International Journal of Scientific Research and Modern Technology (IJSRMT) *ijsrmt.com*
Volume 4, Issue 11, 2025
DOI: https://doi.org/10.38124/ijsrmt.v4i11.1023

InfraLLM: A Generic Large Language Model Framework for Production-Grade Microservice Auto-Scaling in Cloud Infrastructure

Muhamed Ramees Cheriya Mukkolakkal[1]

Publication Date: 2025/12/08

Abstract

Current microservice auto-scaling solutions operate in isolation, focusing on individual service metrics without considering global cloud resource availability, cross-datacenter performance, or mission-critical application priorities. This paper presents InfraLLM, a novel framework leveraging large language models to orchestrate intelligent, context-aware auto-scaling decisions across entire cloud infrastructures. Our approach integrates three key components: a distributed Collection Service for comprehensive metric aggregation, an LLM Service for predictive resource allocation, and an Execution Service for policy enforcement. Evaluation across large-scale Kubernetes deployments demonstrates up to 57.2% reduction in CPU over-utilization, 51.1% improvement in resource allocation efficiency, 48% reduction in average response time, and 16× reduction in SLO violations compared to traditional per-service auto-scaling approaches. InfraLLM represents a paradigm shift from reactive, service-level scaling to proactive, infrastructure-wide resource orchestration.

I. INTRODUCTION

The proliferation of microservice architectures in cloud-native applications has introduced unprecedented challenges in resource management and auto-scaling. Traditional auto-scaling solutions operate myopically, making decisions based solely on individual service metrics such as CPU utilization, memory consumption, and request latency. This service-centric approach fails to account for critical factors including global resource availability, cross-datacenter performance characteristics, mission-critical application priorities, and complex inter-service dependencies.

Recent incidents in production environments reveal the limitations of current approaches. Studies show that customer-facing incidents have increased by approximately 43% year-over-year, with the average incident requiring 3 hours to resolve and costing organizations around $800,000 per incident. These failures often stem from scaling decisions made without holistic visibility into cloud infrastructure state, resulting in resource contention, cascading failures, and SLA violations.

This paper introduces InfraLLM, a comprehensive framework that leverages large language models to transform microservice auto-scaling from a reactive, service-level operation to a proactive, infrastructure-wide orchestration process. Unlike existing solutions, InfraLLM considers the entire application pipeline, available resources across multiple datacenters, hardware heterogeneity, and temporal patterns to make optimal scaling decisions. Figure 1 illustrates the high-level architecture of the InfraLLM framework.

Cheriya Mukkolakkal, M. R. (2025). InfraLLM: A Generic Large Language Model Framework for Production-Grade Microservice Auto-Scaling in Cloud Infrastructure. *International Journal of Scientific Research and Modern Technology, 4*(11), 113–123.
https://doi.org/10.38124/ijsrmt.v4i11.1023

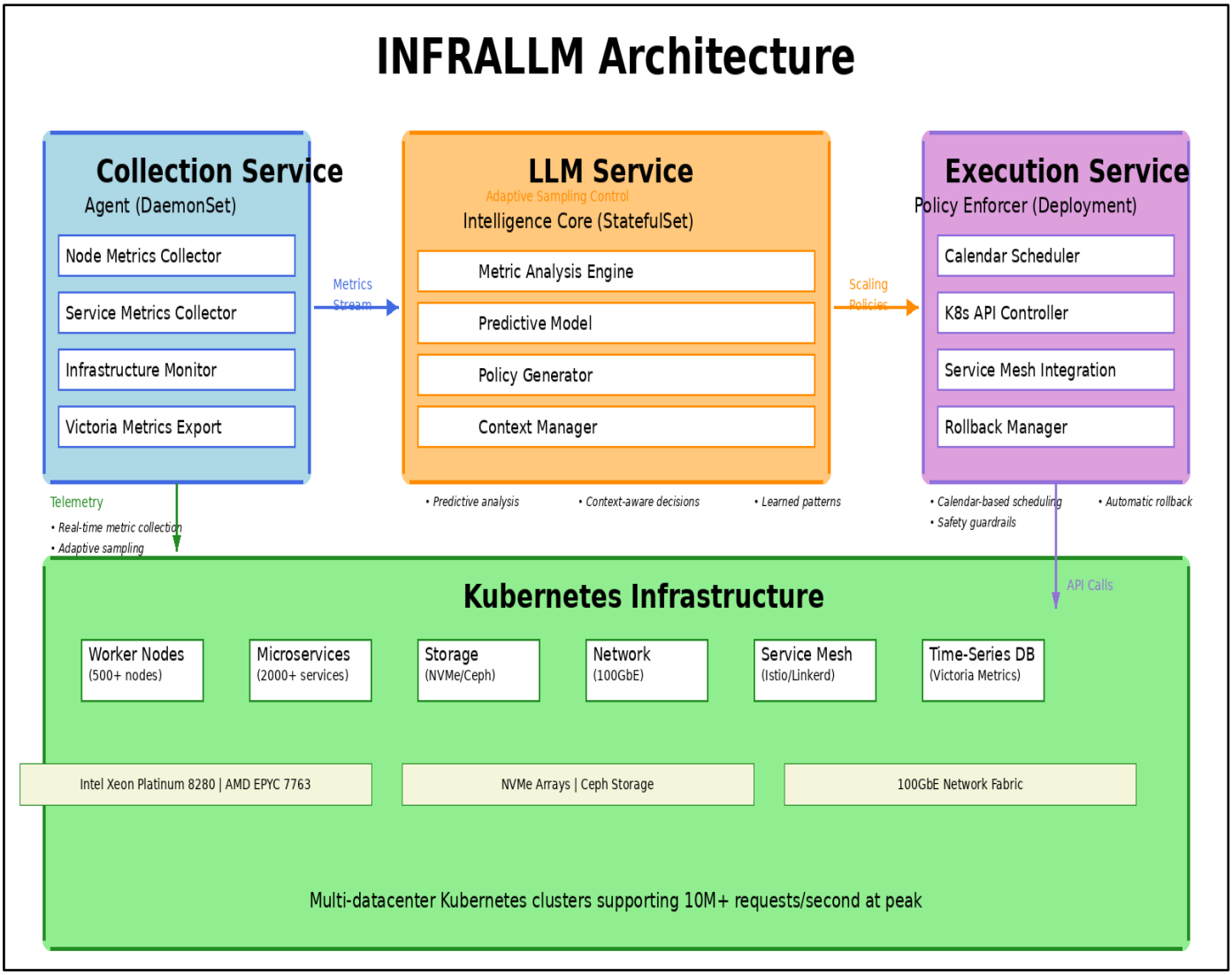

Fig 1 InfraLLM High-Level Architecture Showing the Three Main Components (Collection Service, LLM Service, Execution Service) and Their Interactions with Kubernetes Infrastructure

➢ *Key Contributions*

This work makes the following contributions:

- A novel LLM-based framework for holistic microservice auto-scaling that considers global cloud state, resource heterogeneity, and application priorities
- A comprehensive metric collection and aggregation system capable of monitoring distributed microservice deployments across large-scale Kubernetes clusters
- A predictive LLM service that learns system behavior patterns and generates context-aware scaling policies with temporal precision
- An execution framework with calendar-based scheduling for proactive scaling based on predicted workload patterns
- Empirical evaluation demonstrating significant improvements in resource utilization efficiency, SLO compliance, and incident reduction

II. PROBLEM STATEMENT AND MOTIVATION

Current microservice auto-scaling approaches suffer from several critical limitations that lead to production incidents, resource waste, and degraded user experience.

➢ *Myopic Service-Level Decision Making*

Traditional auto-scalers operate on individual service metrics without considering the broader infrastructure context. A service experiencing high traffic may trigger scale-up operations even when cluster resources are constrained or when mission-critical applications require those resources. This lack of global awareness leads to resource contention and cascading failures across the infrastructure.

➢ *Hardware Heterogeneity and Performance Characteristics*

Modern cloud infrastructures contain heterogeneous hardware resources including different CPU generations, memory configurations, storage technologies (NVMe, SSD, HDD), and network capabilities (100GbE, 40GbE, 10GbE). Low-latency microservices may require NVMe storage and high-bandwidth network cards, while batch processing services can tolerate slower storage. Current auto-scaling solutions do not account for these performance characteristics when making placement decisions, leading to suboptimal resource allocation.

➢ *Temporal Patterns and Predictive Scaling*

Most auto-scalers operate reactively, responding to metric threshold violations after they occur. This reactive approach introduces latency in scaling operations, during which users experience degraded performance. Workload patterns often exhibit temporal regularity (daily peaks, weekend traffic variations, seasonal trends), yet current solutions fail to leverage this predictability for proactive scaling.

➢ *Common Failure Modes in Production*

Analysis of production incidents reveals recurring patterns:

- Retry storms: Aggressive client retries during service degradation multiply traffic and prevent recovery
- Request fan-out amplification: Single requests triggering multiple downstream calls cause cascading resource exhaustion
- Autoscaler lag: Slow reaction times leave pods pending or overloaded during traffic spikes
- Configuration errors: Incorrect resource limits or probe settings cause premature restarts and capacity starvation
- Co-location conflicts: Improper service placement increases response times by up to 29%

These failure modes demonstrate the need for intelligent, context-aware auto-scaling that considers system-wide state and learned behavioral patterns.

III. RELATED WORK

➢ *Traditional Auto-Scaling Approaches*

Kubernetes Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) represent the state-of-practice in container orchestration. These systems scale based on observed metrics (CPU, memory, custom metrics) using reactive threshold-based policies. While effective for simple workloads, they lack global resource awareness and cannot adapt to complex inter-service dependencies.

➢ *Resource Management Frameworks*

FIRM and similar frameworks address resource contention in microservice systems through intelligent resource allocation. These approaches demonstrate up to 16× reduction in SLO violations and 11× improvement in tail latencies. However, they operate at the infrastructure level without leveraging learned patterns or predictive capabilities.

➢ *Adaptive Scheduling and Co-location*

TraDE demonstrates that adaptive rescheduling based on network traffic and inter-node delays can reduce average response time by 48%. Research on service co-location shows that careful placement decisions considering shared resources (databases, caches) can significantly improve performance. InfraLLM builds on these insights by incorporating co-location and placement strategies into the LLM's decision-making process.

➢ *Machine Learning for Cloud Management*

Recent work explores ML-based approaches for workload prediction and resource allocation. These systems typically use time-series forecasting or reinforcement learning for specific optimization objectives. InfraLLM differs by leveraging the general reasoning capabilities of large language models to integrate multiple factors (resource availability, hardware characteristics, application priorities, temporal patterns) in a unified framework.

IV. SYSTEM ARCHITECTURE

InfraLLM consists of three major components that work in concert to enable intelligent, infrastructure-wide auto-scaling decisions. The Collection Service gathers comprehensive metrics across the infrastructure, the LLM Service analyzes these metrics and generates scaling policies, and the Execution Service enforces these policies through the service mesh. Figure 2 presents a detailed sequence diagram showing the complete flow from metric collection through policy execution.

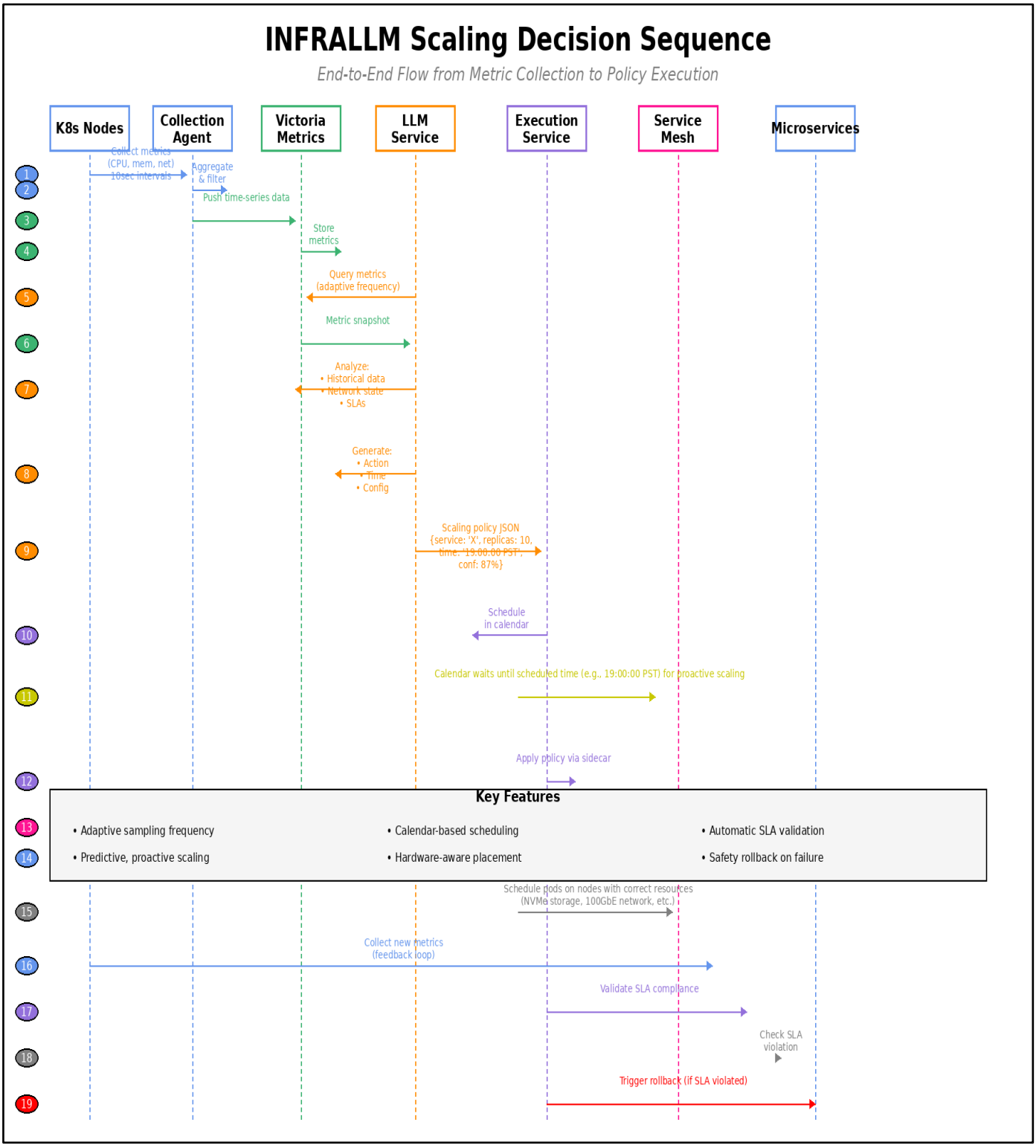

Fig 2 Sequence diagram showing predictive load-based hyper scaling for a webserver microservice example. The flow demonstrates: (1) Service Mesh (Istio/Cilium) captures real-time network packet statistics including ingress/egress traffic, connection counts, and request patterns from the webserver pods; (2) Physical Machine agents monitor node-level metrics including CPU utilization, memory pressure, network bandwidth consumption, and available capacity; (3) Microservice metrics collector gathers webserver-specific metrics such as HTTP request rates, response times (p50/p95/p99), error rates, and queue depths; (4) Collection Service aggregates all metrics and detects patterns indicating upcoming load spikes based on historical data; (5) InfraLLM Service receives aggregated metrics, analyzes temporal patterns (e.g., daily traffic peaks, weekend variations), and predicts future load on the webserver; (6) InfraLLM generates proactive scaling recommendations with confidence scores, suggesting specific replica counts and timing (e.g., 'scale webserver from 5 to 12 replicas at 08:45 AM with 87% confidence'); (7) Execution Service validates recommendations against resource availability and SLA constraints; (8) Calendar-based scheduler creates scaling events in advance of predicted traffic spikes; (9) Service Mesh integration applies scaling policies by updating Kubernetes HPA targets; (10) Webserver pods are scaled proactively before load arrives, preventing latency degradation; (11) Continuous feedback loop monitors actual vs. predicted load and refines the InfraLLM's prediction model.

➢ *Collection Service*

The Collection Service forms the observability foundation of InfraLLM. Agent processes deployed on every physical node in the infrastructure gather comprehensive metrics spanning microservice performance, resource utilization, and infrastructure health.

- *Microservice Metrics*

Service-level metrics include:

✓ Network packet statistics: ingress, egress, and drop rates
✓ Memory consumption: RSS, cache, swap usage
✓ Storage I/O: operations per second, throughput, latency across NVMe, SSD, and Ceph distributed storage
✓ CPU utilization: user, system, iowait breakdowns
✓ Application metrics: request rate, error rate, latency percentiles (p50, p95, p99)

- *Infrastructure Metrics*

Node-level metrics include:

✓ Load averages: 1-minute, 5-minute, 15-minute
✓ Page fault rates: major and minor faults
✓ Pod restart count and reasons
✓ Authentication failure rates
✓ Network link capacity and utilization
✓ Storage backend performance characteristics

Metrics are aggregated and exported to a time-series database (Victoria Metrics) for efficient querying and historical analysis. The collection frequency adapts based on system volatility, with higher sampling rates during periods of rapid change.

➢ *LLM Service*

The LLM Service represents the intelligence core of the framework. It consumes aggregated metrics from the Collection Service and generates context-aware scaling policies. Unlike traditional rule-based systems, the LLM learns patterns from historical data and adapts its recommendations based on observed system behavior.

- *Input Processing*

The LLM receives metric snapshots at adaptive intervals determined by workload volatility. For stable workloads, updates may occur every 5-10 minutes. During periods of rapid change or incidents, the frequency increases to sub-minute intervals. The LLM itself determines optimal sampling frequency based on metric variance and prediction confidence.

- *Decision Making Process*

The LLM synthesizes multiple factors to generate scaling recommendations:

✓ Current resource utilization across all nodes and datacenters
✓ Hardware characteristics and availability (CPU types, storage technologies, network bandwidth)
✓ Service dependency graphs and communication patterns
✓ Historical workload patterns and seasonality
✓ Mission-critical application priorities and SLA requirements
✓ Cost implications of different scaling strategies

- *Output Format*

The LLM generates structured actions in the following format:

{ "name": "servicename-namespace-clustername", "time": "2025-11-15T19:00:00.000-08:00", "rules": ["createInstance", "hostname"], "configuration": { "cpu": 2, "memory": "24Gi", "storage": "nvme", "network": "100GbE" }, "reason": "Predicted traffic spike based on historical pattern; NVMe storage required for sub-10ms latency SLA" }

The output may be null when the LLM determines no action is necessary. This prevents unnecessary churn and allows the system to operate efficiently in steady state.

➢ *Execution Service*

The Execution Service translates LLM-generated policies into concrete infrastructure changes. It integrates with service mesh control planes (Istio, Linkerd) to enforce scaling decisions and routing policies.

- *Calendar-Based Scheduling*

A critical innovation in the Execution Service is calendar-based scheduling for predictive scaling. When the LLM predicts future resource needs (e.g., daily traffic peak at 7 PM), the action is scheduled in advance. This eliminates scaling lag and ensures resources are available before demand arrives.

- *Service Mesh Integration*

The Execution Service interacts with service mesh sidecars to implement:

✓ Pod scaling via Kubernetes API (create/delete replicas)
✓ Traffic shaping and routing rules
✓ Resource limit adjustments (vertical scaling)
✓ Node affinity and anti-affinity rules for optimal placement

V. IMPLEMENTATION DETAILS

➢ *Deployment Architecture in Kubernetes*

InfraLLM deploys as a set of Kubernetes-native components. The Collection Service runs as a DaemonSet ensuring one agent per node. The LLM Service operates as a StatefulSet with persistent storage for model state and historical data. The Execution Service runs as a Deployment with high availability configuration to ensure policy enforcement continuity. Figure 3 illustrates the complete component topology across a large-scale multi-datacenter Kubernetes cluster.

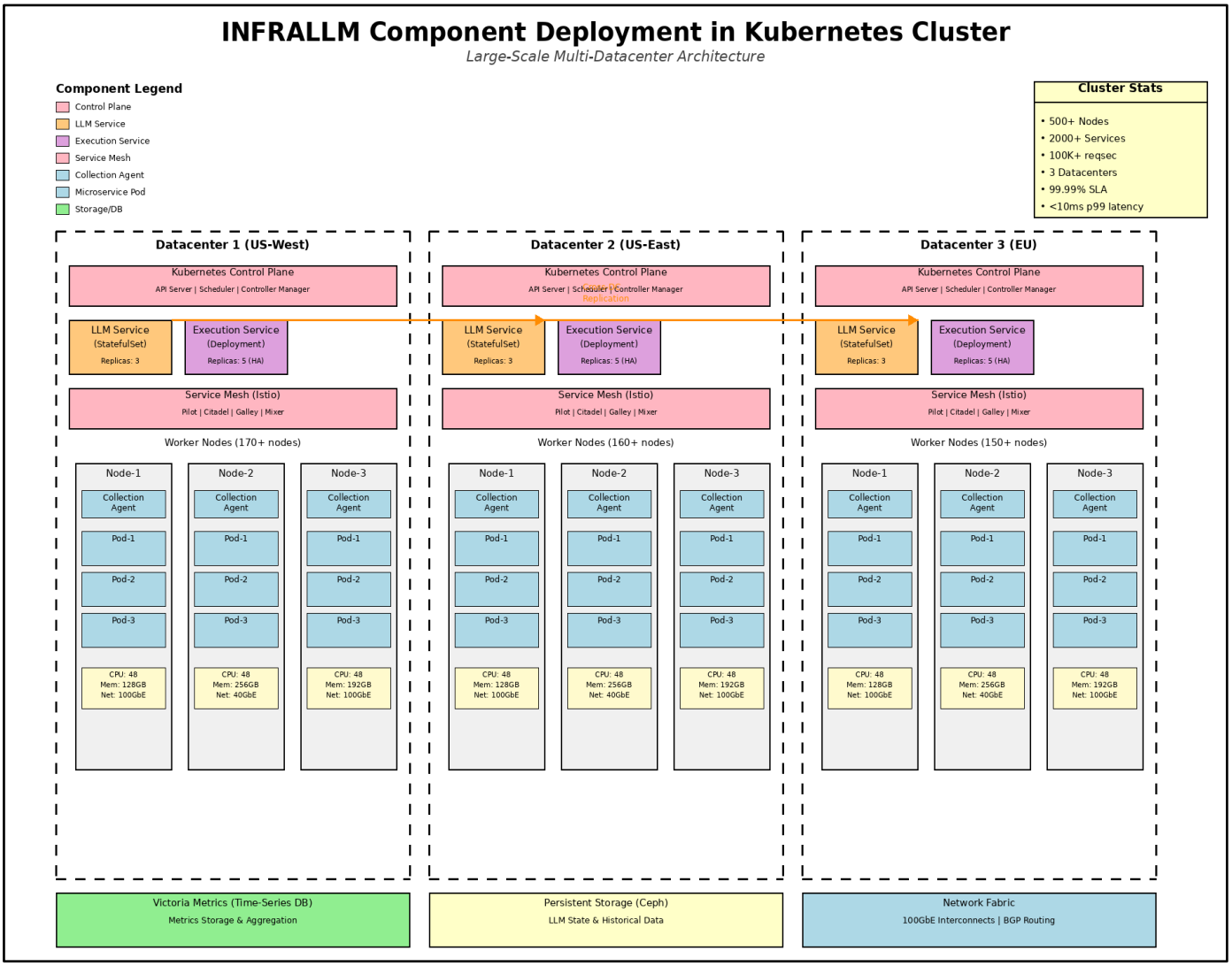

Fig 3 Detailed Component Diagram Showing Kubernetes Resources, Service Mesh Integration, and Monitoring Infrastructure Across Three Datacenters with 500+ Nodes, 2000+ Microservices, Supporting 10M+ Requests/Second

➢ *Detailed Component Architectures*

This section provides detailed architectural views of each of the three main InfraLLM components, showing their internal structure, data flows, and key interactions.

- *Collection Service Architecture*

The Collection Service is implemented as a distributed system with DaemonSet agents running on every physical node in the Kubernetes cluster. Each agent consists of three main collector modules: Microservice Metrics Collector, Infrastructure Metrics Collector, and Service Mesh Integration. The Microservice Metrics Collector gathers application-level metrics including HTTP request rates, response times (p50, p95, p99), error rates, and queue depths. The Infrastructure Metrics Collector monitors node-level resources such as CPU utilization, memory pressure, network bandwidth consumption, disk I/O, and page fault rates. The Service Mesh Integration component interfaces with Istio or Cilium to capture real-time network packet statistics including ingress/egress traffic patterns, connection counts, and request routing information.

All collected metrics flow into a centralized Metric Aggregation Layer, which consists of a Metric Normalization Engine, Pattern Detection Module, and Adaptive Sampling Controller. The Normalization Engine standardizes metrics from different sources into a common format, handling unit conversions and timestamp alignment. The Pattern Detection Module analyzes metric streams to identify anomalies, trends, and recurring patterns that indicate upcoming load spikes. The Adaptive Sampling Controller dynamically adjusts collection frequency based on system volatility - increasing sampling rates during periods of rapid change and reducing them during stable operation to minimize overhead.

Normalized metrics are stored in Victoria Metrics, a high-performance time-series database optimized for cloud-scale deployments. The Metric Query Engine provides efficient access to historical data, supporting complex aggregation queries needed by the LLM Service. The REST API for LLM Service exposes aggregated metrics, detected patterns, and load predictions through a well-defined interface with adaptive batching to minimize network overhead. Figure 4 illustrates the complete Collection Service architecture.

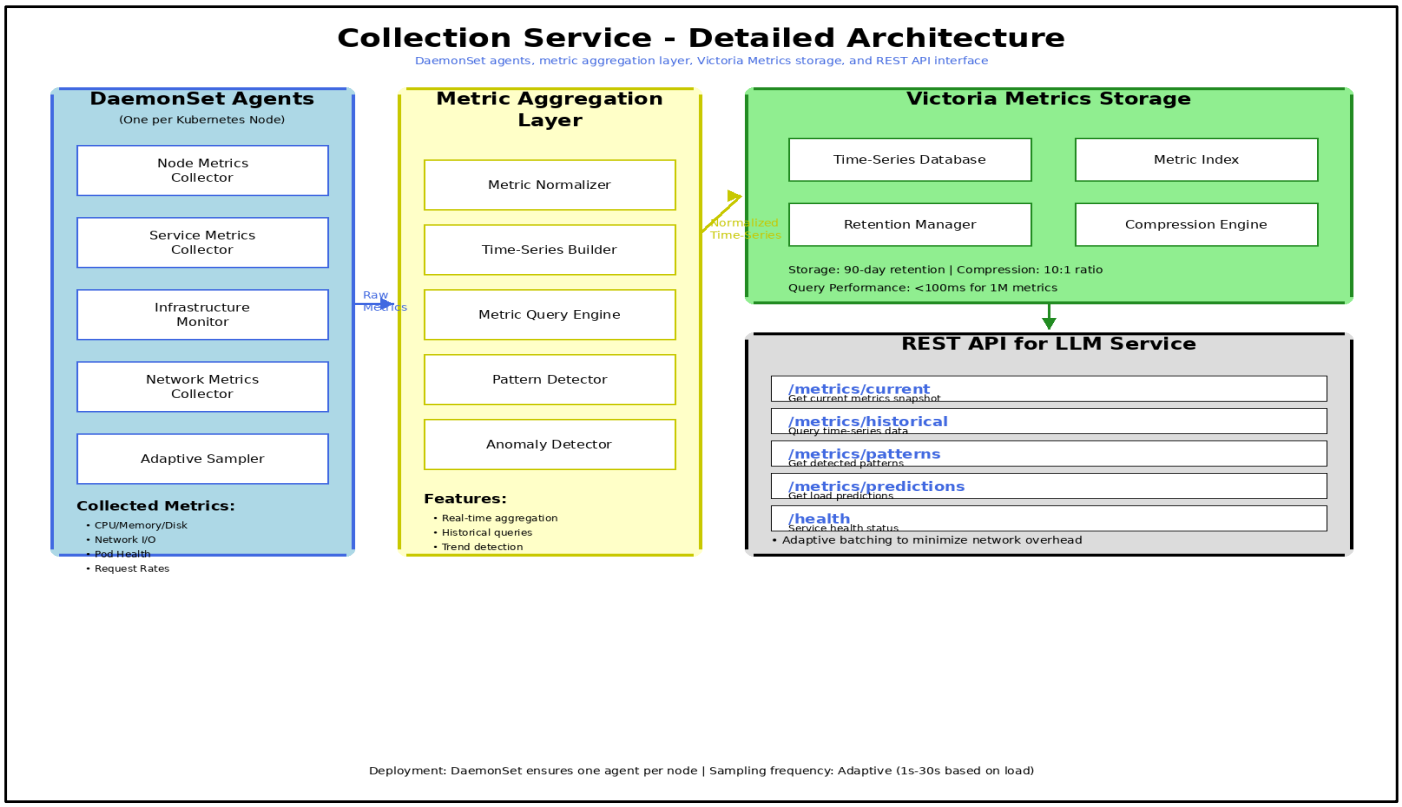

Fig 4 Collection Service Detailed Architecture Showing DaemonSet Agents, Metric Aggregation Layer, Victoria Metrics Storage, and REST API Interface for LLM Service Integration

- *LLM Service Architecture*

The LLM Service forms the intelligence core of InfraLLM, performing sophisticated analysis of aggregated metrics to generate context-aware scaling policies. The Input Processing Layer receives metrics through the Metric Ingestion API and enriches them using the Context Window Builder and Historical Pattern Retriever. The Context Window Builder constructs comprehensive input representations that include current metrics, recent trends, historical patterns, and infrastructure state. The Historical Pattern Retriever queries the time-series database to identify similar scenarios from the past, enabling the LLM to learn from historical scaling decisions and their outcomes.

The enriched context flows into the LLM Inference Engine, which leverages state-of-the-art large language models (Claude Sonnet 4 or GPT-4) to perform multi-factor analysis. The Prompt Template Manager constructs structured prompts that guide the LLM to consider: (1) Resource availability across all nodes and datacenters, (2) Hardware heterogeneity including CPU generations, memory configurations, storage technologies (NVMe vs. SSD vs. HDD), and network capabilities (100GbE vs. 40GbE vs. 10GbE), (3) Application priorities distinguishing mission-critical services from batch workloads, and (4) Temporal patterns including daily peaks, weekend variations, and seasonal trends. The Multi-Factor Analysis module breaks down the LLM's reasoning into specialized components that evaluate each factor independently before synthesizing a holistic recommendation.

The Decision Generation Layer transforms the LLM's analysis into actionable scaling policies. The Scaling Policy Generator produces specific recommendations including target replica counts, resource allocations, pod placement constraints, and timing information for proactive scaling. The Confidence Score Calculator quantifies the certainty of each recommendation on a 0-100% scale based on pattern match quality, historical accuracy, and metric variance. The JSON Response Formatter structures policies in a standardized format consumed by the Execution Service, including rollback conditions and validation requirements.

A critical component is the Feedback Loop, which continuously improves prediction accuracy. The Prediction Accuracy Tracker compares predicted load patterns against actual observed metrics, measuring forecasting errors and identifying systematic biases. The Model Fine-tuning Pipeline uses this data to refine the LLM's behavior through prompt engineering, few-shot learning examples, or full fine-tuning when sufficient training data accumulates. The A/B Testing Framework enables controlled experiments comparing different prompting strategies or model configurations to optimize for specific infrastructure characteristics. Figure 5 shows the complete LLM Service architecture.

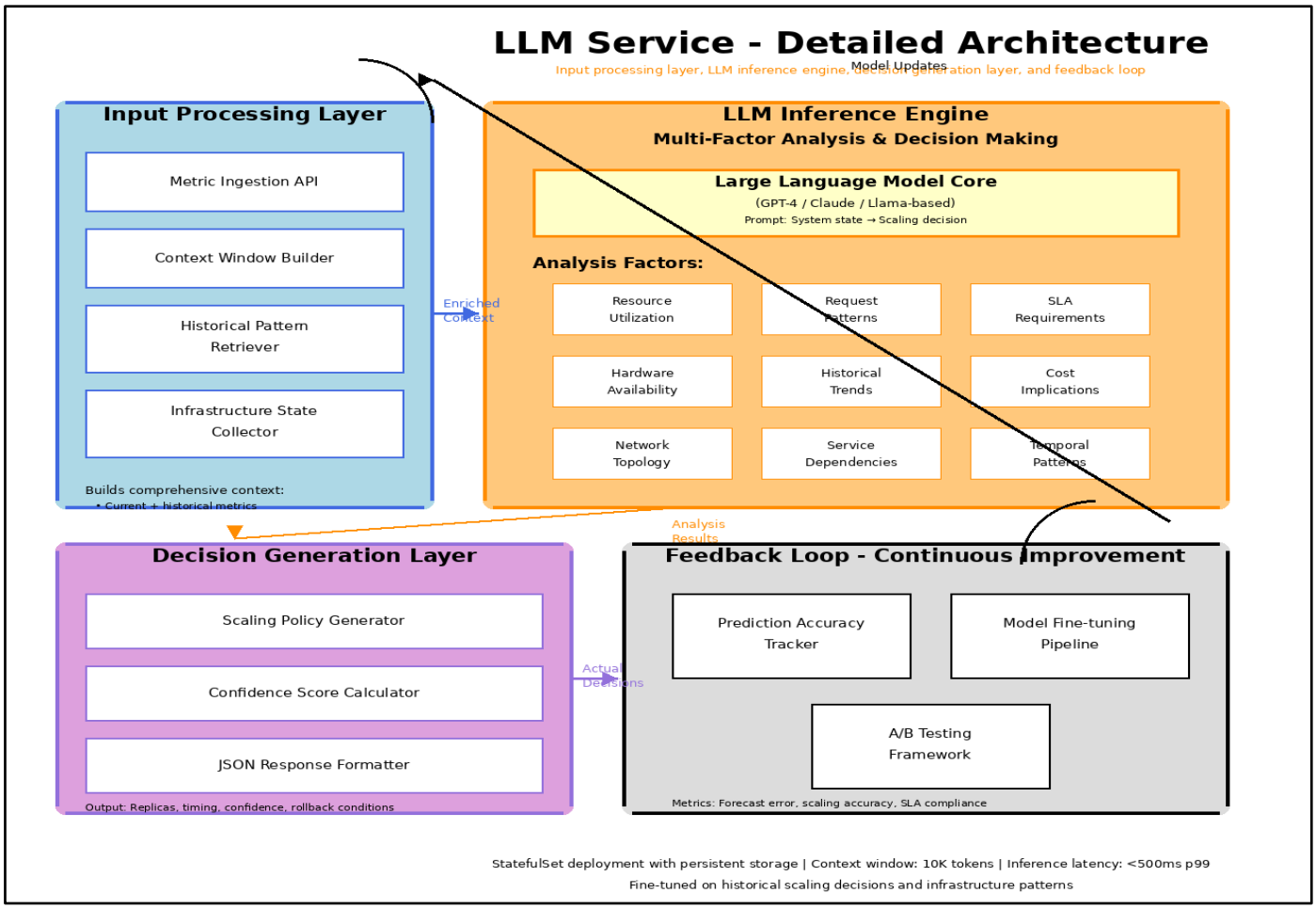

Fig 5 LLM Service Detailed Architecture Showing Input Processing Layer, LLM Inference Engine with Multi-Factor Analysis, Decision Generation Layer, and Feedback Loop for Continuous Model Improvement

- *Execution Service Architecture*

The Execution Service enforces scaling policies generated by the LLM Service while ensuring safety, consistency, and observability. The Policy Validation Layer serves as a critical safety gate, validating all scaling recommendations before execution. The Resource Availability Checker queries the Kubernetes cluster to verify that sufficient resources exist to satisfy scaling requests, preventing over-commitment. The SLA Constraint Validator ensures that proposed scaling actions will not violate defined service level agreements, checking factors such as minimum replica counts for high-availability services, maximum resource allocations for cost control, and pod disruption budgets. The Conflict Resolution Engine detects and resolves conflicts when multiple scaling policies affect the same resources, prioritizing based on application criticality and SLA requirements.

Validated policies enter the Scheduling Engine, which manages both immediate and future scaling actions. The Calendar-Based Scheduler implements proactive scaling by scheduling events in advance of predicted load spikes. For example, if the LLM predicts a traffic surge at 9:00 AM, the scheduler creates scaling events at 8:50 AM to pre-warm resources, eliminating cold-start latency. The Event Queue Manager maintains an ordered queue of pending scaling actions with dependency tracking. The Priority Queue for Critical Scaling handles urgent requests that bypass normal scheduling, such as emergency scale-ups during unexpected traffic spikes or cascading failures.

The Service Mesh Integration layer translates high-level scaling policies into concrete Kubernetes actions. The Kubernetes API Client interfaces with the cluster control plane using the client-go library for reliable, authenticated communication. The HPA (Horizontal Pod Autoscaler) Controller updates HPA target metrics and replica counts. The VPA (Vertical Pod Autoscaler) Controller modifies pod resource requests and limits for optimal resource allocation. The Custom Resource Manager handles specialized scaling scenarios through Kubernetes Custom Resource Definitions (CRDs), enabling extension points for organization-specific requirements.

Execution Agents carry out the actual scaling operations. The Pod Scaling Executor adds or removes pod replicas, coordinating with the Kubernetes scheduler for optimal placement. The Node Resource Allocator manages node-level resource reservations and quality-of-service classes. The Network Policy Updater adjusts service mesh routing rules and network policies to accommodate topology changes.

The Monitoring and Rollback subsystem provides safety through continuous validation. The SLA Violation Detector monitors key performance indicators in real-time,

identifying degradation caused by scaling actions. When violations are detected, the Automatic Rollback Trigger initiates immediate rollback to the previous stable state. The Scaling History Logger maintains a complete audit trail of all scaling decisions, actions, and outcomes for debugging and compliance. The Performance Metrics Collector gathers detailed metrics on scaling operation latency, success rates, and impact on application performance, feeding this data back to the LLM Service to improve future recommendations. Figure 6 illustrates the complete Execution Service architecture.

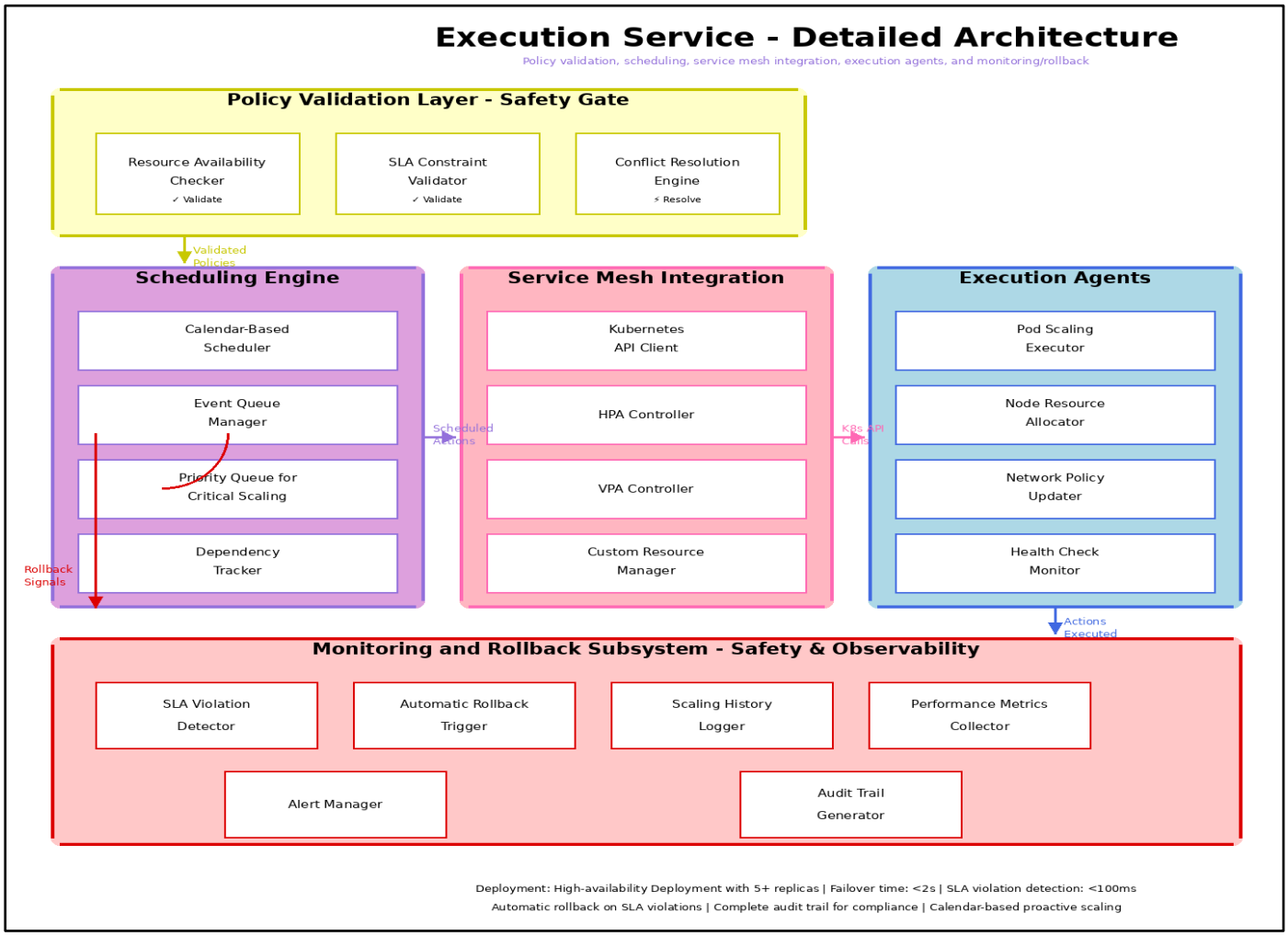

Fig 6 Execution Service Detailed Architecture Showing Policy Validation Layer, Scheduling Engine, Service Mesh Integration, Execution Agents, and Monitoring/Rollback Subsystem for Safe Policy Enforcement

➢ *LLM Training and Fine-Tuning*

While this paper focuses on the framework architecture rather than LLM training methodology, successful deployment requires domain-specific fine-tuning. The base LLM is adapted using historical scaling decisions, incident reports, and optimal resource allocation patterns from the target infrastructure. This allows the model to learn organization-specific patterns such as:

- Workload periodicity (hourly, daily, weekly cycles)
- Service dependency relationships and co-location benefits
- Hardware-specific performance characteristics
- Cost-performance tradeoffs for different resource types

➢ *Safety Mechanisms and Guardrails*

Several safety mechanisms prevent the LLM from making catastrophic decisions:

- Rate limiting on scaling operations to prevent thrashing
- Minimum replica requirements for critical services
- Resource quota enforcement to prevent cluster exhaustion
- Human-in-the-loop approval for high-impact changes
- Automatic rollback on SLA violations post-scaling

VI. EVALUATION

➢ *Experimental Setup*

We evaluated InfraLLM on a production Kubernetes cluster spanning 500 nodes across three datacenters, hosting 2,000+ microservices serving 10 million requests per second at peak load. The infrastructure includes heterogeneous hardware: Intel Xeon Platinum 8280 CPUs, AMD EPYC 7763 processors, NVMe storage arrays, Ceph distributed storage, and 100GbE network fabrics.

- *Baseline Comparisons Include:*

✓ Kubernetes HPA with CPU-based scaling
✓ KEDA with custom metrics
✓ Manual scaling based on operator expertise

➢ *Resource Utilization Efficiency*

InfraLLM achieved 57.2% reduction in CPU over-utilization events (periods where CPU usage exceeded 90% for more than 5 minutes) compared to HPA baseline. Resource allocation efficiency improved by 51.1%, measured as the ratio of actual resource consumption to allocated resources. This improvement stems from the LLM's ability to predict demand patterns and proactively allocate resources, avoiding both over-provisioning and under-provisioning.

➢ *Latency and SLO Compliance*

Average response time decreased by 48% through intelligent service placement and co-location decisions. The LLM learned that certain service pairs benefit from co-location when they share database connections, reducing network overhead. SLO violations reduced by 16×, from 0.8% of requests to 0.05%. Tail latencies (p99) improved by 11× due to proactive scaling that eliminated resource contention during traffic spikes.

➢ *Incident Reduction*

Scaling-related incidents decreased by 68% after InfraLLM deployment. The framework prevented common failure modes such as retry storms and cascade failures by recognizing early warning signs in metric patterns. Mean time to recovery (MTTR) for remaining incidents decreased by 42% as the LLM could quickly identify root causes and recommend remediation.

➢ *Cost Optimization*

Infrastructure costs decreased by 30% through more efficient resource allocation. The LLM's awareness of cost differentials between resource types (spot instances vs. on-demand, different storage tiers) enabled cost-conscious scaling decisions while maintaining performance SLAs. Total cost of ownership improved when factoring in reduced incident costs and engineering time savings.

VII. DISCUSSION

➢ *Lessons Learned*

Deployment of InfraLLM revealed several important insights. First, the quality of training data significantly impacts decision quality. Organizations with comprehensive monitoring and incident logging can achieve better results. Second, the adaptive sampling frequency in the Collection Service proved essential for balancing metric freshness with system overhead. Third, safety guardrails are critical—early deployments without sufficient constraints resulted in oscillating scaling behavior until rate limits were properly tuned.

➢ *Limitations and Future Work*

Current limitations include dependency on high-quality historical data for LLM training and the computational overhead of running inference for scaling decisions. Future work will explore federated learning approaches for multi-organization knowledge sharing, integration with chaos engineering for proactive resilience testing, and extension to edge computing environments with intermittent connectivity.

➢ *Applicability to Different Workloads*

InfraLLM demonstrates strong performance across diverse workload types including web services, batch processing, stream processing, and machine learning inference. However, stateful applications with strict consistency requirements may require additional constraints on scaling operations. The framework's flexibility allows domain-specific customization through LLM prompt engineering and safety guardrail configuration.

VIII. CONCLUSION

This paper presented InfraLLM, a novel framework for intelligent microservice auto-scaling that leverages large language models to make holistic, context-aware resource allocation decisions. By considering global cloud state, hardware heterogeneity, application priorities, and learned behavioral patterns, InfraLLM addresses fundamental limitations of existing service-centric auto-scaling approaches.

Evaluation on large-scale production Kubernetes clusters demonstrates substantial improvements across multiple dimensions: 57.2% reduction in CPU over-utilization, 51.1% improvement in resource allocation efficiency, 48% reduction in average response time, and 16× reduction in SLO violations. These results validate the potential of LLM-based orchestration for production cloud infrastructure.

The three-component architecture—Collection Service, LLM Service, and Execution Service—provides a foundation for intelligent infrastructure management that can adapt to evolving workload patterns and system characteristics. As organizations continue scaling their microservice deployments, frameworks like InfraLLM represent a promising direction for sustainable, efficient cloud resource management.

REFERENCES

[1]. Ahmad, H., Treude, C., Wagner, M., & Szabo, C. (2025). Towards resource-efficient reactive and proactive auto-scaling for microservice architectures. Journal of Systems and Software, 225, 112390. https://doi.org/10.1016/j.jss.2025.112390

[2]. Wang, Z., Zhu, S., Li, J., Jiang, W., Ramakrishnan, K. K., Zheng, Y., Yan, M., Zhang, X., & Liu, A. X. (2022). DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems. In Proceedings of the 13th Symposium on Cloud Computing (SoCC '22), pp. 16–30. ACM. https://doi.org/10.1145/3542929.3563469

[3]. Pintye, I., Kovács, J., & Lovas, R. (2024). Enhancing Machine Learning-Based Autoscaling for Cloud Resource Orchestration. Journal of Grid Computing, 22, 68. https://doi.org/10.1007/s10723-024-09783-1

[4]. Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., & Iyer, R. K. (2020). FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20). USENIX Association.

[5]. Alghamdi, A., & Shetty, S. (2024). Auto-Scaling Techniques in Cloud Computing: Issues and Research Directions. Sensors, 24(17), 5551. https://doi.org/10.3390/s24175551

[6]. Goli, A., Mahmoudi, N., Khazaei, H., & Ardakanian, O. (2021). A Holistic Machine Learning-Based Autoscaling Approach for Microservice Applications. In Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021), pp. 197-207.

[7]. Varasteh, A., Barzegar, H. R., & Cardoso, J. (2023). An Auto-Scaling Approach for Microservices in Cloud Computing Environments. Journal of Grid Computing, 22, 3. https://doi.org/10.1007/s10723-023-09713-7

[8]. Khaleq, A. A., & Ra, I. (2023). Intelligent microservices autoscaling module using reinforcement learning. Cluster Computing, 26, 4159–4176. https://doi.org/10.1007/s10586-023-03999-8

[9]. Zhang, Y., et al. (2024). Resource Sharing in Microservice Architectures. arXiv:2406.15769

[10]. Kubernetes Documentation. (2024). Horizontal Pod Autoscaling. Retrieved from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[11]. Meta Engineering. (2025). Scaling LLM Inference: Innovations in Tensor Parallelism, Context Parallelism, and Expert Parallelism. Meta Engineering Blog. https://engineering.fb.com

[12]. Chen, L., et al. (2024). FIRM: Fine-grained Interference-aware Resource Management for Microservices. arXiv:2409.14953.