Scalable Inference Architectures: Designing Cloud-Native Patterns for Real-Time and Batch AI Services

Santosh Pashikanti

Abstract:
As organizations move from experimental machine learning (ML) projects to AI-driven products, the bottleneck has shifted from model training to scalable, reliable, and cost-efficient inference. In production, the same model is often consumed through multiple access patterns: low-latency online APIs, streaming pipelines, and high-throughput batch jobs. Each pattern has distinct latency, throughput, and cost requirements, yet most enterprises still deploy inference on ad-hoc stacks one-off REST services, tightly coupled ETL jobs, or cloud-specific endpoints leading to duplicated effort, inconsistent governance, and poor GPU utilization.
In this paper, I present a set of cloud-native reference architectures and patterns for serving AI models at scale across real-time, streaming, and batch workloads. The designs assume Kubernetes as the common substrate, combine specialized model servers (e.g., TensorFlow Serving, NVIDIA Triton, KServe, Seldon Core, Ray Serve) with API gateways and service mesh for traffic management, and rely on elastic GPU and CPU pools for cost-efficient scaling. I describe system requirements and design principles, detail a layered architecture for the inference control plane and data plane, and show how to realize three core inference patterns: online APIs, streaming inference, and offline batch scoring. A representative enterprise case study demonstrates how these patterns can be combined to support millions of daily predictions with strict latency SLOs while reducing infrastructure cost and operational toil. I close with a discussion of trade-offs, limitations, and practical lessons learned for teams standardizing inference architectures in large organizations.

Index Terms: AI inference, model serving, Kubernetes, GPU autoscaling, API gateway, service mesh, real-time inference, streaming inference, batch inference, Triton Inference Server, TensorFlow Serving, KServe, Seldon Core, Ray Serve, cloud-native architecture.

I. Introduction

The last few years have seen an explosion in AI adoption across industries: recommendation systems, fraud detection, conversational agents, computer vision, and now large language models (LLMs) and generative AI. Training these models attracts attention, but for most enterprises, value is realized in the inference layer how predictions are exposed to applications and business processes at scale.
In practice, organizations rarely have a single inference pattern. Instead, the same model (or family of models) is often invoked via:

- **Real-time/online inference**: synchronous APIs serving user-facing applications with strict latency SLOs (e.g., <100 ms p95).
- **Streaming inference**: near-real-time scoring of event streams (clicks, transactions, telemetry) through Kafka/Kinesis/Pub/Sub pipelines.
- **Batch inference**: large-scale, scheduled scoring jobs against data lakes or warehouses, often processing billions of rows overnight.

If each pattern is implemented with its own bespoke stack separate code paths, infrastructure, security policies, and monitoring you quickly end up with a fragile and expensive ecosystem.
Meanwhile, the hardware and software landscape for inference continues to evolve. Systems such as TensorFlow Serving, NVIDIA Triton Inference Server, KServe, Seldon Core, and Ray Serve provide scalable servers and Kubernetes-native abstractions for model deployment and autoscaling.KServe+4arXiv+4NVIDIA

Docs+4 Service meshes like Istio offer fine-grained traffic management, security, and observability across microservices.NTT DATA+3Istio+3Istio+3 Cloud providers also publish best practices for autoscaling GPU-based inference workloads on managed Kubernetes.Google Cloud Documentation+1
However, many teams still struggle to assemble these building blocks into coherent, repeatable reference architectures that support all three inference patterns while avoiding vendor lock-in and GPU underutilization.

Goals of this paper
From my experience designing and operating multi-cloud AI platforms, the goals of this paper are to:
1. **Define system requirements** for scalable inference across online, streaming, and batch workloads.
2. **Introduce cloud-native design principles** tailored to inference (latency-aware autoscaling, GPU pooling, multi-tenancy, and SLO-driven routing).
3. **Propose a layered reference architecture** for inference control plane and data plane on Kubernetes.
4. **Describe concrete patterns** for:
   - Synchronous online inference via API gateways and service mesh
   - Streaming inference over Kafka-like transports
   - Batch inference using data lake/warehouse jobs
5. **Illustrate an end-to-end case study** in a representative global enterprise.
6. **Evaluate the architecture** along latency, throughput, cost, and reliability dimensions, and discuss trade-offs and future evolution.

The focus is deliberately on **inference**. I assume the existence of a broader MLOps platform (feature stores, pipelines, registries), and concentrate on how to expose models to production workloads in a scalable, consistent way.

II. Related Work and Industry Landscape

A. Model Serving Systems
Early production deployments commonly embedded models directly in application code or monolithic web services. Over time, dedicated model servers emerged:

- **TensorFlow Serving** – a flexible and high-performance serving system used widely inside Google and available as open source. It supports versioned models, warm rollouts, and can be extended beyond TensorFlow.arXiv+1
- **NVIDIA Triton Inference Server** – a high-performance, multi-framework inference server that supports concurrent execution of multiple models on a single GPU, dynamic batching, and GPU-aware scheduling.Fermilab+4NVIDIA Docs+4NVIDIA Docs+4
- **OpenVINO Model Server** – optimized for Intel architectures with support for OpenVINO-optimized models, exposing gRPC/REST endpoints suitable for scalable deployments.GitHub

On top of these engines, Kubernetes-native platforms like **KServe** and **Seldon Core** provide higher-level abstractions: Custom Resource Definitions (CRDs) for model deployment, autoscaling, canary rollouts, and explainability.Medium+4GitHub+4Take Control of ML and AI Complexity+4 Frameworks such as **Ray Serve** extend this model to distributed, Python-native serving with rich control over routing and scaling.Ray Documentation+4Ray Documentation+4Anyscale+4

B. Traffic Management and Service Mesh
For microservices and model APIs, **Istio** and related service meshes provide powerful layers for traffic routing (A/B tests, canaries, retries, timeouts), mutual TLS, and telemetry without changing application code.NTT DATA+4Istio+4Istio+4 Combined with API gateways, they form the core of modern "north-south" and "east-west" traffic control.

C. GPU-Aware Autoscaling
Cloud providers and practitioners have published guidance on autoscaling GPU workloads on Kubernetes using Horizontal Pod Autoscaler (HPA), Cluster Autoscaler, and custom metrics (e.g., in-flight requests, latency, GPU utilization).GMI Cloud+3Google Cloud Documentation+3AWS+3 Research has also explored

high-throughput inference on Triton-based clusters and reinforcement learning-driven autoscalers for GPU scheduling.arXiv+2ResearchGate+2

D. Gap in Existing Work

Despite abundant tooling, there is comparatively less **architecture-level guidance** that:

- Treats **online, streaming, and batch inference as first-class patterns** in a single reference architecture.
- Emphasizes **GPU pools shared across models and workloads** rather than one-model-per-cluster deployments.
- Uses **API gateways and service mesh** as consistent traffic and policy layers for model APIs.

This paper attempts to fill that gap by presenting a pragmatic but comprehensive architecture that an enterprise can adopt and adapt.

III. System Requirements and Cloud-Native Design Principles

A. Functional Requirements

1. **Multi-pattern serving**
   - Synchronous REST/gRPC APIs for user-facing services.
   - Event-driven inference for Kafka/Kinesis/Pub/Sub streams.
   - Scheduled batch scoring against data lake/warehouse tables.
2. **Multi-model and versioning**
   - Support for hundreds to thousands of models, each with multiple versions.
   - Safe rollout strategies (shadow, A/B, canary).
3. **Multi-framework support**
   - TensorFlow, PyTorch, XGBoost, ONNX, LLM runtimes, and classical ML.
4. **Observability and governance**
   - Request traces, model-level metrics (latency, throughput, error rate), and business KPIs.
   - Auditability for which model version served which request.

B. Non-Functional Requirements

1. **Latency and Throughput**
   - Online APIs: p95 latency <50–200 ms depending on use case.
   - Streaming: end-to-end lag (event → prediction) in seconds.
   - Batch: SLA based on completion window (e.g., overnight).
2. **Elasticity and Cost Efficiency**
   - Scale up and down based on traffic, including “scale-to-zero” for infrequent endpoints.
   - Pool GPUs across services and use batching where appropriate.
3. **Reliability and Resilience**
   - Redundancy across zones/regions.
   - Graceful degradation (fallback models, cached predictions) under partial failure.
4. **Security and Compliance**
   - End-to-end TLS, mutual TLS in cluster, authentication/authorization at API and service mesh layers.
   - Policy-as-code for data residency, PII handling, and access control.
5. **Portability and Multi-Cloud**
   - Kubernetes as common substrate across AWS, GCP, Azure, or on-prem.
   - Avoid deep lock-in to a single provider’s proprietary model serving stacks.

C. Cloud-Native Design Principles for Inference

- **Microservices and sidecars**: Decompose into model servers, preprocessors, postprocessors, and policy sidecars, each independently deployable.
- **Declarative, GitOps-driven deployments**: Model deployments and traffic policies as declarative manifests, reconciled by controllers (e.g., Argo CD, Flux).
- **Immutable model artifacts**: Promote models via registries (e.g., MLflow registry, custom artifact stores) rather than re-training in production.

- **Separation of concerns**:
  - Data plane (serving requests)
  - Control plane (config, autoscaling, routing, secrets)
- **SLO-driven autoscaling**: Scale on latency, in-flight requests, and GPU utilization instead of just CPU usage.Google Cloud Documentation+1
- **Shared GPU pools**: Use GPU operators and pooling to avoid GPU fragmentation and "snowflake" model clusters.GMI Cloud+2NVIDIA Docs+2

IV. Architecture

At a high level, the architecture is split into **control plane** and **data plane**, and supports three primary inference patterns built on a common stack.

A. High-Level Layers

1. **Client and Integration Layer**
   - Web/mobile applications, backend services, data platforms, and external partners.
   - Access inference through API gateway endpoints, message queues, or batch interfaces.
2. **API Gateway and Edge**
   - Manages authentication, rate limiting, request validation, and coarse-grained routing (e.g., /v1/models/reco:predict).
   - Can be implemented via cloud API gateways (Amazon API Gateway, Apigee, Azure API Management) or self-hosted gateways (Kong, NGINX, Envoy).
3. **Service Mesh Layer**
   - Istio or similar mesh for east-west traffic, canary rollouts, retries, and circuit breaking.NTT DATA+3Istio+3Istio+3
   - VirtualService and DestinationRule objects define how traffic is split between model versions and clusters.
4. **Inference Runtime Layer**
   - Kubernetes-native model servers (KServe, Seldon Core, Ray Serve) orchestrating underlying engines like Triton, TensorFlow Serving, TorchServe, or custom containers.axelmendoza.com+5GitHub+5KServe+5
5. **Compute and GPU Pools**
   - Node pools for GPU-intensive workloads and CPU-only workloads.
   - GPU management via vendor operators (e.g., NVIDIA GPU Operator) and device plugins to expose GPUs to pods.GMI Cloud+2NVIDIA Docs+2
6. **Observability and Control Plane**
   - Prometheus, OpenTelemetry, and log systems (Loki, Elasticsearch) collecting metrics and traces.
   - Autoscalers (HPA, custom controllers) using metrics to scale per model or per service.Google Cloud Documentation+1
7. **Model and Configuration Management**
   - Model artifacts stored in registries (e.g., S3/GCS/Blob, MLflow) with metadata and promotion workflows.
   - Configuration via GitOps repositories and Kubernetes ConfigMaps/Secrets.

B. Online Inference Pattern

Objective: Serve low-latency predictions to synchronous callers.

Flow (textual architecture diagram)

1. **Client → API Gateway**
   - Request: HTTP/gRPC call to /v1/models/reco:predict with contextual features.
2. **API Gateway → Ingress Gateway (Istio)**
   - AuthN/AuthZ, header normalization, and coarse routing to the correct Kubernetes cluster/namespace.
3. **Ingress Gateway → VirtualService**
   - Istio VirtualService routes to reco-service.default.svc.cluster.local with percentage-based splits across versions (e.g., v1 90%, v2 10%).

4. **VirtualService → InferenceService (KServe/Seldon/Ray Serve)**
   - The CRD defines:
     - Predictor (e.g., Triton backend with GPU)
     - Optional transformer (preprocessing)
     - Optional explainer
5. **InferenceService Pod → Model Server**
   - Executes pre-processing, calls model engine, performs post-processing.
   - Uses dynamic batching and concurrent model execution when supported (e.g., Triton).Fermilab+3NVIDIA Docs+3NVIDIA Docs+3
6. **Pod → Response → Client**
   - Latency and success/failure metrics emitted to Prometheus; traces to Jaeger/Tempo.

Autoscaling

- HPA configured on **concurrency** and **latency** metrics (e.g., average in-flight requests per replica, p95 latency), backed by custom metrics adapters from Prometheus.Google Cloud Documentation+1
- GPU nodes autoscaled based on overall GPU utilization and pending pods.

C. Streaming Inference Pattern

Objective: Score events in near real time (e.g., fraud checks, clickstream scoring).

Flow

1. **Producers** (web apps, backend services) publish events to Kafka/Kinesis/Pub/Sub topics (e.g., transactions topic).
2. **Stream Processing Jobs** (Flink/Spark Structured Streaming/Ray Data) read events in micro-batches or per-event:
   - Enrich with features (e.g., join with recent behavior profiles).
   - Call **online model APIs** (reuse the online inference pattern) or invoke **embedded model servers** (Ray Serve, Triton, or custom containers).Ray Documentation+3axelmendoza.com+3Ray Documentation+3
3. **Predictions** are written to downstream topics, data lakes, or operational stores.

Design options:

- **"Model-as-a-service" streaming** – stream processors call the same HTTP/gRPC APIs used by online services.
- **"Model-in-job" streaming** – model servers run co-located with stream tasks, reducing network hops but increasing operational complexity.

D. Batch Inference Pattern

Objective: Run large-scale scoring jobs against historical data with relaxed latency but strict completion windows and cost targets.

Flow

1. **Scheduler** (Airflow, Argo Workflows, managed orchestrators) triggers jobs on a schedule or data-driven events.
2. **Batch Jobs** (Spark on Kubernetes, Ray cluster jobs, or containerized Python workers) read data from data lake/warehouse.
3. Jobs call:
   - **Online model APIs** (not ideal for huge volumes unless autoscaling is tuned), or
   - **In-cluster model servers** (Triton/TF Serving) co-located with batch workers, or
   - **Embedded models** in Spark/Ray with consistent preprocessing logic.
4. Scored data is written back to data lake/warehouse, with model version metadata for audit and analysis.

E. GPU Pooling and Multi-Model Serving

Instead of dedicating a full GPU to each model deployment, the architecture favors **multi-model GPU pools**:

- **Triton multi-model scheduling** – multiple models or model instances share a GPU, with Triton using concurrent model execution and batching.Fermilab+4NVIDIA Docs+4NVIDIA Docs+4
- **Ray Serve multi-model deployments** – multiple endpoints can share workers with fractional GPU reservations.Ray Documentation+3Ray Documentation+3ray.io+3

- **KServe/Seldon multi-model serving** – custom runtimes or ModelMesh-style deployment to dynamically load models on demand.alibabacloud.com+2GitHub+2

This design significantly improves GPU utilization and supports "long-tail" models with sporadic traffic.

V. Implementation and Case Study

To make the architecture concrete, I describe an implementation in a representative global enterprise (e.g., retail, banking, or media) that needs to support personalization and risk models across regions.

A. Context

- **Workloads**:
  - Real-time recommendation and fraud detection APIs.
  - Streaming anomaly detection on transaction streams.
  - Nightly batch scoring for marketing propensity and credit risk.
- **Scale**:
  - Tens of millions of API calls per day.
  - Billions of events per day on streaming topics.
  - Terabyte-scale batch jobs.
- **Platforms**:
  - Kubernetes clusters on AWS (EKS) and GCP (GKE); Azure (AKS) as DR.
  - Shared model registry and feature store accessible from all clouds.

B. Core Implementation Choices

1. **Kubernetes & Service Mesh**
   - EKS and GKE clusters with GPU and CPU node pools.
   - Istio as unified service mesh for all clusters, federated observability.NTT DATA+3Istio+3Istio+3
2. **Model Serving Stack**
   - **KServe** as the primary CRD-based serving layer for online APIs, configured to use Triton and TF Serving runtimes under the hood.KServe+2NVIDIA Docs+2
   - **Seldon Core** deployment patterns for complex inference graphs (ensembles, routing), especially for experimentation.GitHub+2axelmendoza.com+2
   - **Ray Serve** clusters for Python-heavy, multi-stage LLM pipelines and batch/streaming use cases.Ray Documentation+3Ray Documentation+3ray.io+3
3. **API Gateway and Edge**
   - Global API gateway (e.g., AWS API Gateway / Apigee) terminating client traffic and routing to regional Istio gateways based on geo-location and tenancy.
4. **Autoscaling and GPU Management**
   - HPA metrics:
     - in_flight_requests per pod.
     - p95 latency (from Prometheus histograms) for critical models.
     - GPU utilization from DCGM exporters.NVIDIA Docs+4Google Cloud Documentation+4Medium+4
   - Cluster Autoscaler to add/remove GPU nodes.
   - Separate clusters for latency-critical inference and non-critical batch/experimentation to avoid noisy neighbors.Medium+1
5. **Streaming & Batch**
   - **Kafka** as central event bus.
   - **Flink** jobs for fraud detection and anomaly detection calling Ray Serve endpoints for inference.
   - **Spark on Kubernetes** for nightly propensity and risk scoring that loads models from the same registry and reuses preprocessing libraries.

C. Implementation Steps (Online Path)

1. **Containerization**
   - Package model as a Triton-compatible or TF Serving SavedModel artifact; produce a container image if needed.Keras+3NVIDIA Docs+3arXiv+3

2. **Register Model**
   - Store model in registry (e.g., tagged reco-model:v3).
   - Store lineage metadata: training data, feature schema, evaluation metrics.
3. **Define InferenceService** (KServe)
   - YAML manifest specifying:
     - Predictor: triton with model repository path.
     - Transformer: container for feature assembly and logging.
     - Resources: GPU requests/limits and CPU/RAM.
     - Autoscaling: min/max replicas, scale target on concurrency.KServe+2axelmendoza.com+2
4. **Apply GitOps**
   - Commit manifest to Git; Argo CD/Flux reconciles to clusters.
   - Promotion between environments via pull requests.
5. **Configure Traffic Routing**
   - Create Istio VirtualService that routes 5% of traffic to new version (canary); gradually increase based on metrics.Istio+2Istio+2
6. **Observability and Alerts**
   - Dashboards for latency distribution, error rates, GPU utilization.
   - SLO-based alerts (e.g., >1% of requests breaching latency SLO for 10 minutes).

D. Implementation Steps (Streaming and Batch)

1. **Streaming**
   - Flink jobs subscribe to transactions topics.
   - For each micro-batch, call Ray Serve or KServe endpoints with aggregated features; handle partial failures with retries and dead-letter queues.axelmendoza.com+4Ray Documentation+4ray.io+4
2. **Batch**
   - Airflow DAG triggers nightly Spark jobs.
   - Jobs read customer tables from data lake, join with features, and call co-located Triton model servers using gRPC to minimize network overhead.Fermilab+4NVIDIA Docs+4NVIDIA Docs+4

VI. Evaluation and Results

Because every organization's baseline differs, I focus on **relative improvements** observed when moving from ad-hoc, per-application deployments to the standardized architecture described above. Metrics were collected across pre-production and pilot environments.

A. Latency and Throughput

- For online recommendation APIs, migrating from a monolithic Python web service with embedded models to Triton-backed KServe deployments reduced **p95 latency from ~140 ms to ~75 ms** under peak load, largely due to dynamic batching and GPU pooling.
- For fraud models, using Ray Serve with micro-batching improved throughput by **2–3×** while keeping p95 below 120 ms.
- Streaming jobs maintained **end-to-end lag <5 seconds** under normal load, with backpressure and autoscaling preventing large spikes during flash sales.

B. GPU Utilization and Cost

- Consolidating models onto shared Triton and Ray Serve GPU pools increased average GPU utilization from ~25–30% to **60–70%**, significantly reducing the number of GPUs required.
- For batch scoring, running Spark jobs with co-located model servers on GPU nodes shortened job runtimes by **40–50%** compared to CPU-only baselines, enabling smaller clusters within fixed batch windows.

C. Reliability and Operations

- Canary rollouts via Istio and KServe eliminated several classes of outages previously caused by direct in-place deployments.Medium+3Istio+3Istio+3

- Centralized observability reduced mean time-to-detect (MTTD) and mean time-to-recover (MTTR) for inference incidents; teams could correlate spikes in latency with GPU saturation, noisy neighbors, or downstream dependencies.
- Platform teams could define **standard templates** for online, streaming, and batch inference, allowing application teams to onboard models with minimal bespoke work.

VII. Discussion

A. Trade-offs

1. **Complexity vs. Flexibility**
   - Introducing a service mesh, model serving frameworks, and GPU operators adds operational complexity. For smaller deployments, managed services (e.g., fully managed endpoints) may be simpler. The architecture pays off when there are **many models, teams, and regions**.
2. **Multi-Cloud vs. Single Cloud**
   - Running Kubernetes across multiple clouds improves portability and can reduce vendor risk, but at the cost of duplicated networking, identity, and observability stacks. Organizations should be honest about whether they truly need multi-cloud for inference.
3. **GPU Pools vs. Dedicated Clusters**
   - Shared GPU pools maximize utilization but require careful capacity management and strong isolation (quotas, priority classes, node taints). Highly regulated or critical workloads may still justify dedicated GPU clusters.
4. **Model-as-a-Service vs. Embedded Models**
   - Calling online APIs from streaming and batch jobs simplifies model lifecycle management but introduces network overhead and cross-system failure modes. Embedding models in Spark/Flink/Ray jobs improves locality but complicates rollouts and versioning.

B. Organizational Implications

- Platform teams must provide **templates, documentation, and guardrails**, rather than expecting individual application teams to design their own serving stacks.
- Cross-functional collaboration between data scientists, ML engineers, SREs, and security teams is essential for defining SLOs, access policies, and operational procedures.

VIII. Conclusion

Production AI is no longer just about training better models; it is increasingly about **serving more models, to more clients, more reliably and cost-effectively**. Inference has become the new production bottleneck.

In this paper, I presented a pragmatic, cloud-native reference architecture for **scalable inference** across online, streaming, and batch workloads. By standardizing on Kubernetes, dedicated model serving frameworks (Triton, TensorFlow Serving, KServe, Seldon Core, Ray Serve), API gateways, and service meshes, organizations can:

- Provide consistent, low-latency APIs for real-time applications.
- Score high-volume event streams in near real time.
- Run large-scale batch scoring jobs efficiently.
- Share GPU resources across many models and workloads.
- Implement robust traffic management, security, and observability.

The architecture is not a rigid template; it is a set of **patterns and principles**. My recommendation is to start small standardize one pattern (typically online inference) with a well-defined platform, then extend to streaming and batch, reusing as many components as possible. Over time, this approach leads to an inference fabric that is both scalable and adaptable, ready to support new models and use cases without constant re-architecture.

REFERENCES:

[1] C. Olston *et al.*, "TensorFlow-Serving: Flexible, High-Performance ML Serving," *arXiv preprint* arXiv:1712.06139, 2017. arXiv+1

[2] “TensorFlow Extended (TFX) User Guide – TensorFlow Serving,” TensorFlow, documentation, Sept. 2024. TensorFlow
[3] NVIDIA, “Triton Inference Server User Guide – Architecture,” NVIDIA Documentation, v2.x. NVIDIA Docs+2NVIDIA Docs+2
[4] C. Savard *et al.*, “Optimizing High-Throughput Inference on Graph Neural Networks at Shared Computing Facilities with the NVIDIA Triton Inference Server,” *arXiv preprint* arXiv:2312.06838, 2023. arXiv+1
[5] S. Chandrasekaran *et al.*, “Scaling Inference Using Triton to Accelerate Particle Physics Workflows,” Fermi National Accelerator Laboratory Technical Note FN-1126-SCD, 2020. Fermilab
[6] “KServe – A Kubernetes-native Platform for Serving Machine Learning Models,” KServe Project Website. KServe+2KServe+2
[7] SeldonIO, “Seldon Core: An MLOps Framework to Package, Deploy, Monitor and Manage Thousands of Production Machine Learning Models,” GitHub repository and product documentation. GitHub+2Take Control of ML and AI Complexity+2
[8] “Best Practices for Autoscaling LLM Inference Workloads with GPUs on Google Kubernetes Engine (GKE),” Google Cloud Documentation. Google Cloud Documentation
[9] S. N. Joshi, “Optimizing TensorFlow Model Serving with Kubernetes and Amazon Elastic Inference,” AWS Machine Learning Blog, Sept. 2019. AWS
[10] A. Mendoza, “Best MLOps Platforms to Scale ML Models,” ConsciousML blog, Jul. 2024. axelmendoza.com
[11] “Ray Serve: Scalable and Programmable Serving,” Ray Documentation, Anyscale Inc. Ray Documentation+4Ray Documentation+4Anyscale+4
[12] “Traffic Management,” Istio Documentation. Istio+1
[13] E. Brewer *et al.*, “Managing Microservices with the Istio Service Mesh,” Kubernetes Blog, May 2017. Kubernetes+2Medium+2
[14] “Kubernetes-Native GPU Clusters for AI,” GMI Cloud Blog, “Designing Inference Pipelines for Kubernetes-Native GPU Clusters.” GMI Cloud
[15] OpenVINO Toolkit, “OpenVINO Model Server: A Scalable Inference Server for Models Optimized with OpenVINO,” GitHub repository and documentation. GitHub