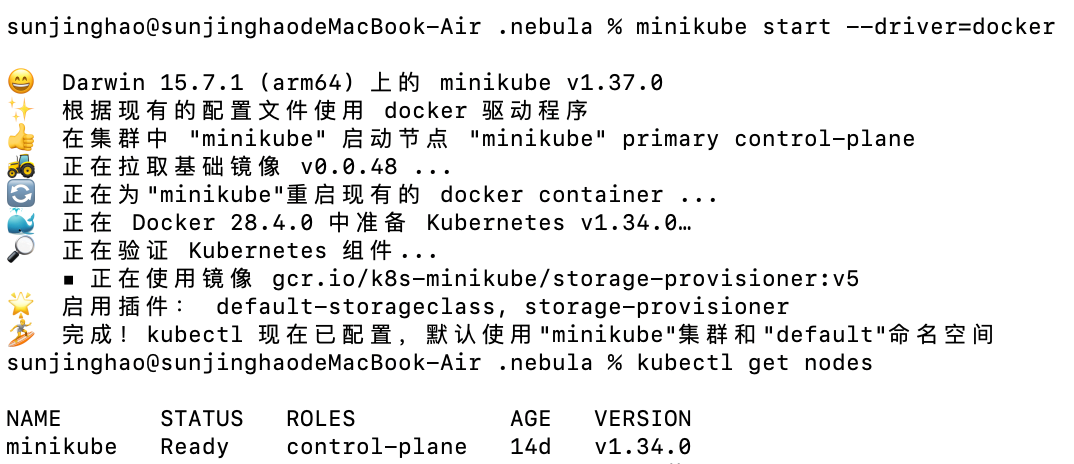
Serverless aims to remove infrastructure and operational burden from developers: auto-scale to zero when idle, automatically scale on demand, and abstract away deployment/instance management so developers can focus on code (or container images) and events. It reduces cost for spiky workloads, simplifies CI/CD for frequent small deployments, and provides built-in request-based autoscaling and routing.

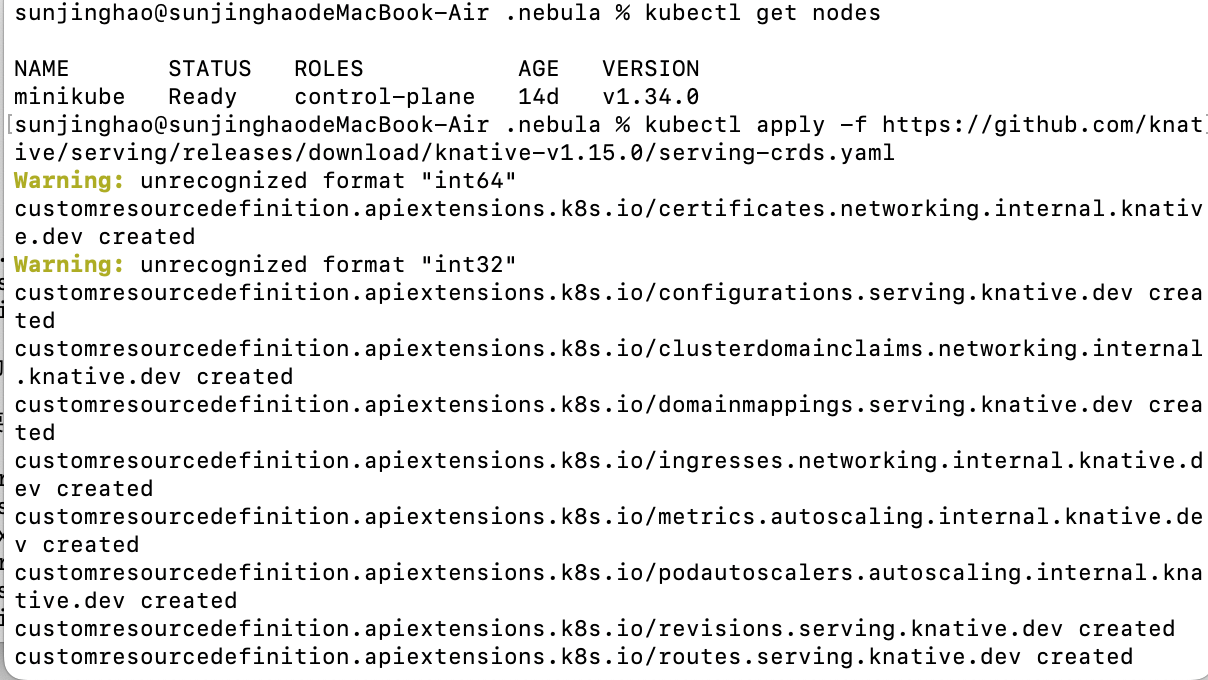
Clearly better: Event-driven, highly variable workloads with lots of short-lived requests (e.g., image thumbnail generation for uploads, webhook handlers). Serverless cuts cost (scale-to-zero), removes instance provisioning, and handles sudden bursts automatically.

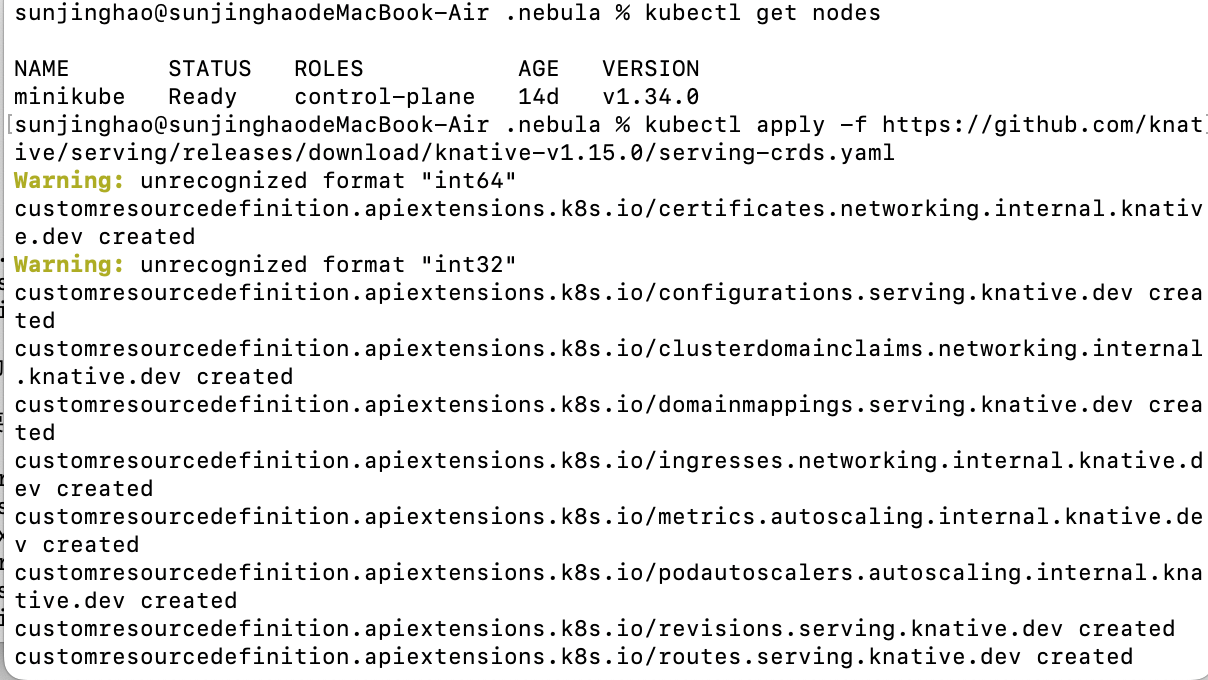
May not be better: Long-running, stateful, GPU-heavy, or latency-sensitive services where cold starts or orchestration delays are unacceptable (e.g., long-training jobs, real-time trading systems, high-throughput GPU inference). Serverless can add cold-start latency and often has limits on execution duration or resource access.











## **2. Advantages of Using a Service Mesh (Istio)**

* Fine-grained traffic control between microservices (routing, retries, failovers).
* Observability: metrics, logging, and distributed tracing built-in without changing application code.
* Security: automatic mutual TLS encryption and identity-based access control.
* Resilience: circuit breaking, rate limiting, and fault injection for testing.

## **3. Role of a Sidecar Proxy**

A sidecar proxy (like Envoy in Istio) runs alongside each service instance and intercepts all incoming/outgoing traffic. It handles traffic routing, observability, security, and retries independently of the application.

It is needed because it decouples network concerns from application code, providing consistent features across all services without modifying each service individually.

## **4. Istio Traffic Management Features**

* **Weighted routing:** Distribute a percentage of traffic to different service versions. Useful for canary deployments to test a new version without affecting all users.
* **Retries and circuit breaking:** Automatically retry failed requests or prevent overload on unhealthy services, improving reliability and fault tolerance in production.

## **5. Knative Serving and Autoscaling**

Knative Serving automatically scales the number of pods based on demand. Scaling up is triggered by incoming HTTP requests or high concurrency, while scaling down occurs when traffic drops to zero.

## **6. Knative Eventing**

Knative Eventing allows applications to react to events from various sources (e.g., Kafka, CloudEvents). It supports event-driven architectures by decoupling producers and consumers, enabling asynchronous communication and flexible workflows.

## **7. How Knative Leverages Kubernetes Primitives**

* **Deployments:** Knative abstracts these, so developers do not manage replicas directly.
* **Services:** Knative auto-creates Services for routing traffic to pods.
* **Horizontal Pod Autoscaler (HPA):** Knative abstracts scaling logic based on request concurrency.

This abstraction allows developers to focus on application logic without worrying about pod management, scaling rules, or networking configuration.

## **8. KServe InferenceService**

An InferenceService defines the entire ML model deployment, including model storage, runtime, and serving endpoints. It simplifies deployment by providing a declarative interface for serving ML models, handling autoscaling, routing, and versioning automatically.

## **9. ML Workflow in KServe**

Workflow for an HTTP request to prediction:

1. Client sends HTTP request → handled by **Istio** for routing and load balancing.
2. **Knative Serving** receives the request, triggers pods if needed, and scales dynamically.
3. **KServe**
4. **Kubernetes**

**Potential latency bottlenecks:** cold starts in Knative, network hops through Istio sidecars, or resource contention in Kubernetes pods.

## **10. Istio Traffic Routing for Canary and A/B Testing**

Istio can split traffic between versions using weighted routing, enable retries, and apply circuit breaking. This allows gradual rollouts or testing new models (A/B testing) without affecting all users.

**Pros:** Safe, automated rollouts, easy rollback, metrics collection per version.

**Cons:** Adds complexity in configuration, potential latency due to sidecar proxies, and reliance on correct routing rules.