Hanzo Platform: PaaS for AI-Native Application Deployment

Marcus Chen David Wei Zach Kelling
Hanzo AI Research
research@hanzo.ai

February 2026

Abstract

We present **Hanzo Platform**, a Platform-as-a-Service (PaaS) designed specifically for deploying AI-native applications—systems that incorporate LLM inference, vector databases, model serving, and GPU compute as first-class primitives. Traditional PaaS offerings (Heroku, Vercel, Railway) treat AI workloads as an afterthought, requiring manual configuration of GPU instances, model serving infrastructure, and inference APIs. Hanzo Platform introduces three innovations: (i) an *AI-aware build system* that automatically detects model dependencies, optimizes container images for inference workloads, and configures GPU scheduling, reducing deployment time for AI applications from hours to minutes; (ii) a *unified resource model* that treats LLM API access, vector database provisioning, GPU allocation, and traditional compute/storage as a single resource plane with declarative configuration; and (iii) a *cost-optimized autoscaler* that dynamically scales inference endpoints based on token throughput, request latency, and cost budgets, achieving 38% lower inference cost compared to static provisioning. We evaluate Hanzo Platform against five competing PaaS offerings on deployment time, operational cost, scaling responsiveness, and developer experience across 15 representative AI application architectures. Results demonstrate 4.7x faster deployment, 38% lower cost, and 2.1x better scaling responsiveness. Production deployment data from 14 months of operation with 2,800+ applications and 47M inference requests validates the approach.

1 Introduction

The architecture of modern applications is undergoing a fundamental shift. AI-native applications—those that incorporate LLM inference, embedding computation, retrieval-augmented generation, and model serving as core components—have distinct infrastructure requirements that existing PaaS offerings do not address.

1.1 The AI Infrastructure Gap

Traditional PaaS platforms were designed for request-response web applications with predictable resource consumption. AI applications differ in several critical ways:

1. **Heterogeneous compute**: AI workloads require GPUs for inference, CPUs for pre/post-processing, and specialized hardware (TPUs, Inferentia) for cost optimization. Traditional PaaS provides only CPU-based containers.
2. **Model lifecycle**: Models must be downloaded (often multi-GB), loaded into GPU memory, warmed up, and version-managed. Traditional PaaS treats all application artifacts as stateless code bundles.
3. **Token-based economics**: LLM inference is priced per-token rather than per-request, requiring fundamentally different cost modeling and optimization strategies.
4. **Streaming responses**: LLM applications typically stream responses via Server-Sent Events or WebSockets, requiring persistent connections that conflict with traditional HTTP load balancers.

5. **Vector storage**: RAG applications require vector databases co-located with inference for low-latency retrieval, a primitive not provided by traditional PaaS.

1.2 Contributions

1. An AI-aware build system with automatic model dependency detection and GPU-optimized container images (§3).
2. A unified resource model treating AI primitives as first-class platform resources (§4).
3. A cost-optimized autoscaler for inference workloads (§5).
4. Comprehensive evaluation against competing platforms (§8).
5. Production deployment analysis from 14 months of operation (§9).

2 Architecture

2.1 System Overview

Hanzo Platform is built on Kubernetes with custom operators for AI workload management:

1. **Control Plane**: API server, scheduler, build system, and resource manager.
2. **Compute Plane**: Kubernetes clusters with GPU and CPU node pools.
3. **Data Plane**: Managed PostgreSQL, Redis, vector databases, and object storage.
4. **Network Plane**: Ingress controllers with WebSocket support, CDN, and DNS management.
5. **Observability Plane**: Metrics, logging, tracing, and cost analytics.

2.2 Application Model

Definition 1 (Hanzo Application)**.** *An application is a tuple $A = (S, R, C, E)$ where:*

- *S: Source specification (Git repository, Docker image, or Nixpack).*
- *R: Resource requirements (compute, storage, AI services).*
- *C: Configuration (environment variables, secrets, domains).*
- *E: Scaling policy (min/max replicas, autoscaler parameters).*

Applications are defined declaratively in a `hanzo.yaml` file:

Listing 1: Example hanzo.yaml for an AI application.

```yaml
name: my-rag-app
runtime: python-3.12

services:
  web:
    build: .
    port: 8000
    gpu: a10g
    replicas: {min: 1, max: 8}
    scaling:
      metric: tokens_per_second
      target: 1000

resources:
  llm:
    provider: hanzo-gateway
    models: [claude-3.5-sonnet,
        gpt-4o]
    budget: $500/month
  vector:
    engine: pgvector
    dimensions: 1024
    storage: 50GB
  db:
    engine: postgres
    storage: 20GB
  cache:
    engine: redis
    memory: 2GB
```

3 AI-Aware Build System

3.1 Model Dependency Detection

The build system automatically scans application code for model dependencies:

Algorithm 1 Model Dependency Detection

Require: Source code S, known model registries $\mathcal{R}$

deps $\leftarrow$ ParseRequirements(S) ▷ pip, npm, cargo
models $\leftarrow \emptyset$
▷ Pattern 1: HuggingFace model references
models $\leftarrow$ models $\cup$ FindHFModels(S)
▷ Pattern 2: Model download URLs
models $\leftarrow$ models $\cup$ FindModelURLs(S)
▷ Pattern 3: Framework-specific configs
models $\leftarrow$ models $\cup$ FindFrameworkModels(S)
▷ Pattern 4: API client configurations
apis $\leftarrow$ FindLLMAPIs(S)
▷ Determine GPU requirements
gpu $\leftarrow$ EstimateGPU(models)
return (models, apis, gpu)

3.2 GPU-Optimized Container Images

For applications requiring local model inference, the build system generates optimized container images:

1. **Base image selection**: Choose from pre-built base images with CUDA, cuDNN, and framework-specific optimizations (PyTorch, TensorFlow, ONNX Runtime, vLLM).
2. **Model caching**: Models are stored in a shared cache layer, avoiding redundant downloads across deployments. Cache hit rate: 87%.
3. **Quantization**: Automatically apply GPTQ or AWQ quantization for models that exceed available GPU memory, with quality verification.
4. **Multi-stage builds**: Separate build dependencies from runtime, reducing final image size by 40–60%.

App Type	Traditional	Hanzo	Speedup
API-only (LLM gateway)	3.2 min	1.1 min	2.9x
RAG application	8.7 min	2.3 min	3.8x
Local model serving	42.1 min	6.8 min	6.2x
Full-stack AI app	15.4 min	3.2 min	4.8x
Fine-tuning pipeline	67.3 min	12.1 min	5.6x

Table 1: Build and deploy times: traditional PaaS vs. Hanzo Platform.

3.3 Incremental Deployment

Hanzo Platform supports zero-downtime deployments with AI-specific optimizations:

Algorithm 2 AI-Aware Rolling Deployment

Require: New image I', current pods $\mathcal{P}$, model warmup time T_w

▷ Phase 1: Pre-warm new pods
$\mathcal{P}' \leftarrow$ StartPods($I', |\mathcal{P}|$)
for each $p \in \mathcal{P}'$ **do**
    Wait for model loading
    Run warmup inference requests
    Verify latency $\leq$ SLA threshold
end for
▷ Phase 2: Gradual traffic shift
for $w = 0.1, 0.25, 0.5, 0.75, 1.0$ **do**
    Route w fraction of traffic to $\mathcal{P}'$
    Monitor error rate and latency for 60s
    if error rate $>$ 1% or P99 latency $> 2\times$ baseline **then**
        Rollback: route all traffic to $\mathcal{P}$
        return failure
    end if
end for
▷ Phase 3: Cleanup
Terminate old pods $\mathcal{P}$
return success

The model pre-warming phase is critical: without it, cold starts for GPU inference can take 30–120 seconds, causing unacceptable latency spikes during deployment.

4 Unified Resource Model

4.1 Resource Types

Hanzo Platform provides seven first-class resource types:

Resource	Description	Provision Time
Compute (CPU)	Container instances	< 30s
Compute (GPU)	GPU-accelerated instances	< 120s
PostgreSQL	Managed relational DB	< 60s
Redis	Managed cache/queue	< 30s
Vector DB	pgvector or Qdrant	< 60s
Object Storage	S3-compatible	< 10s
LLM Gateway	Multi-provider LLM access	Instant

Table 2: Platform resource types and provisioning times.

4.2 Declarative Resource Binding

Resources are bound to applications via environment variables and service mesh connections. The platform automatically injects connection strings, API keys, and service discovery information:

Algorithm 3 Resource Binding

Require: Application A, resource declarations R

```
1: for each resource r ∈ R do
2:     instance ← Provision(r.type, r.config)
3:     env ← GenerateEnvVars(instance)
4:     Inject env into application containers
5:     Configure network policy: A ↔ instance
6: end for
7: return bound application
```

For example, declaring a `vector` resource automatically:

1. Provisions a pgvector instance with the specified dimensions.
2. Injects `VECTOR_DB_URL` into the application environment.
3. Creates a Kubernetes NetworkPolicy allowing direct access.
4. Configures backup and monitoring.

4.3 LLM Gateway Integration

The LLM Gateway resource provides unified access to 100+ models via a single API endpoint:

- **Automatic routing**: Requests are routed to the optimal provider based on model, cost, and latency.
- **Cost budgets**: Per-application monthly cost limits with alerts at 75% and 90%.
- **Key management**: API keys are stored in encrypted vaults and injected at runtime. The application never sees raw provider keys.
- **Usage analytics**: Per-model, per-request cost tracking with real-time dashboards.

5 Cost-Optimized Autoscaler

5.1 Problem Formulation

Traditional autoscalers use CPU/memory utilization as scaling signals, which poorly correlate with AI workload performance. We formulate autoscaling as a constrained optimization problem:

$$\begin{aligned} \min_{k(t)} \quad & \int_0^T c(k(t))\,dt \\ \text{s.t.} \quad & L_{P99}(k(t), \lambda(t)) \leq L_{\max}, \\ & \text{TPS}(k(t), \lambda(t)) \geq \text{TPS}_{\min}, \\ & k_{\min} \leq k(t) \leq k_{\max}, \end{aligned} \tag{1}$$

where $k(t)$ is the number of replicas at time t, $c(k)$ is the cost function, $\lambda(t)$ is the request arrival rate, L_{P99} is the P99 latency, TPS is tokens per second throughput, and $L_{\max}$, $\text{TPS}_{\min}$ are SLA thresholds.

5.2 AI-Specific Scaling Signals

The autoscaler uses four AI-specific signals:

1. **Token throughput**: Tokens generated per second across all replicas.
2. **Queue depth**: Number of requests waiting for inference.
3. **GPU utilization**: Percentage of GPU compute capacity in use.
4. **KV cache pressure**: Fraction of KV cache memory occupied (for transformer inference).

$$\text{Scale signal} = \alpha_1 \cdot \frac{\text{TPS}}{\text{TPS}_{\text{target}}} + \alpha_2 \cdot \frac{\text{Queue}}{\text{Queue}_{\max}} + \alpha_3 \cdot \text{GPU\%} + \alpha_4 \cdot \text{KV} \tag{2}$$

where α_i are learned weights optimized to minimize cost subject to SLA constraints.

5.3 Predictive Scaling

We augment reactive scaling with a time-series forecasting model that predicts load 15 minutes ahead:

Algorithm 4 Predictive Autoscaler

Require: Current replicas k, load history $\lambda_{t-W:t}$, SLA thresholds

$\triangleright$ Reactive component
$s_{\text{react}} \leftarrow \text{ScaleSignal(current metrics)}$
$\triangleright$ Predictive component
$\hat{\lambda}_{t+\Delta} \leftarrow \text{LSTM.Predict}(\lambda_{t-W:t})$
$\hat{k}_{\text{pred}} \leftarrow \text{CapacityModel}(\hat{\lambda}_{t+\Delta}, \text{SLA})$
$\triangleright$ Combine with safety margin
$k' \leftarrow \max(k \cdot s_{\text{react}}, \hat{k}_{\text{pred}})$
$k' \leftarrow \text{clamp}(k', k_{\min}, k_{\max})$
$\triangleright$ Cost gate: prevent over-scaling
if $\text{ProjectedCost}(k') > 1.5 \times \text{Budget}$ **then**
    $k' \leftarrow \text{CostConstrained}(k', \text{Budget})$
end if
return k'

5.4 Spot Instance Integration

For GPU workloads, the autoscaler integrates with spot/preemptible instance markets:

- **Price monitoring**: Continuously monitor spot prices across regions and instance types.
- **Hybrid scaling**: Maintain a base of on-demand instances for SLA guarantees; scale burst capacity with spot instances.
- **Graceful preemption**: When spot instances are reclaimed, drain requests and redirect to on-demand capacity.

Strategy	Avg. Cost	P99 Lat.	Avail.
On-demand only	\$1.00	850ms	99.99%
Spot only	\$0.35	920ms	97.2%
Hybrid (Hanzo)	\$0.62	870ms	99.95%

Table 3: Cost-availability trade-off with spot instance integration.

5.5 Scaling Performance

Comparison of autoscaler responsiveness:

Platform	Scale-up	Scale-down	Overprov.
Vercel	12s	300s	41%
Railway	45s	180s	28%
Render	120s	600s	52%
Fly.io	8s	120s	23%
Hanzo	**6s** (CPU) **45s** (GPU)	**90s**	**14%**

Table 4: Autoscaler responsiveness. Overprov. = average over-provisioning.

6 Developer Experience

6.1 CLI Interface

The Hanzo CLI provides a streamlined deployment workflow:

Listing 2: Hanzo CLI deployment workflow.

```
# Initialize project
hanzo init

# Deploy (auto-detects framework)
hanzo deploy

# View logs
hanzo logs --service web --follow

# Scale manually
hanzo scale web --replicas 4

# View cost breakdown
hanzo cost --period 30d

# Manage secrets
hanzo secrets set API_KEY=sk-...

# Open dashboard
hanzo dashboard
```

6.2 Git-Based Deployment

Pushing to a connected Git repository triggers automatic deployment:

1. Push triggers webhook.
2. Build system detects changes, constructs optimized image.
3. Preview deployment created for pull requests.

4. Production deployment on merge to main branch.
5. Automatic rollback if health checks fail within 5 minutes.

6.3 Preview Environments

Every pull request automatically receives a preview environment with:

- Unique URL (e.g., `pr-42.my-app.hanzo.dev`).
- Isolated database snapshot (copy-on-write from production).
- Shared LLM gateway access with sandboxed cost tracking.
- Automatic teardown on PR close.

7 Security and Compliance

7.1 Secret Management

Secrets are managed via integration with Hanzo KMS (based on Infisical):

- Secrets are encrypted at rest (AES-256-GCM) and in transit (TLS 1.3).
- Access is scoped per-application and per-environment (dev/staging/prod).
- Audit logging for all secret access.
- Automatic rotation for database credentials (every 30 days).

7.2 Network Isolation

Each application runs in a dedicated Kubernetes namespace with:

- Network policies restricting inter-application communication.
- Egress filtering with allowlists for external API access.
- Service mesh (Istio) providing mTLS between services.
- WAF (Web Application Firewall) at the ingress layer.

7.3 SOC 2 Compliance

Hanzo Platform maintains SOC 2 Type II compliance through:

1. Comprehensive audit logging (CloudTrail-equivalent).
2. Automated vulnerability scanning (Trivy) on every build.
3. Annual penetration testing by third-party auditors.
4. Incident response procedures with 15-minute acknowledgment SLA.

8 Evaluation

8.1 Benchmark Applications

We evaluate Hanzo Platform using 15 representative AI application architectures:

App Type	**Components**	**GPU?**
Chat API	LLM Gateway, Redis	No
RAG App	LLM, Vector DB, Web	No
Code Assistant	LLM, Git, Sandbox	No
Image Gen API	Diffusion model, S3	Yes
Voice Agent	STT, LLM, TTS	Yes
Search Engine	Embedding, Index, LLM	Opt.
Recommendation	ML model, DB, API	Opt.
Content Moderation	Classifier, Queue	Yes
Document Processor	OCR, LLM, Storage	Opt.
Fine-tuning Service	Training, Eval, API	Yes

Table 5: Subset of 10/15 benchmark AI applications.

8.2 Deployment Time Comparison

Time from `git push` to serving first request:

App Type	**Hanzo**	**Vercel**	**Rly.**	**Fly**	**Render**
Chat API	1.1m	0.8m	2.4m	1.9m	3.2m
RAG App	2.3m	4.1m	5.7m	4.8m	7.1m
Image Gen	6.8m	N/A	28m	15m	35m
Voice Agent	5.2m	N/A	22m	12m	29m
Fine-tuning	12.1m	N/A	N/A	41m	N/A
Average	**5.5m**	—	14.5m	14.9m	18.6m

Table 6: Deployment times. N/A = platform does not support GPU workloads.

8.3 Cost Comparison

Monthly cost for running a medium-traffic RAG application (1000 req/hour):

Component	Hanzo	AWS	Fly	Rly.
Compute	$47	$89	$62	$58
Database	$15	$42	$20	$25
Vector DB	$12	$67	$35	N/A
LLM API	$180	$180	$180	$180
Storage	$5	$12	$8	$7
Total	**$259**	$390	$305	$270+

Table 7: Monthly cost comparison for a medium-traffic RAG application.

8.4 Developer Experience Survey

We surveyed 120 developers who deployed AI applications on multiple platforms:

Dimension	Hanzo	Vercel	Fly	AWS
Setup ease	4.6/5	4.7/5	3.8/5	2.4/5
AI support	4.8/5	2.9/5	3.2/5	3.8/5
Cost clarity	4.5/5	3.4/5	3.9/5	2.1/5
Debugging	4.3/5	3.8/5	3.5/5	3.2/5
Scaling	4.4/5	4.2/5	4.0/5	4.5/5
Overall	**4.5/5**	3.8/5	3.7/5	3.2/5

Table 8: Developer experience survey results (120 respondents).

9 Production Deployment

9.1 Infrastructure

Hanzo Platform runs on two DOKS (DigitalOcean Kubernetes Service) clusters:

- **hanzo-k8s** (24.199.76.156): Control plane, databases, core services.
- **GPU pool**: 8x A100 GPU nodes for inference workloads, expandable to 32.
- **Edge nodes**: 12 PoPs globally for CDN and edge compute.

9.2 Usage Statistics (14 Months)

Metric	Value
Applications deployed	2,847
Total deployments	41,293
Active organizations	891
Inference requests served	47.2M
Tokens processed	18.7B
Total LLM cost facilitated	$2.1M
Avg. deployment time	3.4 min
Platform uptime	99.97%
Avg. cost savings vs. AWS	34%

Table 9: Production statistics (Dec 2024 – Feb 2026).

9.3 Application Distribution

App Category	Count	Avg. Monthly Cost
Chat/conversational	34.2%	$127
RAG/search	21.7%	$234
API services	18.3%	$89
Model serving	12.1%	$412
Data pipelines	8.4%	$187
Other	5.3%	$156

Table 10: Application category distribution in production.

10 Related Work

10.1 General-Purpose PaaS

Heroku [6] pioneered the PaaS model with git-push deployment. Vercel [16] specializes in frontend and serverless deployments. Railway [11] provides database-friendly PaaS. Fly.io [5] offers edge computing with container support. Render [12] provides managed infrastructure. None of these treat AI workloads as first-class citizens.

10.2 AI-Specific Infrastructure

Modal [10] provides serverless GPU compute for AI workloads. Replicate [13] offers model hosting with API generation. Baseten [1] specializes in ML model deployment. Together AI [15] provides inference API with custom model support. Hanzo Platform differs by providing a full application platform (not just model serving) with integrated databases, secrets, and frontend hosting.

10.3 Kubernetes-Based PaaS

Dokku [3] provides Heroku-like deployment on single servers. KubeSphere [8] offers Kubernetes-based application management. Dokploy [4] provides self-hosted PaaS on Kubernetes. Hanzo Platform builds on Dokploy's foundation with AI-specific extensions for GPU management, model caching, and inference autoscaling.

10.4 ML Deployment Platforms

MLflow [9] provides ML lifecycle management. Seldon Core [14] offers ML model serving on Kubernetes. KServe [7] provides serverless inference. BentoML [2] packages ML models for deployment. These tools focus on model serving; Hanzo Platform encompasses the full application stack.

11 Discussion

11.1 Vendor Lock-in Mitigation

Hanzo Platform uses standard Kubernetes primitives and OCI containers, enabling applications to be extracted and deployed on any Kubernetes cluster. The `hanzo eject` command generates standard Kubernetes manifests, Dockerfiles, and Helm charts for migration.

11.2 Limitations

1. **GPU availability**: GPU capacity is finite; burst scaling may be delayed during high-demand periods.

2. **Cold starts**: GPU inference cold starts (30–120s for model loading) remain a challenge despite pre-warming.

3. **Regional coverage**: Currently deployed in 3 regions (US East, US West, EU West); Asia-Pacific coverage is planned.

4. **Custom hardware**: Only NVIDIA GPUs are currently supported; AMD and custom accelerators are on the roadmap.

11.3 Future Work

- **Edge inference**: Deploy small models at CDN edge nodes for ultra-low-latency inference.

- **Multi-cloud**: Support AWS, GCP, and Azure as compute backends alongside DigitalOcean.

- **Fine-tuning as a service**: Managed fine-tuning with automatic deployment of resulting models.

- **AI-assisted operations**: Use LLMs to diagnose deployment failures and suggest fixes.

12 Conclusion

We have presented Hanzo Platform, a PaaS designed for AI-native applications that treats LLM inference, vector databases, GPU compute, and model serving as first-class platform primitives. The AI-aware build system reduces deployment time by 4.7x, the unified resource model simplifies infrastructure management, and the cost-optimized autoscaler reduces inference costs by 38%. Evaluation against five competing platforms demonstrates superior performance across deployment time, cost, scaling, and developer experience. Production deployment with 2,800+ applications and 47M inference requests over 14 months validates the practical viability of the approach. Hanzo Platform is available at `platform.hanzo.ai`.

References

[1] Baseten. Baseten: The fastest way to deploy ML models. *Baseten Documentation*, 2022.

[2] BentoML. BentoML: The unified model serving framework. *GitHub Repository*, 2022.

[3] Dokku. Dokku: The smallest PaaS implementation you've ever seen. *GitHub Repository*, 2013.

[4] Dokploy. Dokploy: Self-hosted platform as a service. *GitHub Repository*, 2024.

[5] Fly.io. Fly.io: Run your full stack apps close to your users. *Fly.io Documentation*, 2020.

[6] Heroku. Heroku: Cloud application platform. *Heroku Documentation*, 2007.

[7] KServe. KServe: Highly scalable and standards based model inference platform. *KServe Documentation*, 2021.

[8] KubeSphere. KubeSphere: The container platform tailored for Kubernetes. *KubeSphere Documentation*, 2020.

[9] M. Zaharia, A. Chen, A. Davidson, et al. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Engineering Bulletin*, 41(4):39–45, 2018.

[10] Modal. Modal: End the struggle with cloud infrastructure. *Modal Documentation*, 2023.

[11] Railway. Railway: Instant deployments, effortless scale. *Railway Documentation*, 2021.

[12] Render. Render: Cloud application hosting. *Render Documentation*, 2019.

[13] Replicate. Replicate: Run and fine-tune open-source models. *Replicate Documentation*, 2022.

[14] Seldon. Seldon core: An open source platform to deploy machine learning models. *Seldon Documentation*, 2019.

[15] Together AI. Together: Fast inference and fine-tuning. *Together Documentation*, 2023.

[16] Vercel. Vercel: Develop. Preview. Ship. *Vercel Documentation*, 2020.

[17] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and Kubernetes. *ACM Queue*, 14(1):70–93, 2016.

[18] B. Hindman, A. Konwinski, M. Zaharia, et al. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[19] W. Kwon, Z. Li, S. Zhuang, et al. Efficient memory management for large language model serving with PagedAttention. In *SOSP*, 2023.

[20] L. Zheng, L. Yin, Z. Xie, et al. SGLang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2024.

[21] Y. Sheng, L. Zheng, B. Yuan, et al. FlexGen: High-throughput generative inference of large language models with a single GPU. In *ICML*, 2023.

[22] R. Pope, S. Douglas, A. Chowdhery, et al. Efficiently scaling transformer inference. In *MLSys*, 2023.

[23] R. Y. Aminabadi, S. Rajbhandari, M. Zhang, et al. DeepSpeed inference: Enabling efficient inference of transformer models at unprecedented scale. In *SC22*, 2022.

[24] P. Patel, E. Choukse, C. Zhang, et al. Splitwise: Efficient generative LLM inference using phase splitting. In *ISCA*, 2024.

[25] A. Agrawal, N. Kedia, A. Panwar, et al. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *OSDI*, 2024.