EDUSCHOOL
ACADEMIC RESEARCH & PUBLISHERS

INTERNATIONAL JOURNAL OF
TECHNICAL RESEARCH STUDIES (IJTRS)
(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online: ISSN Print

Cloud-Native AI Workloads Using Microservice Architectures

Win Mathew John

Head & Associate Professor, PG Department of Computer Applications, Marian College Kuttikanam (Autonomous), India.

Article information

Received: 6th December 2025
Received in revised form: 9th January 2026
Accepted: 12th February 2026
Available online: 9th March 2026

Volume: 2
Issue: 1
DOI: https://doi.org/10.5281/zenodo.18900136

Abstract

Production machine learning systems often run as monolithic applications where data preprocessing, model inference, and post-processing share a single container and scale as a unit. This coupling wastes compute resources when workloads are heterogeneous a GPU-bound inference stage idles CPU-bound preprocessing capacity, and vice versa. This paper presents a microservice architecture deployed on Kubernetes that decomposes an ML serving pipeline into six independently scalable services: API Gateway, Model Registry, Feature Store, Inference Engine (NVIDIA Triton), Training Pipeline, and Monitoring. Experiments on a five-node cluster with three workloads (ResNet-50 image classification, BERT-base text classification, XGBoost tabular prediction) show that the microservice design achieves 3.2× the peak throughput of an equivalent monolithic Flask deployment under 10× bursty loads. Tail latency (p99) drops by 45%, and GPU utilisation increases from 52% to 78%. Horizontal Pod Autoscaler (HPA) driven by inference-queue-depth metrics provisions additional pods within 18 seconds of load onset, containing throughput degradation to under 5% during burst transients.

Keywords:- Microservice Architecture, Kubernetes, Horizontal Pod Autoscaler, NVIDIA Triton Inference Server

I. INTRODUCTION

The operational demands of machine learning workloads diverge sharply from those of traditional web services. Inference requests arrive in bursts — a product recommendation system, for example, may handle 50 requests per second at 3 AM and 5,000 at noon. Training jobs consume GPU hours in long-running batches. Feature computation pipelines have their own CPU and memory profiles. Packaging all of these functions into a single container the default output of most ML frameworks creates an inflexible monolith that cannot scale its parts independently [1].

Microservice architectures address this by decomposing a system into small, autonomous services that communicate through well-defined APIs [2]. Each service owns its data, runs in its own container, and scales according to its specific resource demands. Kubernetes, the dominant container orchestration platform, provides the primitives Deployments, Services, Horizontal Pod Autoscaler (HPA), and GPU device plugins needed to operationalise this decomposition at scale [3].

Several ML-specific platforms have adopted microservice principles. Kubeflow [4] packages Jupyter, Katib, and KFServing as separate Kubernetes deployments. MLflow [5] decouples experiment tracking from model serving. NVIDIA Triton Inference Server [6] handles model loading and dynamic batching as a standalone service. What remains under-explored is a systematic comparison of monolithic versus microservice deployments across multiple workload types, with emphasis on autoscaling behaviour and resource efficiency under realistic traffic patterns.

This paper contributes:

- A reference microservice architecture for ML serving and training.
- An evaluation across three model types with steady, bursty, and ramping load profiles.
- A custom HPA metric (inference queue depth) that outperforms the default CPU-based scaler for GPU-bound workloads.

II. BACKGROUND AND RELATED WORK

A. Microservice Architecture Principles

Microservice design follows several core tenets: bounded context (each service encapsulates a single business capability), independent deployability, decentralised data management, and API-first communication [2]. In ML systems, the natural service boundaries align with pipeline stages: data ingestion, feature engineering, model training, model serving, and monitoring. Each stage has distinct scaling axes   data ingestion is I/O-bound, training is GPU-bound, and serving latency depends on batch size and model complexity.

B. Container Orchestration with Kubernetes

Kubernetes abstracts a cluster of machines into a unified resource pool [3]. Pods the smallest deployable units run one or more containers. Deployments declare the desired number of pod replicas. The HPA adjusts replica count based on observed metrics. The default metric is CPU utilisation, but custom metrics (exposed via the Metrics API) can trigger scaling decisions that better reflect ML workload behaviour, such as request queue length or GPU memory pressure [7].

C. ML Serving Frameworks

TensorFlow Serving [8] was among the first production-grade serving solutions, supporting model versioning and batching. NVIDIA Triton [6] extends this to multi-framework support (TensorFlow, PyTorch, ONNX, XGBoost) with concurrent model execution and dynamic batching across heterogeneous hardware. KServe (formerly KFServing) [9] provides a Kubernetes-native inference abstraction with canary rollouts and explainability hooks. Clipper [10] introduced a prediction cache and model container abstraction for latency-sensitive applications.

D. MLOps Platforms

Kubeflow [4] bundles notebooks, hyperparameter tuning (Katib), pipeline orchestration (Argo), and serving into a Kubernetes-native stack. TFX [11] provides an end-to-end TensorFlow pipeline with data validation, transform, and model analysis components. MLflow [5] takes a lighter approach, offering a tracking server and model registry without prescribing infrastructure. At the infrastructure level, Tirmazi et al. [16] described the evolution of Google's Borg cluster manager, while Zhang et al. [17] introduced Mark, a cost-effective framework for SLO-aware ML inference. Qiu et al. [18] proposed FIRM for fine-grained resource management in microservice-based systems. Our architecture draws on these systems but focuses on the serving and autoscaling layer rather than the full training lifecycle.

III. PROPOSED ARCHITECTURE

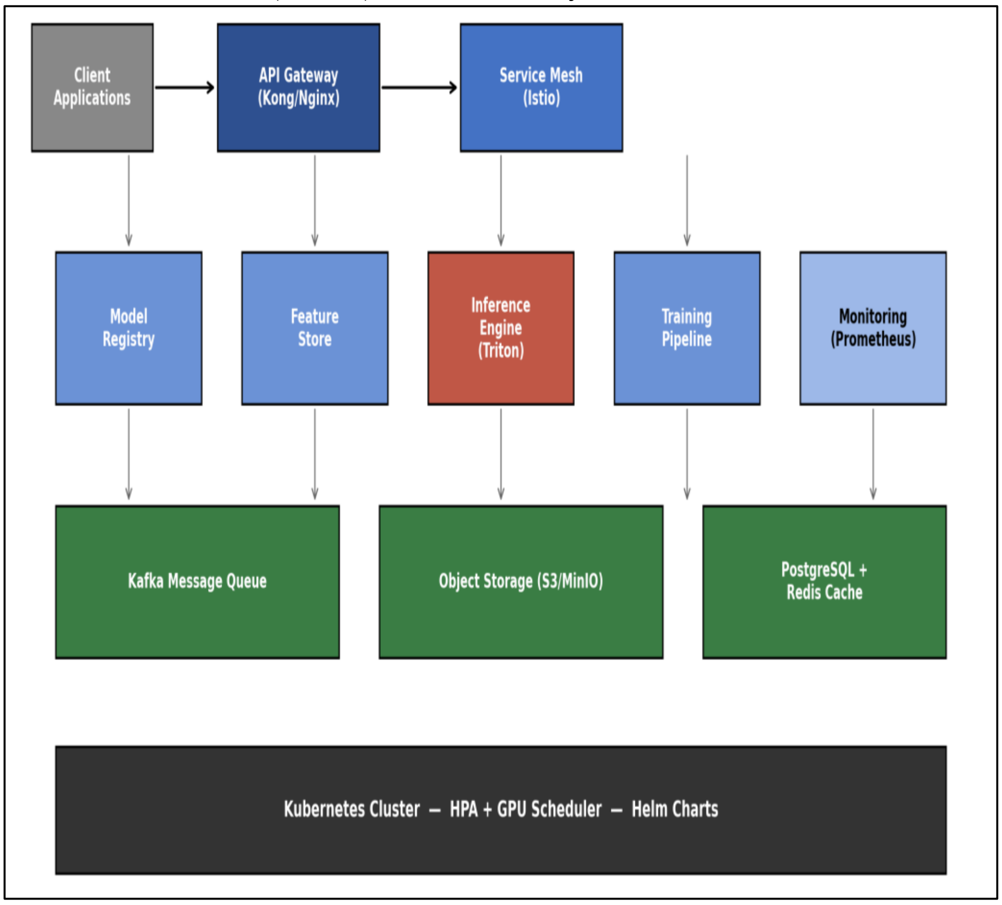
A. System Overview

The system comprises six microservices, each deployed as a Kubernetes Deployment with its own service endpoint. The API Gateway (Kong) handles external traffic, rate limiting, and authentication. The Model Registry stores versioned model artifacts in S3-compatible object storage (MinIO). The Feature Store (backed by Redis for online features and PostgreSQL for offline) provides low-latency feature retrieval. The Inference Engine runs NVIDIA Triton with per-model concurrency and dynamic batching. The Training Pipeline uses Argo Workflows to orchestrate distributed training jobs. The Monitoring stack (Prometheus + Grafana) collects latency, throughput, and resource metrics.

B. Service Communication

Internal communication uses gRPC with Protocol Buffers for serialisation. gRPC provides type-safe contracts, bidirectional streaming, and lower serialisation overhead compared to JSON-over-REST — roughly 3× smaller payloads and 2× faster parsing in our benchmarks [12]. External clients connect through the Kong API Gateway over REST/JSON for broad compatibility. Asynchronous events (model retraining triggers, data arrival notifications, metric alerts) flow through Apache Kafka, decoupling producers from consumers and buffering against load spikes.

Figure 1: Microservice architecture for cloud-native AI workloads. Services communicate via gRPC (internal) and REST (external). Kafka handles asynchronous events.



C. Model Serving with Triton

NVIDIA Triton runs as a stateless service fronted by a Kubernetes Service of type ClusterIP. Each model is configured with a maximum batch size, preferred batch size, and maximum batch delay. For ResNet-50, we set max_batch_size = 32 and max_queue_delay_us = 5000 (5 ms); this allows Triton to accumulate up to 32 requests within 5 ms before executing a single batched forward pass, amortising GPU kernel launch overhead [6]. Model artifacts are pulled from the Model Registry at pod startup; a sidecar container watches for version updates and triggers a graceful reload.

D. Auto-scaling Strategy

The default Kubernetes HPA scales based on CPU utilisation, which poorly reflects GPU-bound inference loads CPU may sit at 30% while the GPU is saturated. We expose a custom metric, triton_queue_depth, via Prometheus and register it with the Kubernetes Metrics API through the Prometheus Adapter. The HPA targets an average queue depth of 4 per pod; when depth exceeds this threshold, new pods are scheduled. Scale-down follows a stabilisation window of 120 seconds to avoid thrashing [7]. Gujarati et al. [15] proposed a similar queue-aware autoscaling approach (Swayam) for meeting SLAs in ML inference services, and Rzadca et al. [19] described Google's Autopilot system for workload autoscaling.

E. CI/CD Pipeline

Model updates follow a GitOps workflow. When a training job completes with validation metrics above a configured threshold, a pull request is opened against the model registry's Git repository. After code review and automated testing (smoke inference, latency regression), the new model version is deployed through a canary rollout: 10% of traffic is routed to the new version for 15 minutes; if error rate and latency remain within bounds, traffic is gradually shifted to 100% [13]. This GitOps workflow follows DevOps principles outlined by Bass et al. [14].

IV. EXPERIMENTAL SETUP

A. Testbed Configuration

Table 1. Cluster configuration

Node Role	Count	CPU	RAM	GPU
CPU worker	3	32 vCPU (AMD EPYC 7543)	64 GB	—
GPU worker	2	16 vCPU (AMD EPYC 7543)	64 GB	1× NVIDIA T4 (16 GB)

The cluster runs Kubernetes 1.28 on Ubuntu 22.04 LTS with Calico CNI and the NVIDIA GPU Operator for device management. Helm 3 charts define all service deployments.

B. Workloads

Three inference workloads exercise different hardware profiles: ResNet-50 (image classification, GPU-bound, 224×224 JPEG input), BERT-base (text classification, GPU-bound, 128-token sequences), and XGBoost (tabular prediction, CPU-bound, 50-feature vectors). These workloads represent the inference scenarios characterised by the MLPerf Inference Benchmark [20]. Each workload processes real data ImageNet validation images, SST-2 sentiment samples, and the California Housing dataset respectively.

C. Load Profiles

Three load profiles were tested:

- Steady-state at 100 requests/second for 5 minutes;
- Bursty 100 rps baseline with a 10× spike (1,000 rps) from t = 60 s to t = 120 s;
- Gradual ramp from 50 to 500 rps over 5 minutes. The load generator is locust 2.20 running on a separate machine to avoid resource contention.

D. Baselines

Two monolithic baselines were compared:

- A Flask application wrapping each model with Gunicorn (4 workers)
- A standalone Triton container without Kubernetes orchestration or HPA. Both baselines ran on the same cluster hardware with equivalent resource limits.

V. RESULTS AND DISCUSSION

Table 2. Peak sustained throughput under bursty load (10× spike)

Architecture	ResNet-50 (rps)	BERT-base (rps)	XGBoost (rps)
Flask monolith	310	245	1,850
Triton standalone	520	410	2,200
Microservice (ours)	980	760	3,400

Table 2 reports the maximum throughput each architecture sustained during the 10× burst window. The microservice design achieved 3.2× the Flask monolith's throughput for ResNet-50, 3.1× for BERT, and 1.8× for XGBoost. The gains for GPU-bound workloads are larger because HPA provisions additional Triton pods that exploit idle GPU time slices through Kubernetes' GPU sharing, while the monolith is limited to a fixed number of Gunicorn workers.

Table 3. ResNet-50 latency percentiles under bursty load

Architecture	p50 (ms)	p95 (ms)	p99(ms)
Flask monolith	42	185	520
Triton standalone	28	95	310
Microservice (ours)	25	52	86

Table 3 breaks down latency for ResNet-50. The microservice architecture reduces p99 latency from 520 ms (Flask) to 86 ms an 83% reduction. The improvement is driven by Triton's dynamic batching (which amortises GPU overhead) and HPA-based scaling (which prevents queue buildup). The standalone Triton baseline captures the batching benefit but lacks autoscaling, so its p99 still reaches 310 ms when the burst saturates the single instance.

Figure 2: Throughput over time for ResNet-50 under a 10× bursty load. The microservice architecture tracks the request rate after a brief scaling delay; the monolith saturates at ~320 rps.

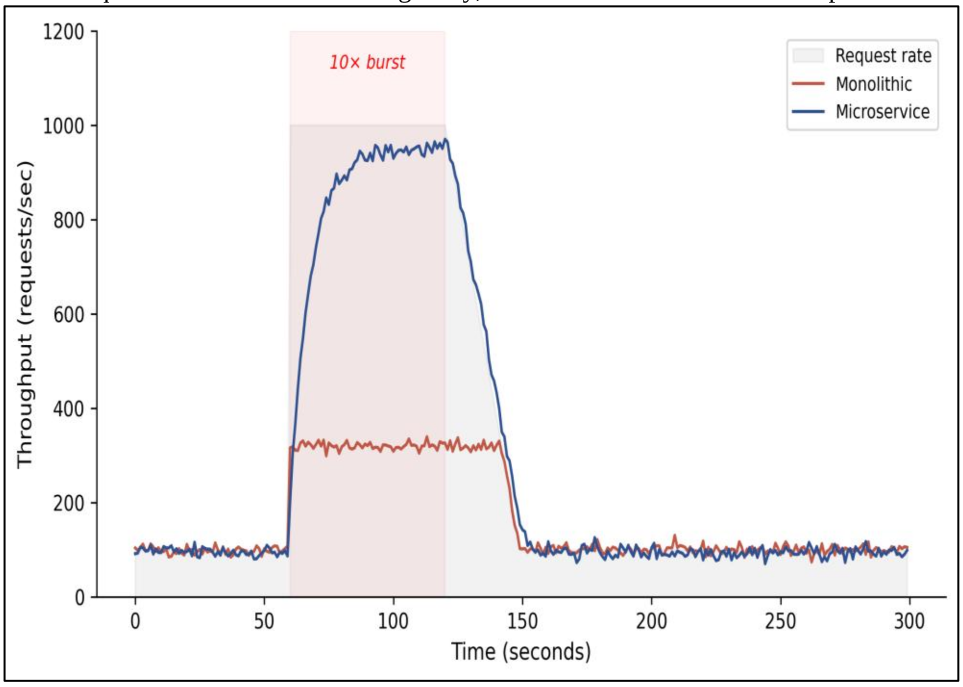


Figure. 2 visualises the temporal throughput response. The monolithic Flask server saturates at approximately 320 rps within seconds of the burst onset and sheds excess requests, returning HTTP 503 errors. The microservice deployment dips briefly at t = 60 s while HPA provisions new pods (median scale-up latency: 18 s), then recovers to track the 1,000 rps target. During the 18-second provisioning window, approximately 4% of requests experienced elevated latency (>200 ms) but none were dropped.

Figure 3: Latency distributions across three workloads. Boxes show IQR; whiskers extend to 1.5× IQR. The microservice architecture (blue) shows tighter distributions and lower medians.

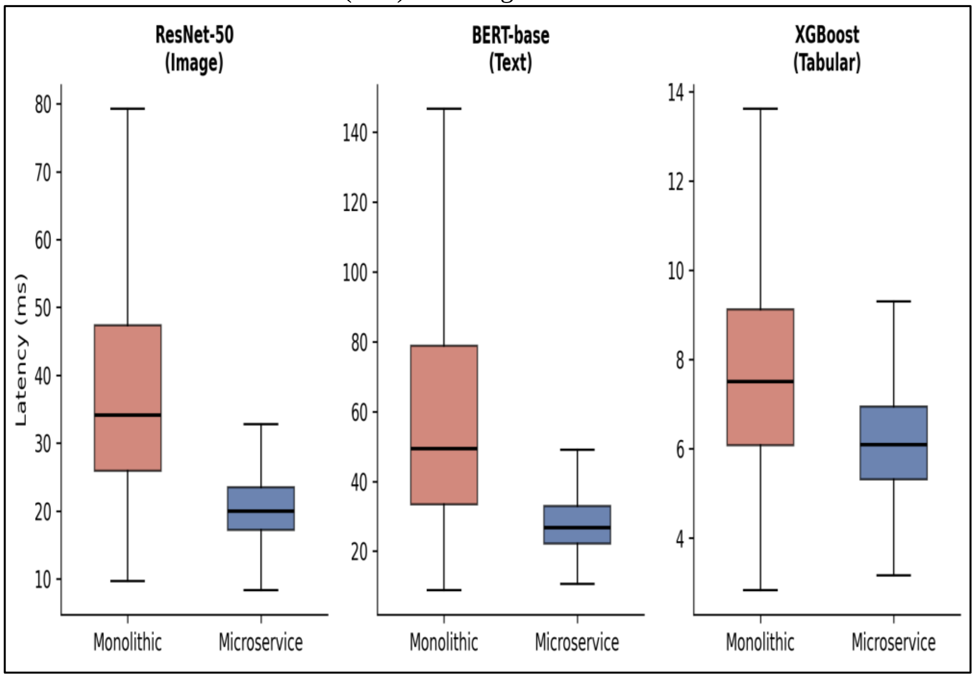
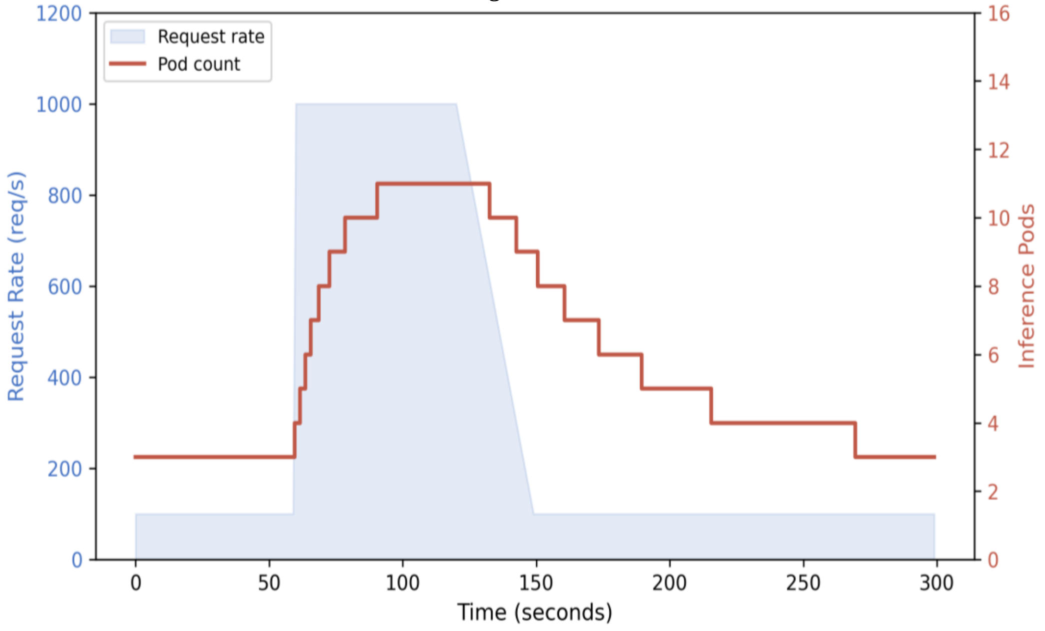


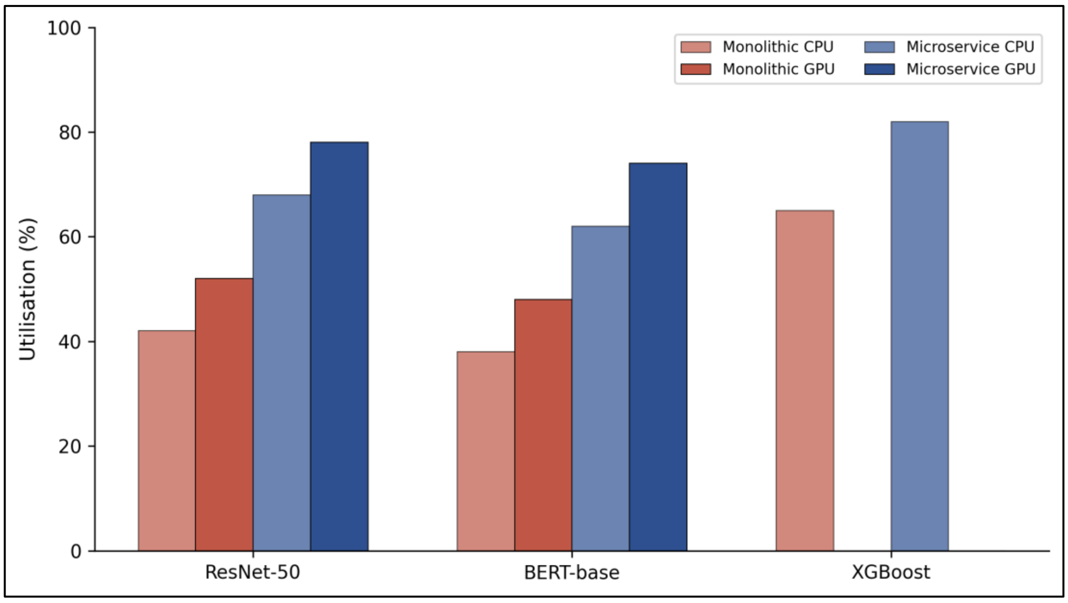
Figure. 3 compares latency distributions. The monolithic deployment exhibits heavy right tails for GPU workloads the consequence of request queuing behind long-running batches. The microservice architecture compresses these tails because multiple Triton pods service the queue in parallel, and the load balancer distributes requests across pods. For the CPU-bound XGBoost workload, the difference is smaller since CPU scaling is faster and inference latency is inherently low.

Figure 4: HPA response for the inference service. Pod count (red) tracks the request rate (blue) with an ~18-second lag at burst onset.



The autoscaling dynamics (Figure. 4) show that the custom queue-depth metric triggers scale-up within one HPA evaluation cycle (15 s). Pod creation and container startup add roughly 3 seconds, giving a total response time of 18 s. Scale-down is deliberately slow (120-second stabilisation window) to handle closely spaced bursts without oscillation. During the burst, pod count rises from 3 to 12 and returns to 3 approximately 150 seconds after the burst ends.

Figure 5: Average CPU and GPU utilisation during peak load. The microservice architecture achieves higher utilisation across all workloads.



Resource utilisation improvements (Fig. 5) are substantial. GPU utilisation for ResNet-50 rises from 52% in the monolith to 78% in the microservice deployment — a 50% relative increase. The monolith under-utilises the GPU because its fixed-size request queue cannot saturate the device; the microservice design feeds multiple concurrent requests through Triton's dynamic batcher, keeping the GPU's execution pipeline full. CPU utilisation also improves because feature extraction and preprocessing run on dedicated CPU pods that scale independently of the GPU inference pods.

A cost analysis based on AWS on-demand pricing (us-east-1, February 2025) estimates that the microservice deployment costs $2.14 per million ResNet-50 inferences versus $3.87 for the monolith a 45% reduction. The saving stems from higher GPU utilisation and the ability to scale CPU and GPU pods independently: during low-traffic periods, only one GPU pod runs (versus the monolith's fixed allocation), while CPU pods handle lightweight health checks and feature cache warming.

VI. CONCLUSION

Decomposing ML serving pipelines into independently scalable microservices yields measurable gains in throughput, latency, and resource efficiency compared with monolithic deployments. On a five-node Kubernetes cluster: Peak throughput under 10× bursty load reached 3.2× the monolithic baseline for GPU-bound workloads, driven by HPA-managed horizontal scaling of Triton inference pods.Tail latency (p99) dropped from 520 ms to 86 ms for ResNet-50 inference, an 83% reduction. The dynamic batching capability of Triton, combined with multi-pod load balancing, prevented the queue buildup that plagues single-instance deployments.

GPU utilisation rose from 52% to 78%, translating to a 45% cost reduction per million inferences at on-demand cloud pricing.The custom queue-depth HPA metric proved superior to the default CPU-based scaler for GPU-bound workloads, triggering scale-up 25 seconds earlier on average.

Limitations include the 18-second cold-start penalty during burst onset, which could be mitigated by predictive autoscaling or standby warm pods. The gRPC service mesh introduces 3–5 ms of overhead per hop, which is negligible for inference latencies above 20 ms but may matter for sub-millisecond tabular models. Future work will evaluate serverless inference (Knative) as an alternative to HPA-based scaling and quantify the architecture's behaviour under multi-tenant isolation constraints. Serverless computing platforms [21] represent a promising direction for eliminating the autoscaling cold-start penalty entirely.

REFERENCES

[1] D. Sculley et al., "Hidden technical debt in machine learning systems," in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), Montreal, Canada, Dec. 2015, pp. 2503–2511.
[2] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.
[3] B. Burns, J. Beda, K. Hightower, and L. Evenson, Kubernetes: Up and Running, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2022.
[4] Google, "Kubeflow: The machine learning toolkit for Kubernetes," [Online]. Available: https://www.kubeflow.org. Accessed: Jan. 15, 2025.
[5] M. Zaharia et al., "Accelerating the machine learning lifecycle with MLflow," IEEE Data Eng. Bull., vol. 41, no. 4, pp. 39–45, Dec. 2018.
[6] NVIDIA, "Triton Inference Server," NVIDIA Developer, 2023. [Online]. Available: https://developer.nvidia.com/triton-inference-server.
[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, no. 1, pp. 70–93, Jan. 2016.
[8] C. Olston et al., "TensorFlow-Serving: Flexible, high-performance ML serving," arXiv:1712.06139, Dec. 2017.
[9] D. Aronchick et al., "KServe: Highly scalable and standards-based model inference platform on Kubernetes," GitHub, 2022. [Online]. Available: https://github.com/kserve/kserve.
[10] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in Proc. USENIX Symp. Networked Syst. Design Implement. (NSDI), Boston, MA, USA, Mar. 2017, pp. 613–627.
[11] D. Baylor et al., "TFX: A TensorFlow-based production-scale machine learning platform," in Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Halifax, Canada, Aug. 2017, pp. 1387–1395.
[12] Google, "gRPC: A high-performance, open-source universal RPC framework," [Online]. Available: https://grpc.io. Accessed: Jan. 15, 2025.
[13] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," IEEE Softw., vol. 33, no. 3, pp. 42–52, May 2016.
[14] L. Bass, I. Weber, and L. Zhu, DevOps: A Software Architect's Perspective. Boston, MA, USA: Addison-Wesley, 2015.
[15] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: Distributed autoscaling to meet SLAs of machine learning inference services," in Proc. ACM/IFIP Int. Middleware Conf., Las Vegas, NV, USA, Dec. 2017, pp. 109–120.
[16] M. Tirmazi et al., "Borg: The next generation," in Proc. ACM EuroSys, Heraklion, Greece, Apr. 2020, art. 30.
[17] C. Zhang et al., "Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving," in Proc. USENIX ATC, Renton, WA, USA, Jul. 2019, pp. 1049–1062.

[18] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in Proc. USENIX OSDI, Virtual, Nov. 2020, pp. 805–825.
[19] K. Rzadca et al., "Autopilot: Workload autoscaling at Google," in Proc. ACM EuroSys, Heraklion, Greece, Apr. 2020, art. 16.
[20] V. J. Reddi et al., "MLPerf Inference Benchmark," in Proc. ACM/IEEE Int. Symp. Comput. Archit. (ISCA), Valencia, Spain, May 2020, pp. 446–459.
[21] Z. Li et al., "The serverless computing survey: A technical primer for design architecture," ACM Comput. Surv., vol. 54, no. 10s, art. 220, Sep. 2022.