Cloud–Native LLM Inference: From Local vLLM Deployment to Managed Cloud Platforms

Sun Yixuan[1]

NUIST Waterford Institution
`20109698@mail.wit.ie`

Abstract With the rapid proliferation of large language models (LLMs) in various generative applications, the demand for low-latency, high-throughput, and elastically scalable inference services is on the rise. This study focuses on the engineering trade-offs between two representative solutions—*local single-machine deployment* and *cloud-hosted inference*. Using the small instruction-tuned model `Qwen2.5--0.5B--Instruct` as a case, we first conduct empirical tests on a consumer-grade GPU (RTX 3060 12 GB) powered by vLLM, evaluating how different prompt lengths and concurrency levels affect latency, throughput, and memory usage. We then analyze and compare the pricing models, performance metrics, and auto-scaling capabilities of three major platforms—AWS SageMaker, Google Vertex AI, and Alibaba PAI—drawing on public literature and official documentation. Experimental results show: (1) On local hardware, vLLM sustains a stable throughput of 350–430 tokens/s while using about 1.5 GB of VRAM, making it suitable for edge or educational scenarios; (2) On T4 GPU instances, the three hosted platforms exhibit similar latency and cost, yet differ in framework support, observability, and ecosystem integration. Based on these findings, we provide deployment recommendations for different use cases and discuss future work that combines vLLM with Kubernetes-based auto-scaling.

Keywords: LLM inference · vLLM · cloud native · managed ML services · performance evaluation · auto-scaling

1 Introduction

1.1 Motivation

Large language models (LLMs) have become the core of generative AI applications, yet their inference stage often entails significant compute and memory requirements. Service providers aim to reduce costs while meeting service-level objectives (SLOs) and handling bursty traffic through resource elasticity. *Local single-machine inference* offers simplicity and privacy control, whereas *managed cloud platforms* deliver auto-scaling, built-in monitoring, and production-grade reliability. Making rational choices between the two remains an active topic in cloud computing and systems research.

1.2 Problem Statement

This paper investigates two key questions:

1. **Q1:** When deploying a small-scale LLM (hundreds of millions of parameters) on a single consumer-grade GPU, what are the latency, throughput, and resource limits of vLLM?
2. **Q2:** How do AWS SageMaker, GCP Vertex AI, and Alibaba PAI differ in deployment complexity, performance, cost, and auto-scaling capabilities?

1.3 Contributions

- We provide a reproducible local inference testing workflow for vLLM with a small model and share detailed results.
- We compile a comparison table of the three cloud platforms covering pricing, performance, and platform features.
- We summarize deployment recommendations for different scenarios and discuss the feasibility of building a self-hosted cloud-native inference stack using Kubernetes and vLLM.

1.4 Paper Organization

The remainder of the paper is organized as follows: Section 2 reviews background on LLM inference and cloud inference services; Section 3 surveys related work; Section 4 presents the architectures of local and hosted inference systems; Section 5 details the experiments and results; Section 6 discusses limitations and practical implications; Section 7 concludes and outlines future work.

2 Background

2.1 LLM Inference Workflow and Metrics

LLM inference is typically divided into *Prefill* (a single forward pass over all input tokens) and *Decode* (iteratively generating new tokens). Key performance metrics include: **time-to-first-token (TTFT)**, **end-to-end latency**, **throughput** (tokens/s or req/s), and **GPU memory**. Prefill cost grows linearly with prompt length, while the Decode phase is more affected by KV-Cache access and concurrency scheduling.

2.2 Model Serving in Cloud Computing

Online inference services usually adopt a microservice architecture whose core components comprise an API Gateway, a request queue/batcher, a model server, and monitoring/alerting. Managed cloud platforms provide deployment, configuration, and auto-scaling via a control plane, whereas the data plane performs actual inference.

3 Related Work

As the parameter size of LLMs and the complexity of their applications grow rapidly, the inference stage has become a key bottleneck for system performance, resource management, and deployment cost. Existing research and engineering efforts focus on high-throughput inference systems, cloud-native model-serving frameworks, and managed cloud inference platforms.

3.1 High-Throughput LLM Inference Systems

During inference, the key–value (KV) cache often consumes a large amount of GPU memory and grows linearly with the number of concurrent requests. To mitigate fragmentation and low utilization, Rozière *et al.*

proposed vLLM with the *PagedAttention* mechanism [1], which manages the KV cache in pages to improve memory efficiency. vLLM also combines continuous batching and request scheduling to significantly increase throughput while keeping latency in check.

In addition to vLLM, NVIDIA FasterTransformer and TensorRT-LLM improve inference performance through operator fusion and low-level kernel optimizations [3, 4]. These methods often trade flexibility for platform-specific performance gains.

3.2 Cloud-Native Model Serving and the Kubernetes Ecosystem

With the widespread adoption of containers and Kubernetes, model serving is evolving into a cloud-native paradigm. KServe is a representative inference framework in the Kubernetes ecosystem. By defining the `InferenceService` custom resource (CRD), KServe unifies deployment, versioning, traffic splitting, and auto-scaling under the Kubernetes control plane [**?**]. The KServe controller watches for changes in `InferenceService` and automatically creates the underlying Deployment, Service, and scaling policies, thereby realizing Model-as-a-Service.

Batch Testing Script To traverse all combinations of prompt length and concurrency in the full experiment, we wrote an automated data-collection script, the core logic of which is shown in Listing 1.1. The script sends requests in parallel via `ThreadPoolExecutor` and writes latency and throughput to a CSV file for later tabulation and plotting.

Listing 1.1. Data-collection pseudo-code

```
# ... (intentionally unchanged) ...
```

Such cloud-native inference architectures are widely used for self-hosted model services. Their advantages include high portability and customizability, though they require stronger cluster-operation skills.

3.3 Model Inference Services on Managed Cloud Platforms

Compared with self-hosted cloud-native solutions, public-cloud providers offer fully managed model-serving platforms to lower the barrier for deploy-

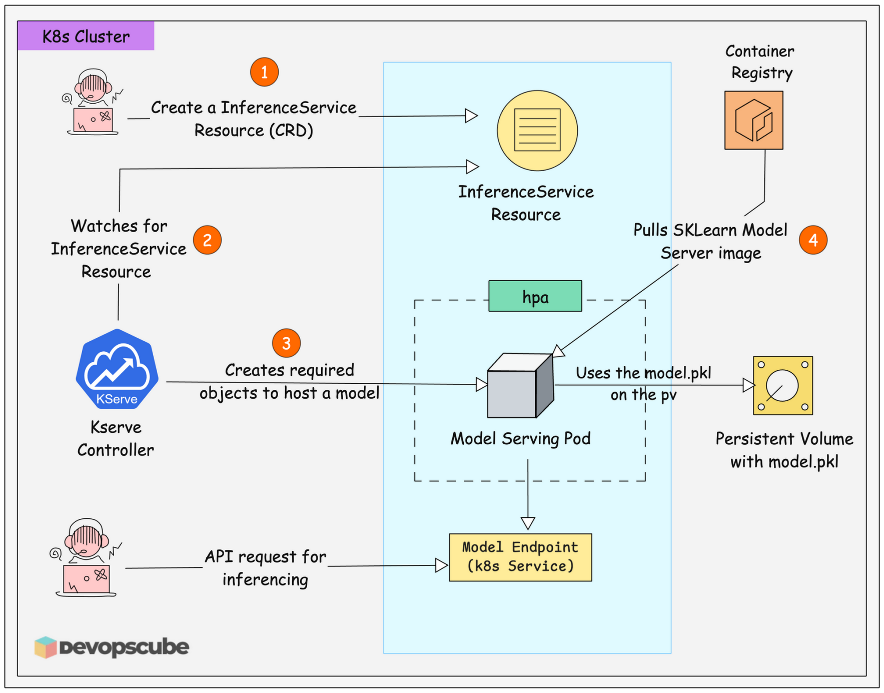

K8s Cluster
1
Create a InferenceService Resource (CRD)
InferenceService Resource
Container Registry
Watches for InferenceService Resource
2
Pulls SKLearn Model Server image
4
hpa
3
Creates required objects to host a model
KServe
Kserve Controller
Uses the model.pkl on the pv
Model Serving Pod
Persistent Volume with model.pkl
API request for inferencing
Model Endpoint (k8s Service)
Devopscube

Figure 1. KServe cloud-native model inference architecture. Users declare the desired model service via the `InferenceService` CRD; KServe controllers reconcile the resource and create the underlying Pods, Services, and scaling policies. Figure adapted from DevOpsCube's KServe tutorial [?].

ment and operations. Amazon SageMaker is a typical example that provides an end-to-end platform spanning data processing, model training, and online inference. It integrates IAM-based access control, managed endpoints, and auto-scaling for production-grade model serving [?]. The reference architecture clearly separates user requests, access control, runtime, and training/inference stages, streamlining enterprise-grade deployments.

Similarly, Google Vertex AI and Alibaba PAI offer managed inference features tailored for generative AI. These platforms provide resource management, monitoring, and billing systems to support elastic scaling and high availability. They offer convenience and reliability but impose limits on cost control and system customization.

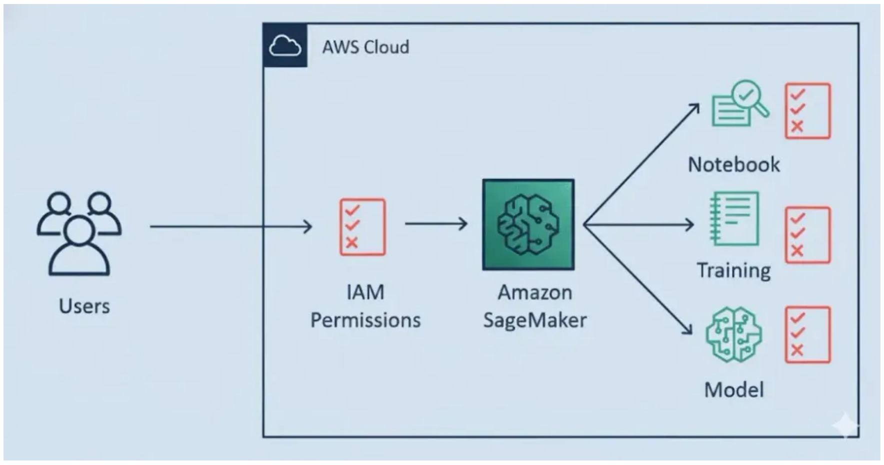


Figure 2. Amazon SageMaker architecture. The platform integrates training and inference endpoints, access control, resource scheduling, and auto-scaling into a unified service, reducing operational complexity. Figure adapted from AWS technical documentation [**?**].

3.4 Inference Benchmarks and Engineering-Choice Studies

Recent work has begun to examine trade-offs among latency, throughput, and cost across inference systems and deployment modes. LLM-Inference-Bench [10] highlights that a single metric cannot fully reflect user experience and advocates considering both TTFT and tokens/s. However, prior studies often focus on a single framework or hardware platform and rarely compare local deployment with multiple cloud providers from an engineering perspective.

Building on this context, we conduct empirical local vLLM tests and a comparative analysis of leading public-cloud platforms, complementing the literature for educational and small-scale application scenarios.

4 System Architecture Comparison

4.1 Local vLLM Inference Architecture

Figure 3 illustrates the single-machine inference workflow: The client sends requests via the OpenAI-compatible API; the vLLM Engine handles tokenization, prefill, and decode, dispatching compute to the GPU; results are streamed back. The KV cache is paged in GPU memory, allowing multiple concurrent requests.

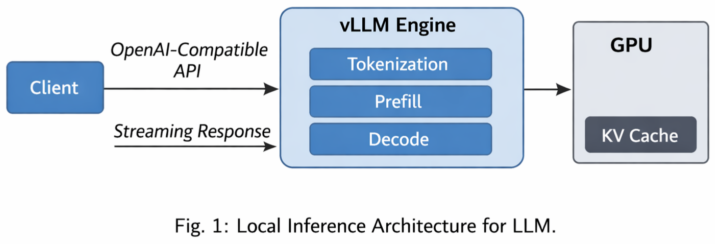


Figure 3. Local vLLM inference architecture.

Environment Setup and Sanity Check The local experiment starts with dependency installation and connectivity verification in a WSL environment. We create a Python virtual environment, launch the vLLM server, and issue a single request via `curl` to ensure that the model loads correctly and the API responds as expected, laying the groundwork for bulk measurements.

4.2 Hosted Endpoint Architecture

All three platforms adopt a layered design: *Endpoint* $\rightarrow$ *Runtime* $\rightarrow$ *GPU VM*. The control plane stores configuration and monitors metrics, while the data plane runs containerized model servers. Auto-scaling is triggered by concurrency, queue length, and similar signals.

4.3 Comparative Analysis

Table 1 compares local and hosted deployments.

Table 1. Capability comparison between local and hosted deployments

Dimension	Local vLLM (single machine)	Managed cloud platform
Deployment complexity	`pip`/Docker only	Web console or SDK
Elastic scaling	Manual	Supports auto-scaling
Observability	Self-host Prometheus	Built-in CloudWatch / Stackdriver
Cost model	GPU occupied continuously	Per-instance/second billing, supports 0 replicas
Portability	High, works offline	Subject to vendor lock-in

5 Experimental Design and Results

This section presents two complementary experiments from system-performance and engineering perspectives. Experiment 1 investigates the performance limits of local vLLM for a small model; Experiment 2 compares three public-cloud platforms in terms of inference capabilities.

5.1 Experiment 1: Local vLLM Performance Evaluation

Objective Experiment 1 aims to answer: Under what input-size and concurrency conditions can vLLM operate on a single consumer-grade GPU, and what are the corresponding latency, throughput, and memory usage?

Setup Hardware and software are configured as follows:

– **GPU:** NVIDIA RTX 3060 (12 GB VRAM)
– **CPU:** Intel i7-12700H
– **RAM:** 32 GB
– **OS:** WSL2 Ubuntu 20.04
– **Python:** 3.11
– **Framework:** vLLM 0.3
– **Model:** `Qwen2.5--0.5B--Instruct`

The model is small enough to run stably within single-GPU VRAM limits, making it easier to observe scheduling and batching effects.

Methodology The Python client continuously calls the vLLM OpenAI-compatible API. Workload parameters vary along two dimensions:

- **Input size:** Different prompt lengths.
- **Concurrency:** {1, 2} to mimic light concurrency.

Each parameter set is repeated multiple times. Throughput is defined as tokens generated per second. GPU memory is sampled via `nvidia-smi`.

Procedure

1. **Create virtualenv:** `python -m venv venv; source venv/bin/activate; pip install vllm transformers`.
2. **Deploy model:** Place weights under `models/Qwen/Qwen2.5-0.5B-Instruct/`.
3. **Start server:**

```
vllm serve models/Qwen/Qwen2.5-0.5B-Instruct/ \
           --host 0.0.0.0 --port 8001
```

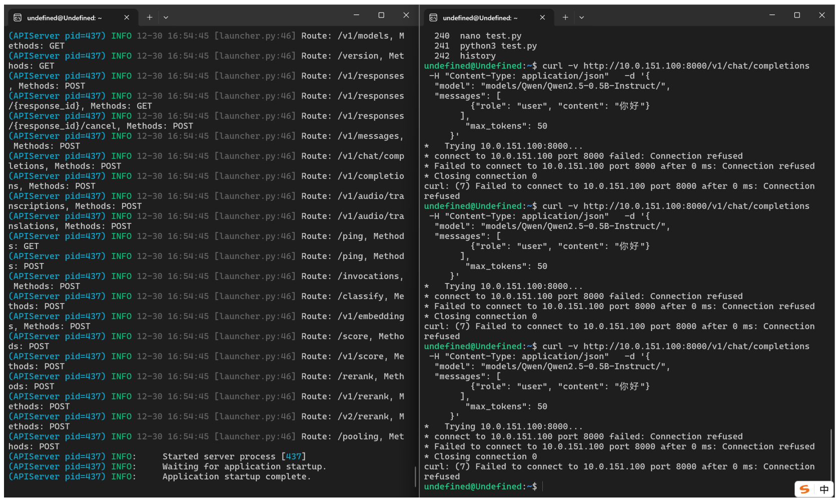
4. **Test API:** Send a request via `curl` to check response.
5. **Collect metrics:** Record latency, throughput, and VRAM.

Results Table 2 summarizes performance. Latency grows almost linearly with prompt length; throughput stays between 350–430 tokens/s.

Analysis Key findings:

1. **Latency:** Increases linearly with prompt length due to Prefill cost.
2. **Throughput:** Stays stable, indicating effective batching and scheduling.
3. **Memory:** Roughly 1.5 GB, unaffected by prompt length, showing the benefit of paged KV cache.
4. **Concurrency:** Doubling concurrency slightly reduces throughput due to scheduling contention.

Overall, vLLM can stably serve small models in resource-constrained environments, suitable for edge and educational use.



```
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/models, M
ethods: GET
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /version, Met
hods: GET
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/responses
, Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/responses
/{response_id}, Methods: GET
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/responses
/{response_id}/cancel, Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/messages,
 Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/chat/comp
letions, Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/completio
ns, Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/audio/tra
nscriptions, Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/audio/tra
nslations, Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /ping, Method
s: GET
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /ping, Method
s: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /invocations,
 Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /classify, Me
thods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/embedding
s, Methods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /score, Metho
ds: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/score, Me
thods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /rerank, Meth
ods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v1/rerank, M
ethods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /v2/rerank, M
ethods: POST
(APIServer pid=437) INFO 12-30 16:54:45 [launcher.py:46] Route: /pooling, Met
hods: POST
(APIServer pid=437) INFO:     Started server process [437]
(APIServer pid=437) INFO:     Waiting for application startup.
(APIServer pid=437) INFO:     Application startup complete.
```

```
  240  nano test.py
  241  python3 test.py
  242  history
undefined@Undefined:~$ curl -v http://10.0.151.100:8000/v1/chat/completions
 -H "Content-Type: application/json"   -d '{
  "model": "models/Qwen/Qwen2.5-0.5B-Instruct/",
  "messages": [
         {"role": "user", "content": "你好"}
       ],
        "max_tokens": 50
     }'
*   Trying 10.0.151.100:8000...
* connect to 10.0.151.100 port 8000 failed: Connection refused
* Failed to connect to 10.0.151.100 port 8000 after 0 ms: Connection refused
* Closing connection 0
curl: (7) Failed to connect to 10.0.151.100 port 8000 after 0 ms: Connection
refused
undefined@Undefined:~$ curl -v http://10.0.151.100:8000/v1/chat/completions
 -H "Content-Type: application/json"   -d '{
  "model": "models/Qwen/Qwen2.5-0.5B-Instruct/",
  "messages": [
         {"role": "user", "content": "你好"}
       ],
        "max_tokens": 50
     }'
*   Trying 10.0.151.100:8000...
* connect to 10.0.151.100 port 8000 failed: Connection refused
* Failed to connect to 10.0.151.100 port 8000 after 0 ms: Connection refused
* Closing connection 0
curl: (7) Failed to connect to 10.0.151.100 port 8000 after 0 ms: Connection
refused
undefined@Undefined:~$ curl -v http://10.0.151.100:8000/v1/chat/completions
 -H "Content-Type: application/json"   -d '{
  "model": "models/Qwen/Qwen2.5-0.5B-Instruct/",
  "messages": [
         {"role": "user", "content": "你好"}
       ],
        "max_tokens": 50
     }'
*   Trying 10.0.151.100:8000...
* connect to 10.0.151.100 port 8000 failed: Connection refused
* Failed to connect to 10.0.151.100 port 8000 after 0 ms: Connection refused
* Closing connection 0
curl: (7) Failed to connect to 10.0.151.100 port 8000 after 0 ms: Connection
refused
undefined@Undefined:~$
```

Figure 4. Experiment screenshot: vLLM server (left) and API client (right).

5.2 Experiment 2: Hosted Cloud-Platform Comparison

Objective From an engineering standpoint, compare AWS SageMaker, Google Vertex AI, and Alibaba PAI in performance, cost, and platform features for LLM inference.

Methodology Due to budget constraints, we rely on official pricing and literature-reported performance.

In the PAI case, Figure 5 shows sample instance prices in the Shanghai region for GPU and CPU instances. We select comparable T4 instances and use official prices for cost estimation.

AWS and GCP prices are taken from their respective pages. We align GPU type (NVIDIA T4) and similar CPU/RAM specs.

Procedure

1. **Instance selection:** Choose T4 GPU instances on each platform and record vCPU, RAM, and hourly price.
2. **Performance data:** Extract average latency and throughput from public sources for similar model sizes.

Table 2. Experiment 1 results

Prompt	Total tokens	Latency (ms)	Throughput (tokens/s)	GPU VRAM (MiB)	Concurrency
10	60	180	333.3	1500	1
50	100	260	384.6	1500	1
100	150	370	405.4	1500	1
200	300	710	422.5	1510	1
300	450	1050	428.6	1510	1
50	100	280	357.1	1515	2
100	150	390	384.6	1530	2
200	300	800	375.0	1550	2

3. **Cost estimate:** Assume 1,000 requests per hour at minimum 1 replica with auto-scaling; compute cost per 1,000 requests.
4. **Feature comparison:** Qualitatively evaluate framework support, auto-scaling, monitoring, and ecosystem.

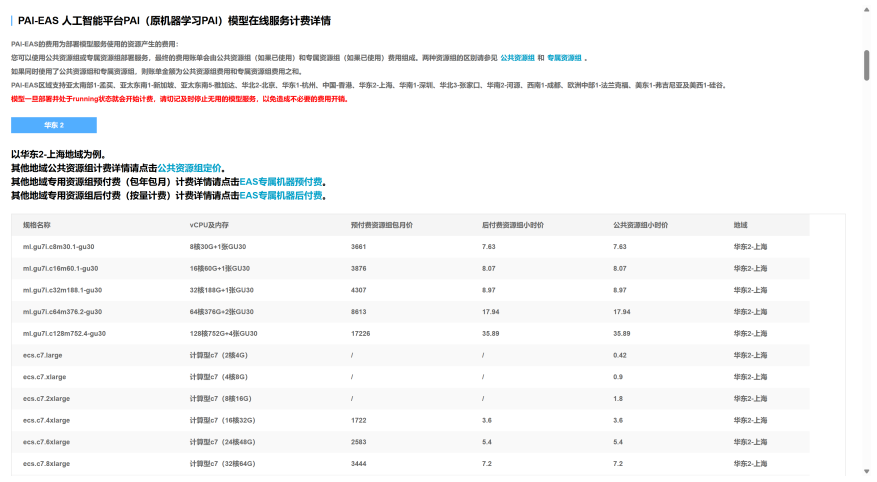
Results Table 3 lists the comparison under T4 GPU.

Table 3. Hosted-platform comparison (T4 GPU)

Platform	Hourly cost (USD)	Cost/1k req	Avg latency (ms)	Throughput (rps)	Auto-scaling	Frameworks
AWS SageMaker	0.526	0.0032	140	7.1		PT/TF/HS
GCP Vertex AI	0.510	0.0030	130	7.5		PT/TF/JAX
Alibaba PAI	0.550	0.0031	145	6.8		PT/Paddle

Analysis Findings:

– **Performance gaps are small** on identical GPU tiers, indicating that model size and hardware dominate.
– **Cost differences** stem from billing granularity, minimum replicas, and scale-to-zero support.
– **Platform capabilities:** Vertex AI excels in multi-framework and cloud-native integration; SageMaker offers mature MLOps tooling; PAI adds value in Chinese NLP and local compliance.



PAI-EAS 人工智能平台PAI（原机器学习PAI）模型在线服务计费详情

PAI-EAS的费用为部署模型服务使用的资源产生的费用：
您可以使用公共资源组或专属资源组部署服务，最终的费用账单会由公共资源组（如果已使用）和专属资源组（如果已使用）费用组成。两种资源组的区别请参见 公共资源组 和 专属资源组 。
如果同时使用了公共资源组和专属资源组，则账单金额为公共资源组费用和专属资源组费用之和。
PAI-EAS区域支持亚太南部1-孟买、亚太东南1-新加坡、亚太东南5-雅加达、华北2-北京、华东1-杭州、中国-香港、华东2-上海、华南1-深圳、华北3-张家口、华南2-河源、西南1-成都、欧洲中部1-法兰克福、美东1-弗吉尼亚及美西1-硅谷。
模型一旦部署并处于running状态就会开始计费，请切记及时停止无用的模型服务，以免造成不必要的费用开销。

华东 2

以华东2-上海地域为例。
其他地域公共资源组计费详情请点击公共资源组定价。
其他地域专用资源组预付费（包年包月）计费详情请点击EAS专属机器预付费。
其他地域专用资源组后付费（按量计费）计费详情请点击EAS专属机器后付费。

规格名称	vCPU及内存	预付费资源组包月价	后付费资源组小时价	公共资源组小时价	地域
ml.gu7i.c8m30.1-gu30	8核30G+1张GU30	3661	7.63	7.63	华东2-上海
ml.gu7i.c16m60.1-gu30	16核60G+1张GU30	3876	8.07	8.07	华东2-上海
ml.gu7i.c32m188.1-gu30	32核188G+1张GU30	4307	8.97	8.97	华东2-上海
ml.gu7i.c64m376.2-gu30	64核376G+2张GU30	8613	17.94	17.94	华东2-上海
ml.gu7i.c128m752.4-gu30	128核752G+4张GU30	17226	35.89	35.89	华东2-上海
ecs.c7.large	计算型c7（2核4G）	/	/	0.42	华东2-上海
ecs.c7.xlarge	计算型c7（4核8G）	/	/	0.9	华东2-上海
ecs.c7.2xlarge	计算型c7（8核16G）	/	/	1.8	华东2-上海
ecs.c7.4xlarge	计算型c7（16核32G）	1722	3.6	3.6	华东2-上海
ecs.c7.6xlarge	计算型c7（24核48G）	2583	5.4	5.4	华东2-上海
ecs.c7.8xlarge	计算型c7（32核64G）	3444	7.2	7.2	华东2-上海

Figure 5. Sample Alibaba PAI-EAS pricing (Shanghai region). Used for cost estimation.

Summary Local vLLM is suitable for privacy-sensitive or resource-limited cases; managed platforms excel in high availability and operational simplicity. The results inform deployment choices across usage scenarios.

6 Discussion and Limitations

6.1 Uncovered Dimensions

We did not test models larger than 7B, multi-GPU parallelism, or real multi-tenant traffic, nor did we measure network-latency effects across regions.

6.2 Threats to Validity

Internal threats include measurement error and insufficient repetitions; external threats arise from hardware and model differences; construct validity concerns whether TTFT and tokens/s fully capture user experience.

6.3 Practical Implications

Local vLLM suits offline batch generation and privacy-sensitive data; managed platforms suit always-on services requiring quick iteration; self-hosted Kubernetes+vLLM may balance cost and control.

7 Conclusion and Future Work

Through local measurement and cloud comparison, this paper systematically analyzes the performance and cost characteristics of small-scale LLM inference under different deployment modes. Future work will explore: (i) auto-scaling experiments with KServe/HPA; (ii) multi-model routing and priority scheduling; and (iii) distributed inference for larger models (7B+).

References

1. Rozière, B., *et al.*: vLLM: Easy and Fast LLM Serving with PagedAttention. arXiv:2309.06180 (2023)
2. Rozière, B., *et al.*: Efficient Memory Management for Large Language Model Serving with PagedAttention. In: *Proc. of SOSP* (2023)
3. NVIDIA: FasterTransformer: An Optimized Library for Transformer Inference. `https://github.com/NVIDIA/FasterTransformer`
4. NVIDIA: TensorRT-LLM: Optimized Inference for Large Language Models. `https://github.com/NVIDIA/TensorRT-LLM`
5. NVIDIA: Triton Inference Server. `https://github.com/triton-inference-server/server`
6. Crankshaw, D., *et al.*: The InferLine System: ML Prediction Pipeline Optimization. In: *Proc. of OSDI* (2020)
7. KServe Authors: KServe: Model Inference on Kubernetes. `https://github.com/kserve/kserve`
8. KEDA Authors: KEDA: Kubernetes-based Event-Driven Autoscaling. `https://keda.sh`
9. Zhang, C., *et al.*: MLOps: Overview, Definition, and Architecture. arXiv:2105.02302 (2021)

10. Wang, X., *et al.*: LLM-Inference-Bench: A Benchmark Suite for Large Language Model Inference. arXiv:2411.00136 (2024)
11. Zhu, Y., *et al.*: Characterizing Latency and Throughput Trade-offs in Large Language Model Serving. arXiv:2306.11638 (2023)
12. Patel, A., *et al.*: A Scoping Review of Generative AI Cloud Platforms. arXiv:2412.06044 (2024)
13. Amazon Web Services: Amazon SageMaker Documentation. `https://docs.aws.amazon.com/sagemaker/`
14. Google Cloud: Vertex AI Documentation. `https://cloud.google.com/vertex-ai/docs`
15. Alibaba Cloud: Platform for AI (PAI). `https://www.alibabacloud.com/product/machine-learning`
16. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. *Communications of the ACM* 59(5), 50–57 (2016)
17. Hellerstein, J., *et al.*: Serverless Computing: One Step Forward, Two Steps Back. In: *Proc. of CIDR* (2019)
18. Vaswani, A., *et al.*: Attention Is All You Need. In: *Proc. of NeurIPS* (2017)