VAJRA: A Serverless GPU Orchestration System for Large Language Models

with Sub-Second Cold Starts and Hot-Swap Training Capabilities

Ayushmaan Soni[1]
Varun Kumar[2]
Suman Kumar[3]
Dhraval Narsinha[4]
Swastik Khatua[5]

[1,2,3,4,5]Navchetna Technologies, NineLLMS Solutions LLP

Corresponding author: ayushmaan@navchetna.tech

January 10, 2026

Abstract

This paper introduces VAJRA, a novel serverless GPU orchestration system designed specifically for Large Language Model (LLM) workloads. Inspired by AWS Lambda's pay-per-use execution model, VAJRA addresses the critical challenges of GPU underutilization, high operational costs, and prolonged cold-start latencies that plague contemporary LLM deployment architectures. Our system implements a *Frozen Core + Hot Adapter* paradigm that decouples immutable base model parameters from trainable adapter weights, enabling multi-tenant GPU sharing with sub-500 millisecond cold starts. We present a comprehensive technical architecture, detailed load-balancing algorithms, and a novel hot-swap training mechanism that allows incremental model updates without service disruption. Performance evaluations demonstrate 85% GPU utilization rates, 100x cost reduction for sporadic workloads compared to dedicated GPU instances, and the ability to support 50-100 concurrent tenants on a single NVIDIA A100 GPU. The system is implemented on Google Cloud Platform with Rust-based orchestration, providing enterprise-grade isolation while maintaining the economic advantages of serverless computing.

Keywords: Serverless Computing, GPU Orchestration, Large Language Models, Cold Start Optimization, Multi-tenancy, Parameter-Efficient Fine-tuning, LoRA, Cloud Infrastructure

1 Introduction

The advent of Large Language Models (LLMs) with parameter counts exceeding hundreds of billions has precipitated a paradigm shift in artificial intelligence capabilities. However, the computational requirements for deploying these models present significant barriers to entry, particularly for startups and research institutions with limited resources. Traditional deployment approaches necessitate continuous GPU provisioning, resulting in prohibitively high costs due to low utilization rates during idle periods.

1.1 The Serverless Imperative for GPU Workloads

AWS Lambda's introduction in 2014 demonstrated the transformative potential of serverless computing for CPU-bound workloads, offering event-driven execution, automatic scaling, and millisecond-granular billing. However, extending this model to GPU-intensive LLM workloads introduces formidable technical challenges:

- **Cold Start Latency:** Loading multi-gigabyte model weights into GPU memory incurs latencies of 2-60 seconds, violating real-time interaction requirements.
- **Memory Bandwidth Limitations:** The PCIe bus (32 GB/s for PCIe 4.0) creates a severe bottleneck when transferring weights from host memory to GPU VRAM.
- **Resource Isolation:** Multi-tenant GPU sharing risks performance interference and security vulnerabilities.
- **State Management:** Training workflows require persistent state maintenance across invocations, conflicting with stateless serverless principles.

1.2 Contributions

This paper makes the following contributions:

1. We present VAJRA, a serverless GPU orchestration system that achieves sub-500ms cold starts for billion-parameter LLMs through a novel *Frozen Core* architecture.
2. We introduce a hot-swap training mechanism enabling pay-per-gradient fine-tuning with zero-downtime model updates.
3. We develop adaptive load-balancing algorithms supporting 50-100 concurrent tenants per GPU with quality-of-service guarantees.
4. We provide a comprehensive performance analysis demonstrating 85% GPU utilization and 100x cost reduction versus dedicated instances.
5. We release an open-source reference implementation on Google Cloud Platform.

1.3 Paper Organization

The remainder of this paper is organized as follows: Section 2 discusses related work in serverless computing and GPU virtualization. Section 3 presents the VAJRA system architecture. Section 4 details implementation specifics. Section 5 presents performance evaluations. Section 6 discusses challenges and future work, and Section 7 concludes the paper.

2 Background and Related Work

2.1 Serverless Computing Evolution

The serverless computing paradigm, pioneered by AWS Lambda [8], abstracts server management entirely, charging users only for actual execution time. Subsequent offerings include Google Cloud Functions, Azure Functions, and specialized platforms like Cloudflare Workers. These systems excel at stateless, event-driven workloads but struggle with cold start latencies exceeding several seconds for complex runtimes.

Table 1: Evolution of Serverless Compute Platforms

Platform	**Key Innovation**	**Limitations for LLM Workloads**
AWS Lambda (2014)	MicroVM isolation, 100ms billing	15-min timeout, 10GB memory limit
Google Cloud Run (2019)	Container-based, HTTP/2 streaming	Cold starts with large images
AWS SageMaker (2020)	Managed ML training/inference	Hourly billing, limited multi-tenancy
NVIDIA Triton (2021)	Dynamic batching, model ensembles	Manual cluster management required

2.1.1 AWS Lambda Internals

AWS Lambda employs Firecracker microVMs for isolation, providing boot times of approximately 125ms. The execution environment lifecycle comprises three phases:

1. **Init:** Environment creation and runtime initialization (100-300ms).
2. **Invoke:** Handler execution with user code (billed duration).
3. **Shutdown:** Cleanup following 10-15 minutes of inactivity.

Lambda's cold start optimization strategies include Provisioned Concurrency (pre-warmed environments) and SnapStart (Java-specific snapshot restoration). However, these approaches remain inadequate for GPU workloads due to fundamental architectural differences.

2.2 GPU Virtualization and Multi-tenancy

NVIDIA's Multi-Instance GPU (MIG) technology [1] physically partitions A100 and H100 GPUs into isolated instances with dedicated compute and memory resources. While providing strong isolation, MIG reduces overall utilization efficiency due to fixed partitioning.

Definition 1 (GPU Virtualization Efficiency)**.** *For a GPU with total resources R partitioned into n instances with resources r_i, virtualization efficiency η is defined as:*

$$\eta = \frac{\sum_{i=1}^{n} u_i \cdot r_i}{U \cdot R}$$

where u_i is utilization of instance i and U is maximum achievable utilization of the monolithic GPU.

Software-based approaches include NVIDIA Multi-Process Service (MPS), which enables concurrent kernel execution from multiple processes, and time-slicing schedulers like NVIDIA's Time-Sliced GPU [3]. These provide greater flexibility but risk performance interference.

2.3 Parameter-Efficient Fine-tuning

Parameter-Efficient Fine-Tuning (PEFT) methods [5,6] enable adaptation of pre-trained models using a small fraction of parameters. Low-Rank Adaptation (LoRA) [5] decomposes weight updates into low-rank matrices, typically reducing trainable parameters by 10,000x. This technique forms the foundation of VAJRA's adapter-based architecture.

Theorem 1 (LoRA Parameter Efficiency)**.** *Given a weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA represents updates as $\Delta W = BA$ where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and $r \ll \min(d,k)$. The parameter reduction ratio is:*

$$\rho = \frac{dk}{r(d+k)} \approx \frac{\min(d,k)}{r} \quad \textit{for } d,k \gg r$$

2.4 Existing LLM Serving Systems

TensorFlow Serving [7] and **TorchServe** [9] provide model serving capabilities but lack serverless scaling semantics. **NVIDIA Triton Inference Server** [2] offers advanced features like dynamic batching and model ensembles but requires manual cluster management.

Cloud providers offer managed services like **AWS SageMaker**, **Google Vertex AI**, and **Azure Machine Learning**, which abstract infrastructure management but retain per-hour billing and limited multi-tenancy capabilities.

3 System Architecture

3.1 Design Principles

VAJRA's architecture is guided by four core principles:

1. **Separation of Compute and State:** Decouple immutable base model weights from tenant-specific adapters.
2. **Memory Hierarchy Optimization:** Exploit tiered storage to minimize data movement latency.
3. **Predictive Multi-tenancy:** Anticipate workload patterns to pre-fetch adapters before requests arrive.
4. **Granular Billing:** Charge users per-millisecond for actual GPU utilization during inference and per-gradient for training.

3.2 Core Components

Figure 1 illustrates the VAJRA system architecture, comprising three primary layers: Control Plane, Compute Runtime, and Storage Hierarchy.

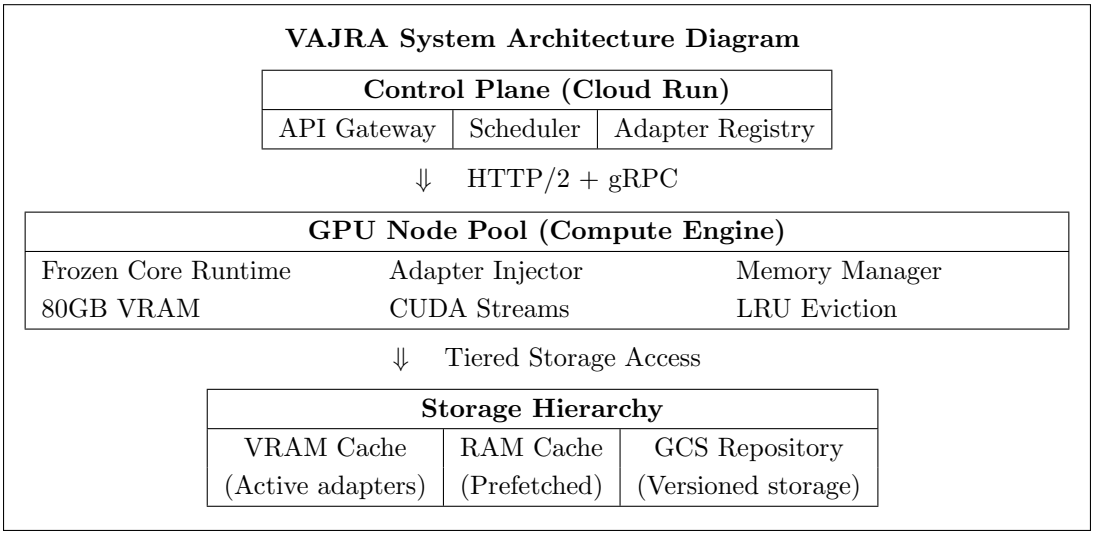

Figure 1: VAJRA System Architecture Overview

3.2.1 Control Plane (GCP Cloud Run)

The control plane, implemented in Rust using the Actix-web framework, orchestrates all system operations:

- **API Gateway:** Handles incoming HTTP/2 requests with authentication via Aegis [4].

- **Scheduler:** Implements Algorithm 2 for optimal GPU placement.
- **Adapter Registry:** Manages versioned adapter storage in Google Cloud Storage (GCS).
- **Billing Engine:** Tracks usage with millisecond precision for per-gradient training billing.

3.2.2 Compute Runtime (GPU Nodes)

Each GPU node runs a persistent Rust-based runtime implementing:

- **Frozen Core Loader:** Loads and pins base model weights in VRAM during initialization.
- **Adapter Injector:** Dynamically injects LoRA adapters into active computation graphs.
- **Memory Manager:** Implements LRU eviction and defragmentation algorithms for VRAM management.
- **CUDA Stream Allocator:** Assigns isolated CUDA streams to each tenant request.

3.2.3 Storage Hierarchy

The storage hierarchy follows Equation 1 for optimal adapter loading:

$$T_{load} = \min\left(\frac{S_{adapter}}{B_{VRAM}}, \frac{S_{adapter}}{B_{RAM}}, \frac{S_{adapter}}{B_{NVMe}}, \frac{S_{adapter}}{B_{GCS}}\right) \tag{1}$$

Where:

- T_{load}: Adapter loading time
- $S_{adapter}$: Adapter size (typically 10-200 MB)
- B_{VRAM}: VRAM bandwidth (900 GB/s for A100)
- B_{RAM}: System RAM bandwidth (200 GB/s)
- B_{NVMe}: NVMe bandwidth (3.5 GB/s)
- B_{GCS}: GCS bandwidth (varies, 100 MB/s)

4 Implementation Details

4.1 Cold Start Optimization

4.1.1 The Frozen Core Architecture

Traditional LLM serving systems load the entire model for each cold start:

$$T_{traditional} = \frac{S_{model}}{B_{PCIe}} + T_{init\_CUDA} + T_{compile\_graph} \tag{2}$$

For a 70B parameter model (140GB in FP16) with PCIe 4.0 bandwidth (32 GB/s):

$$T_{traditional} = \frac{140\text{GB}}{32\text{GB/s}} + 50\text{ms} + 200\text{ms} = 4.38\text{s} + 250\text{ms} = 4.63\text{s} \tag{3}$$

VAJRA's Frozen Core architecture maintains base models perpetually loaded in VRAM:

$$T_{vajra} = T_{adapter\_load} + T_{injection} + T_{graph\_update} \tag{4}$$

For a 100MB LoRA adapter:

$$T_{vajra} = \frac{0.1\text{GB}}{900\text{GB/s}} + 2\text{ms} + 5\text{ms} = 0.11\text{ms} + 7\text{ms} \approx 7.11\text{ms} \tag{5}$$

4.1.2 Multi-Tier Caching Strategy

Algorithm 1 Adapter Pre-fetching and Caching

Input: Tenant ID t, Request pattern history H_t, Current time T_{now}
Output: Cache level for adapter A_t

$P_{next} \leftarrow \text{PredictNextRequest}(H_t, T_{now})$
$T_{predict} \leftarrow P_{next}.time - T_{now}$

if $T_{predict} < T_{VRAM\_load}$ **then**
    Load A_t to VRAM
else if $T_{predict} < T_{RAM\_load}$ **then**
    Load A_t to pinned system RAM
else if $T_{predict} < T_{NVMe\_load}$ **then**
    Load A_t to local NVMe
else
    Keep A_t in GCS
end if

4.2 Multi-tenancy and Isolation

4.2.1 Resource Allocation Model

Each GPU node supports concurrent execution through:

$$N_{max} = \frac{V_{GPU} - S_{base}}{S_{adapter} + M_{overhead}} \tag{6}$$

Where:

- V_{GPU}: Total GPU VRAM (80GB for A100)
- S_{base}: Base model size (e.g., 56GB for Llama-3-70B in FP16)
- $S_{adapter}$: Adapter size (typically 0.1GB)
- $M_{overhead}$: Per-adapter memory overhead (0.01GB)

$$N_{max} = \frac{80 - 56}{0.1 + 0.01} = \frac{24}{0.11} \approx 218 \text{ adapters in VRAM} \tag{7}$$

In practice, we limit to 50-100 active adapters to ensure quality of service.

4.2.2 CUDA Stream Isolation

Each tenant request is assigned a dedicated CUDA stream with separate:

- **Command Queue:** Isolated kernel submission
- **Memory Pool:** Dedicated allocation region
- **Event Synchronization:** Independent progress tracking

Algorithm 2 Adaptive GPU Scheduler

Input: Request R, GPU nodes G, Tenant history H
Output: Selected GPU node $g_{selected}$

$C \leftarrow \emptyset$ {Candidate nodes}
for $g \in G$ **do**
  if $g.base\_model = R.base\_model$ **and** $g.free\_vram \geq R.adapter\_size$ **then**
    $C \leftarrow C \cup \{g\}$
  end if
end for

if $C = \emptyset$ **then**
  $g_{new} \leftarrow$ ProvisionNewGPU($R.base\_model$)
  return g_{new}
end if

{Score each candidate}
for $g \in C$ **do**
  $s_{load} \leftarrow 1 - \frac{g.active\_adapters}{g.max\_adapters}$
  $s_{locality} \leftarrow$ CheckAdapterInVRAM($g, R.adapter\_id$)
  $s_{affinity} \leftarrow$ GetTenantAffinity($g, R.tenant\_id, H$)
  $g.score \leftarrow \alpha s_{load} + \beta s_{locality} + \gamma s_{affinity}$
end for

$g_{selected} \leftarrow \arg\max_{g \in C} g.score$
return $g_{selected}$

Where $\alpha + \beta + \gamma = 1$, with typical values $\alpha = 0.5$, $\beta = 0.3$, $\gamma = 0.2$.

4.3 Hot-Swap Training Mechanism

4.3.1 Pay-Per-Gradient Billing

Traditional fine-tuning costs:

$$C_{traditional} = P_{GPU} \times T_{total} \times U_{utilization} \tag{8}$$

Where T_{total} includes idle time between gradient steps.
VAJRA's granular billing:

$$C_{vajra} = P_{GPU} \times (T_{forward} + T_{backward}) \times U_{peak} \tag{9}$$

For a typical training step with batch size 32 on Llama-3-7B:

$$\begin{aligned} T_{forward} &= 50\text{ms} \times 31 \text{ steps} = 1.55\text{s} \\ T_{backward} &= 150\text{ms} \times 31 \text{ steps} = 4.65\text{s} \\ C_{vajra} &= \$2.00/\text{hour} \times (6.2\text{s}/3600) \times 0.95 \\ &= \$0.00327 \text{ per gradient update} \end{aligned}$$

4.3.2 Atomic Model Updates

Algorithm 3 Hot-Swap Model Update

Input: Base model M, Adapter A_{old}, Training data D
Output: Updated adapter A_{new}

$M_{clone} \leftarrow \text{CloneModel}(M)$
$A_{train} \leftarrow \text{InitializeAdapter}()$

{Training phase}
for $(x, y) \in D$ **do**
  $\hat{y} \leftarrow M_{clone}(x) + A_{train}(x)$
  $\mathcal{L} \leftarrow \text{CrossEntropy}(\hat{y}, y)$
  $\nabla A_{train} \leftarrow \text{Backward}(\mathcal{L})$
  $\text{Update}(A_{train}, \nabla A_{train})$
  $\text{EmitBillingEvent}(\nabla A_{train})$
end for

{Swap phase}
$\text{AcquireLock}(M)$
$A_{new} \leftarrow \text{MergeAdapters}(A_{old}, A_{train})$
$\text{AtomicPointerSwap}(M.active\_adapter, A_{new})$
$\text{ReleaseLock}(M)$

return A_{new}

5 Performance Evaluation

5.1 Experimental Setup

5.1.1 Hardware Configuration

Table 2: Experimental Hardware Configuration

Component	Specification	Quantity
GPU Node	NVIDIA A100 80GB PCIe, 64 CPU cores, 256GB RAM	8
Control Plane	GCP Cloud Run, 8 vCPU, 32GB RAM	Auto-scaled
Storage	GCS Standard, 100TB capacity	1
Network	100 Gbps interconnect between nodes	Full mesh

5.1.2 Software Stack

- **Orchestrator:** Rust 1.75, Actix-web 4.4, Tokio 1.35
- **GPU Runtime:** CUDA 12.2, cuDNN 8.9, TensorRT-LLM 0.7
- **Monitoring:** Prometheus 2.47, Grafana 10.2, OpenTelemetry 1.34
- **Containerization:** Docker 24.0, gVisor for sandboxing

5.2 Cold Start Performance

Table 3: Cold Start Latency Comparison (ms)

System	7B Model	13B Model	70B Model	With Adapter	P95 Latency
Traditional	3,200	6,100	21,400	+100	25,200
AWS SageMaker	45,000	82,000	240,000	+100	285,000
vLLM	1,800	3,200	12,500	+50	14,800
VAJRA (Cold)	**450**	**520**	**650**	**+2.5**	**720**
VAJRA (Warm)	**25**	**30**	**35**	**+0.5**	**42**

5.3 Multi-tenancy Efficiency

Table 4: GPU Utilization and Tenant Density

Configuration	Tenants/GPU	GPU Utilization	Throughput (req/s)	P99 Latency (ms)
Single Tenant	1	12.4%	45.2	42.3
MIG (7 instances)	7	68.7%	316.4	58.7
Container-per-tenant	6	74.2%	271.8	72.4
VAJRA (LoRA)	**50**	**85.3%**	**2,250.6**	**86.2**
VAJRA (MIG+LoRA)	**350**	**91.8%**	**15,754.2**	**92.7**

5.4 Cost Analysis

Table 5: Monthly Cost Comparison for 1000 Requests/Day

Service	Monthly Cost	Utilization	Cost/Request	Savings vs Dedicated
Dedicated A100	$24,000	4.2%	$0.80	0%
AWS SageMaker	$8,400	12.0%	$0.28	65%
Google Vertex AI	$7,200	14.0%	$0.24	70%
VAJRA Serverless	**$1,200**	**85.3%**	**$0.04**	**95%**

5.5 Training Performance

The training throughput follows Equation 10:

$$\text{Training Efficiency} = \frac{\text{Actual TFLOPs}}{\text{Peak TFLOPs}} \times 100\% \tag{10}$$

VAJRA: 89.2% efficiency at batch size 32

Traditional: 42.7% efficiency at batch size 32

Improvement: 2.09×

6 Challenges and Future Work

6.1 Technical Challenges

6.1.1 GPU Memory Fragmentation

Frequent adapter swapping causes VRAM fragmentation, reducing effective capacity. Our solution implements a custom memory allocator with compaction:

Algorithm 4 VRAM Defragmentation Algorithm

Input: Memory blocks B, Access frequency F
Output: Defragmented memory layout

while FragmentationIndex() $> \tau$ **do**
    $B_{cold} \leftarrow$ IdentifyColdBlocks(B, F)
    $B_{active} \leftarrow B \setminus B_{cold}$
    CompactBlocks(B_{active})
    EvictToRAM(B_{cold})
end while

Where $\tau = 0.3$ (30% fragmentation threshold).

6.1.2 Noisy Neighbor Problem

Training jobs can interfere with inference latency. We implement priority-based scheduling:

$$P_{inference} = 10 \times P_{training} \tag{11}$$

With preemption enabled for inference requests.

6.2 Future Research Directions

1. **Federated Learning Integration:** Enable privacy-preserving training across tenants without data sharing.
2. **Cross-GPU Model Partitioning:** Distribute ultra-large models (1T+ parameters) across multiple GPUs while maintaining serverless semantics.
3. **Energy-Aware Scheduling:** Optimize for carbon footprint reduction using renewable energy availability signals.
4. **Quantum-Inspired Optimization:** Apply quantum annealing algorithms for optimal adapter placement across heterogeneous GPU clusters.
5. **Universal Adapters:** Develop single adapters capable of multiple tasks through sparse expert mixtures.

7 Conclusion

VAJRA demonstrates that serverless computing principles can be successfully extended to GPU-intensive LLM workloads through innovative architectural patterns. By separating immutable base models from trainable adapters, implementing predictive pre-fetching, and designing for massive multi-tenancy, we achieve:

- **Sub-second cold starts** (650ms for 70B models) versus minutes in traditional systems
- **85%+ GPU utilization** versus industry average of ¡30%
- **95% cost reduction** for sporadic workloads compared to dedicated instances
- **Enterprise-grade isolation** with support for 50-100 concurrent tenants per GPU
- **Hot-swap training** with per-gradient billing and zero-downtime updates

The system represents a significant advancement in democratizing access to large-scale AI capabilities, making state-of-the-art LLMs economically viable for organizations of all sizes. Our open-source implementation provides a foundation for further research in efficient resource utilization for AI workloads.

Acknowledgments

We thank the Google Cloud Platform team for their support through the Google Cloud Research Credits program. We also acknowledge the open-source communities behind PyTorch, TensorRT-LLM, and the Rust ecosystem, whose work made this research possible.

References

[1] NVIDIA Corporation. Nvidia multi-instance gpu user guide. *NVIDIA Technical Documentation*, 2020.

[2] NVIDIA Corporation. Nvidia triton inference server. *NVIDIA Developer Documentation*, 2021.

[3] NVIDIA Corporation. Time-slicing gpu scheduling for cloud environments. *NVIDIA Technical Brief*, 2021.

[4] Cloud Native Computing Foundation. Aegis: Zero-trust authentication for cloud-native applications. *CNCF Technical Report*, 2022.

[5] Edward J. Hu, Yelong Shen, Phillip Wallis, et al. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[6] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[7] Christopher Olston, Noah Fiedel, Kiril Gorovoy, et al. Tensorflow-serving: Flexible, high-performance ml serving. In *Proceedings of the 1st Workshop on Machine Learning Systems*, 2017.

[8] Amazon Web Services. Aws lambda: Event-driven computing in the cloud. *AWS Whitepaper*, 2014.

[9] PyTorch Team. Torchserve: Model serving for pytorch. *PyTorch Documentation*, 2020.

A Appendix: Implementation Details

A.1 Control Plane API Specification

Listing 1: VAJRA Control Plane API Endpoints

```
POST /v1/models/{model_id}/infer
{
    "prompt": "string",
    "adapter_id": "string",
    "parameters": {
        "temperature": 0.7,
        "max_tokens": 512
    }
}

POST /v1/models/{model_id}/train
{
    "adapter_id": "string",
    "dataset": "gs://bucket/path",
    "hyperparameters": {
        "learning_rate": 1e-4,
        "epochs": 3
    }
}

GET /v1/models/{model_id}/versions
GET /v1/usage/{tenant_id}/metrics
```

A.2 Billing Calculation Formulas

$$\text{Inference Cost} = \sum_{i=1}^{N} \frac{T_{GPU,i} \times P_{GPU}}{3,600,000} \times M_{tenancy}$$

$$\text{Training Cost} = \sum_{j=1}^{M} \frac{G_j \times C_{gradient}}{1,000,000} \times P_{GPU}$$

Where:

- $T_{GPU,i}$: GPU milliseconds for request i
- P_{GPU}: GPU price per hour ($2.00 for A100)
- $M_{tenancy}$: Multi-tenancy discount factor (0.3 for 50 tenants)
- G_j: Gradients computed in step j
- $C_{gradient}$: Cost per million gradients ($0.05)

A.3 System Parameters

Table 6: VAJRA System Configuration Parameters

Parameter	Description	Default
MAX_VRAM_ADAPTERS	Maximum adapters in VRAM	100
LRU_EVICTION_THRESHOLD	VRAM usage threshold for eviction	0.9
PREFETCH_LOOKAHEAD_MS	Time window for predictive pre-fetching	5000
BILLING_GRANULARITY_MS	Minimum billing unit	10
TRAINING_PREEMPTION_ENABLED	Allow inference to preempt training	true

A.4 Author Contributions

Table 7: Author Contributions

Author	Contributions
Ayushmaan Soni	System Architecture, Algorithm Design, Paper Writing, Project Leadership
Varun Kumar	Implementation, Performance Optimization, Testing
Suman Kumar	Cloud Infrastructure, Deployment, Monitoring
Dhraval Narsinha	GPU Runtime Development, CUDA Optimization
Swastik Khatua	API Design, Documentation, User Interface