Xinference: Making Large Model Serving Easy

Weizheng Lu[1], **Lingfeng Xiong**[1], **Feng Zhang**[1], **Xuye Qin**[2*], **Yueguo Chen**[1*]
[1]Renmin University of China, [2]Xorbits Inc.
{luweizheng,lenfeng2022,fengzhang,chenyueguo}@ruc.edu.cn, qinxuye@xprobe.io

Abstract

The proliferation of open-source large models necessitates dedicated tools for deployment and accessibility. To mitigate the complexities of model serving, we develop Xinference, an open-source library designed to simplify the deployment and management of large models. Xinference effectively simplifies deployment complexities for users by (a) preventing users from writing code and providing built-in support for various models and OpenAI-compatible APIs; (b) enabling full model serving lifecycle management; (c) guaranteeing efficient and scalable inference and achieving high throughput and low latency. In comparative experiments with similar products like BentoML and Ray Serve, Xinference outperforms these tools and offers superior ease of use. Xinference is available at `https://github.com/xorbitsai/inference`.

1 Introduction

Recently, open-source large models are quickly catching up with the closed-source models (MetaAI, 2024; Google, 2024; Yang et al., 2024a). There is a growing demand to deploy these open-source models in private user environments, as an increasing number of AI applications and even non-ML/AI practitioners require a straightforward and effective inference toolkit for managing model deployment. Although there are inference engines and frameworks for large model inference (Miao et al., 2023), many current inference toolkits are not as simple and convenient to use. Therefore, this paper focuses on how to build an efficient and easy-to-use inference toolkit.

Streamlining large model inference is crucial. Open-source large models, customizable and free from data privacy concerns, are highly suitable for private deployment. A simple toolkit can enable more users to access the capabilities of large models and focus on AI applications rather than spend time managing inference services.

However, building an easy-to-use inference toolkit is a non-trivial task. First, AI applications often rely on different types of models, such as chat, embedding, or multimodal models, along with technologies like function calling (Wang et al., 2024) or retrieval-augmented generation (RAG) (Karpukhin et al., 2020). Each model type or technology mentioned possesses distinct characteristics and may require specific configurations. Second, the landscape of inference engines and hardware is vast, with options like vLLM (Kwon et al., 2023), llama.cpp (Gerganov, 2023), SGLang (Zheng et al., 2024), and various CPUs and GPUs, such as x86, Apple Silicon, NVIDIA, AMD. A particular inference engine is typically designed for specific users and application scenarios. Third, users need to scale inference workloads onto clusters to achieve high throughput and low latency. Therefore, there is a need for a framework-agnostic inference toolkit to manage multiple models and various inference engines while providing users with convenient and user-friendly services.

Several toolkits, such as Ray Serve (Moritz et al., 2018; Ray Team, 2024) and BentoML (Yang et al., 2024b), aim to facilitate serving services for deploying various models across diverse hardware platforms. However, we have found that these tools fail in user-friendliness, often requiring extensive coding, or suffer performance degradation. For example, first, BentoML and Ray Serve both require users to write code to deploy models, which can be quite challenging for users who are not familiar with model inference. Second, these two tools do not cover the full model serving lifecycle and lack essential features. Third, BentoML suffers performance degradation when scaling models replicas.

To address the aforementioned issues, we have developed **Xinference**, an inference toolkit de-

*Corresponding authors: Xuye Qin and Yueguo Chen.

Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing System Demonstrations, pages 291–300
November 12-16, 2024 ©2024 Association for Computational Linguistics

signed to streamline the serving of large models. First, it is designed for ease of use, eliminating the need for users to write additional code, and provides built-in support for various of models, features, and inference engines. Second, it can manage the entire lifecycle of model serving, from scaling models to clusters to managing computing resources. Third, it has minimal performance loss when integrating with an inference engine, ensuring high throughput and low latency on clusters, and offers scheduling optimizations. Xinference leverages Xoscar (Xorbits, 2024; Lu et al., 2024), an actor programming framework we designed, as its core component to manage machines, devices, and inference engines. Each actor serves as a basic unit for model inference tasks and different inference engines can be integrated into the actor, enabling us to manage multiple inference engines and model replicas.

Xinference targets a broad range of audiences, including AI application developers, ML engineers, and even non-AI/ML practitioners who simply wish to use large AI models. It is open-sourced with the Apache 2.0 license and available on GitHub[1].

Experiments demonstrate that, compared to BentoML and Ray Serve, Xinference maintains excellent latency and throughput across various workload scenarios. When deploying a single model replica, compared to the original inference engine, Xinference's performance loss is within 3.64%.

2 Background and Motivation

In this section, we outline the motivation and the design principles for the user-centric inference service.

2.1 User-Centric Design for Inference

Building a large model inference service typically requires three modules: the inference engine, the model specification, and an endpoint or a Web User Interface (Web UI). Inference engines are the backend for model serving, with notable works such as PyTorch (Paszke et al., 2019; Ansel et al., 2024), Transformers (Wolf et al., 2020), vLLM (Kwon et al., 2023), and llama.cpp (Gerganov, 2023). Typically, these inference engines can only serve with one model replica, lacking the capability to scale out. To manage different models, tools such as BentoML (Yang et al., 2024b) and Ray Serve (Ray Team, 2024) require users to write code and provide model specifications for configuration. These model specifications include the system prompt and the end-of-sequence (EOS) token of the model, settings for ingress traffic, as well as replica and device management, among other configurations. These tools offer OpenAI-style endpoints but do not provide a Web UI to assist users with the aforementioned configuration and management tasks. Moreover, they do not cover all aspects of model serving lifecycle, leaving users to manage these stuff themselves. Therefore, we develop Xinference to address the usability issues of large model serving.

2.2 Design Principles

To provide an easy-to-use inference service, we adhere to the following principles in the design and implementation of Xinference.

- **Simplicity.** Users do not need to write code or configure model specifications; these configurations are integrated and implemented by the serving toolkit. Users simply need to specify which model to launch via the Web UI or command line. The toolkit should be *engine/hardware-agnostic* and can integrate various inference engines and different hardware. The toolkit supports fully OpenAI-compatible APIs and offers all model types and features, including function calling. All these features will facilitate users' easy migration of their applications from closed-source models to this toolkit.
- **Full Lifecycle Management.** The toolkit should handle the entire lifecycle of model serving, allowing users to launch and utilize models as well as monitor and terminate them. It can also manage computing resources and enable models to scale across a cluster.
- **Efficiency.** When using inference engines like vLLM, Xinference should not bring extra performance loss, and with multiple model replicas, it should guarantee high throughput and low latency. It can provide necessary optimizations like continuous batching.

Table 1 compares Xinference and other platforms, highlighting Xinference's features.

[1] https://github.com/xorbitsai/inference

Table 1: A comparative feature analysis that showcases the strengths of Xinference. The ✓ symbol indicates built-in support within the framework, and ❍ denotes that the framework requires users to implement the functionality by writing additional code by users themselves.

Feature	BentoML	Ray Serve	Xinference
OpenAI-style Endpoint	✓	✓	✓
Web UI	❍	❍	✓
Cluster Deployment	✓	✓	✓
Serving Lifecycle Management	✓	❍	✓
External Tool Function Calling	❍	❍	✓
Muli-Inference Engines Support	❍	❍	✓
Muli-Hardware Support	❍	❍	✓
Multi-Types Models Support	❍	❍	✓

3 Xinference Usage: Designing for User-Friendliness

This section discusses Xinference's usage and highlights its user-friendly features. We describe the following aspects: launching services, managing the model serving lifecycle, interacting with its user interfaces, integrating inference engines, multi-tenant serving, and use case study.

3.1 Launch Service

Xinference can be deployed on a local machine or a cluster.

Local Server. On a local machine, users can execute the following command to start the service. Then, users can access the Web UI by visiting `http://127.0.0.1:9997/`.

```
xinference-local --port 9997
```

Cluster. To start a Xinference cluster, users need to execute the following commands:

```
# on the supervisor server
xinference-supervisor -H '${sv_host}'

# on the worker server
xinference-worker -e 'http://${sv_host
   }:9997'
```

Users should first launch the supervisor, and start workers on other servers. The supervisor is responsible for coordination, whereas the worker manages the available resources (i.e., CPUs or GPUs) and executes the inference requests. The workers establish connections to the supervisor, thereby setting up a Xinference cluster. In the local mode, both the supervisor and worker are launched on the same local computer.

3.2 Model Serving Lifecycle

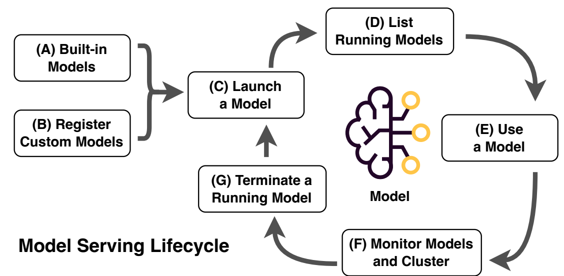


Figure 1: Lifecycle of model serving.

Xinference manages the entire process of model serving, as illustrated in Figure 1. Figure 2 is the Web UI with each stage in the lifecycle denoted. Xinference's lifecycle of model serving is centered around models, including managing models (using built-in open-source models or registering custom models), launching a model, listing running models, using a model, monitoring, and terminating running models. Here, we highlight only a few key stages that make Xinference a user-friendly platform different from other toolkits.

Launch a Model. During this step, Xinference helps users choose an inference engine and a quantization method. Xinference automatically detects available devices on the machine and provides corresponding options. For instance, on a Mac laptop, Xinference suggests engines such as PyTorch and MLX. On a server equipped with NVIDIA GPUs, it recommends options like vLLM, SGLang, or llama.cpp. Xinference checks the chosen engine and quantization settings, eliminating the need for users to worry about installing quantization libraries or selecting the right quantization methods. Moreover, Xinference supports LoRA (Hu et al., 2022) fine-tuned models, which are commonly used by enterprises to tackle domain-specific issues.

Using a Model. Users can interact with a model through the OpenAI-compatible RESTful API. Unlike other tools or frameworks that support only a subset of OpenAI APIs, Xinference fully supports all model types and features. As shown in Table 2, Xinference offers built-in support for various model types and model families, including chat, generate,

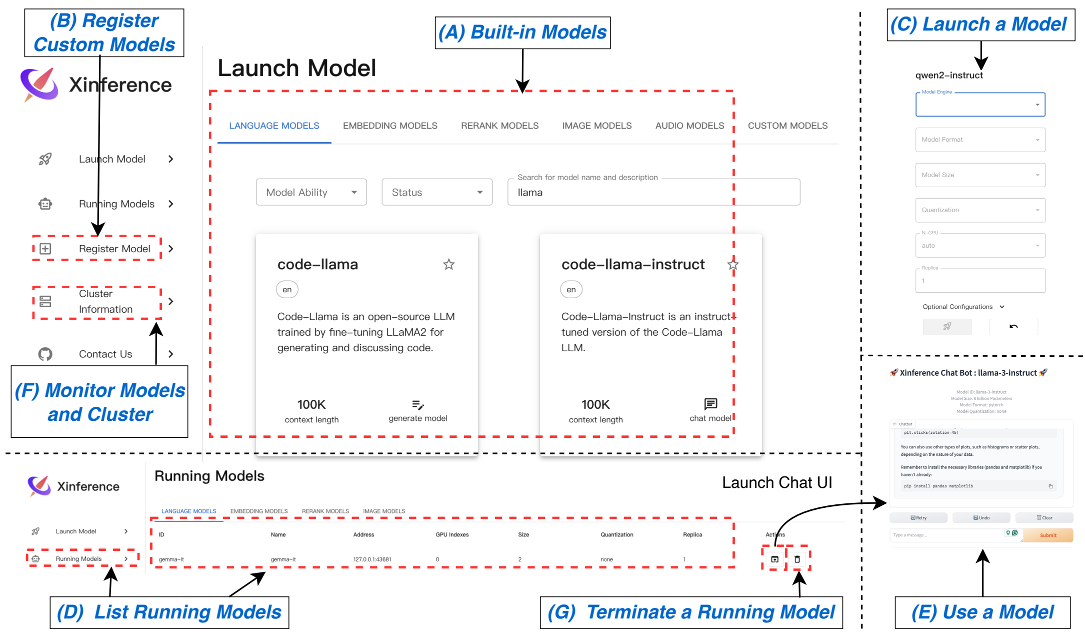


Figure 2: Web UI of Xinference, with each stage of serving lifecycle denoted.

vision language, embedding, rerank, audio, and image. Xinference releases new versions weekly, supporting the latest models published within that week. Besides various models, we also support external tool function calling, which is crucial for building agents. With these models and APIs, Xinference can serve as a drop-in replacement for OpenAI, while other framework users need to write additional code to build a specific model service.

Table 2: Xinference supports a wide range of models. The abbreviations for the 'Type' column are as follows. **C**: Chat, **G**: Generate, **VL**: Vision Language, **E**: Embedding, **R**: Rerank, **A**: Audio, and **I**: Image. The 'Models' column counts the number of model families; for example, the Llama 3.1 is a model family with 8B, 70B, and 405B parameters.

Type	Models	Example Model
C	81	Llama 3.1 instruct (MetaAI, 2024)
G	33	Code Llama (Rozière et al., 2024)
VL	8	Qwen-VL (Bai et al., 2023)
E	29	BGE Embedding (Xiao et al., 2023)
R	7	BGE Reranker (Xiao et al., 2023)
A	16	Whisper (Radford et al., 2023)
I	8	Stable Diffusion (Rombach et al., 2022)

3.3 User Interface

We offer users easy-to-use interfaces to interact with our system.

Web UI. Users can access the Web UI in their browser, as illustrated in Figure 2. The entire life cycle of the model serving can be completed on the graphical user interfaces. This interface suits beginners or non-AI/ML practitioners with limited technical knowledge.

Command Line and RESTful Client. Users can also interact with Xinference on the node where the supervisor is located using command lines such as `xinference launch` for launching a model, and `xinference terminate` for shutting down a model. Xinference also offers RESTful APIs that enable users to perform the aforementioned model management tasks using Python, Node.js, or curl scripts. The command lines and RESTful client target users with programming experience.

3.4 Inference Engines

Our platform currently supports five state-of-the-art inference engines: PyTorch, vLLM, SGLang, llama.cpp, and MLX. PyTorch is a widely used framework for both training and inference, with numerous models initially released based on the Transformers library. However, it is not a dedicated large model serving toolkit and may not be ideal for high-concurrency scenarios. To enhance performance, Xinference implements continuous batching (Yu et al., 2022), which is compatible with all Transformers models. This feature enables models without dedicated engine support to achieve substantial throughput enhancements. To help users make informed choices on selecting the optimal inference engine, we conduct benchmarks and show results in Appendix A.3.

3.5 Multi-tenant Serving

To support users operating in a multi-user or multi-tenant setting, we offer features such as user authentication and isolation of computing resources.

User Authentication. Xinference currently provides user authentication, ensuring that access to the Xinference service is limited to verified users, thereby enhancing security.

Computing Resource Isolation. Xinference itself is incapable of isolating computing resources. Users can deploy Xinference using the Kubernetes Helm charts or Docker images we provide to enable effective resource isolation and avoid resource contention with other software.

3.6 Use Case Study

Xinference has been integrated into many well-known AI tools, such as LlamaIndex (Liu, 2022), a retrieval framework, and Dify (Zhang, 2023), an AI application development platform. As a specific example of an AI application, LangChain-Chatchat[2] is a popular question answering application on GitHub. It enable users to build RAG or agent applications based on local knowledge bases and utilizes Xinference as its default inference service toolkit.

4 System Implementation

In this section, we show how Xinference manages models and supports scalable inference.

4.1 Architecture

Figure 3a illustrates the system architecture of Xinference, which consists of three layers: API, Core Service, and Actor. The API layer offers users RESTful APIs based on FastAPI. The Core Service layer implements several actor classes based on Xoscar, with key actor classes including `SupervisorActor`, `WorkerActor`, `ModelActor`, etc. Xoscar is a lightweight Python actor framework that we have developed, which abstracts low-level concurrency, communication, and device management tasks.

4.2 Core Service on Actor

In our system, the `SupervisorActor` plays a key role in management and coordination, supervising multiple `WorkerActors`. Each `WorkerActor` manages computing resources and several `ModelActors`, which load and execute models within the ActorPool. The ActorPool is like a resource pool that manages all CPU and GPU computing devices.

Actor Call Workflow. Figure 3b illustrates the workflow of a `launch` request from a user. Upon receiving the request, the RESTful API sends a message instructing the `SupervisorActor` to execute `launch_builtin_model`. The `SupervisorActor` then communicates with the `WorkerActor`, which checks for available computing resources across all workers within the actor pool and allocates GPU devices as needed. Subsequently, the `ModelActor` is invoked to load model checkpoints utilizing a designated inference engine. Note that, in Figure 3b, multiple replicas indicate that `ModelActors` are assigned to multiple GPU devices. After the model is launched, it is assigned a unique model identifier (`model_uid`), which will be returned to the user. In addition to the actor calls in Figure 3b, the `SupervisorActor` records and monitors the newly launched model and the GPU devices the model occupies.

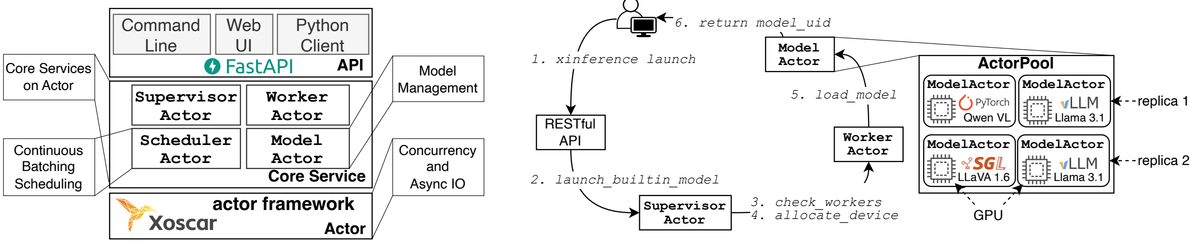
Actor Usage. In Appendix A.1, we describe some usage guides of our actor framework, using `ModelActor` as an example to demonstrate the implementation of model inference tasks and how actors communicate with each other. The corresponding code can be found in Listing 1.

4.3 Scheduling and Concurrency

Continuous Batching. Continuous Batching (CB) is a scheduling mechanism that can substantially enhance throughput and GPU memory utilization in high-concurrency scenarios. We've supported this feature in Xinference by a) incorporating a `SchedulerActor`, which dynamically groups requests into batches; b) developing the batch inference code using PyTorch, while ensuring compatibility with all models in the PyTorch Transformers library.

Concurrency and Async IO. Our inference framework is designed in an asynchronous, non-blocking manner, enabling it to handle concurrent requests. We have extensively used the philosophy of coroutine (Pythons's `asyncio` (Python, 2024)) in our internal implementation. We treat the model inference task as an asynchronous task: we push the task into the actor pool when the request arrives and pull the task when the computing resource is available.

[2] https://github.com/chatchat-space/Langchain-Chatchat



(a) System Architecture.

(b) Actor Call Workflow when `launch` a Model.

Figure 3: Xinference is built on our actor framework.

4.4 Model Management

For the inference engine management part, we have written modular code that includes loading models, formatting prompts, and stopping when encountering EOS tokens. Different models can reuse these codes. We utilize JSON files to manage the metadata of emerging open-source models. Adding a new model does not necessitate writing new code; it merely requires appending new metadata to the existing JSON file. In Appendix A.2, we present a snippet of a JSON file that registers a Llama model.

5 Experiments and Evaluation

5.1 Experimental Setup

We compare Xinference's performance and scalability with BentoML and Ray Serve, two similar frameworks that aim for engine-agnostic serving. We also evaluate the improvements of our scheduling optimization in high-concurrency scenarios. We use Llama 3 8B and 70B models and execute them on three platforms: an on-premises NVIDIA A800 GPU cluster, an Alibaba Cloud instance with NVIDIA A10 GPUs, and a MacBook laptop with Apple M3 chip. We measure latency and throughput, two metrics widely recognized in the industry. Latency is the total average response time, denoting user waiting time. Throughput assesses the number of tokens generated per second by the inference service, and is expressed in tokens per second (token/s).

5.2 Performance: Latency and Throughput

We evaluate the latency and throughput of Xinference, BentoML, and Ray Serve, along with the bare vLLM engine without any wrapper. We conduct experiments on a NVIDIA A800 GPU cluster.

As shown in Figure 4a, Xinference exhibits lower latency with the 70B model. Subsequently, we scale the number of replicas of the 8B model from 1 to 16, conducting tests under two different scenarios. The first is a low concurrency case where we simulate 10 concurrent requests at a time, and the results are depicted in Figure 4b. The second is a high concurrency one where we generate 50 concurrent requests, and the results are presented in Figure 4c. Both Xinference and Ray Serve can scale inference workloads nearly linearly with multiple replicas. While BentoML scales poorly with 8 replicas and cannot directly scale to 16 without third-party tools. In the low concurrency scenario, Xinference demonstrates superior throughput with 4, 8, and 16 replicas. In the high concurrency scenario, Xinference's throughput is on par with Ray Serve. In both scenarios, with a single replica, the performance loss of Xinference compared to the backend inference engine is within 3.64%, while BentoML is 5.66%.

In summary, Xinference can efficiently manage a single model as well as scale to multiple replicas, ensuring high throughput and low latency.

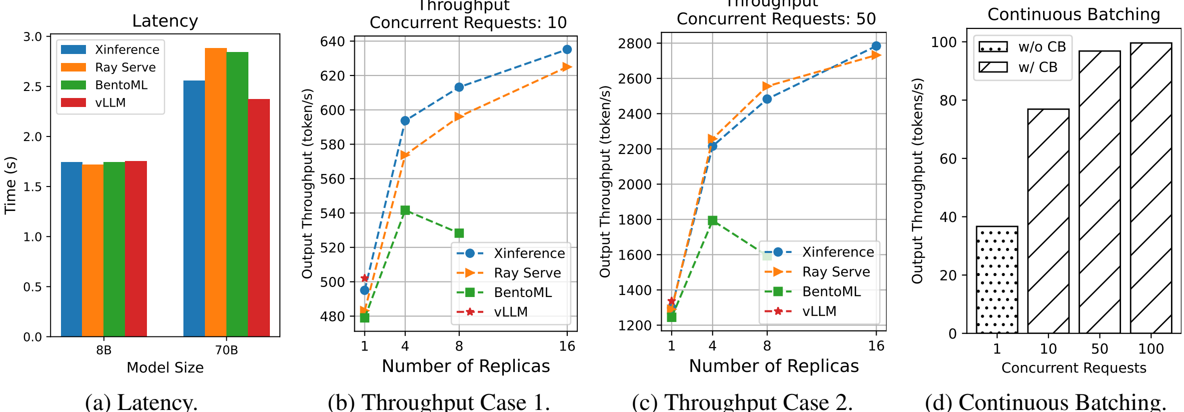
5.3 Scheduling Optimization Analysis

We assess the performance gains of our continuous batching scheduling using the PyTorch backend, as depicted in Figure 4d. In this experiment, we test three concurrency scenarios. The horizontal axis represents the number of concurrent requests that can be handled. The far left is the PyTorch Transformer backend, which lacks continuous batching. It only supports one concurrent request. With continuous batching, Xinference can support higher concurrency levels. When there are 100 concurrent requests, the throughput of Xinference with continuous batching is 2.7× that of PyTorch Transformers without it.

In conclusion, Xinference's continuous batching scheduling effectively enhances throughput, and it can benefit a broader range of models that are only available in the Transformers library.

5.4 Inference Engines Analysis

Xinference can support various inference engines. We test the performance and usability of all Xinfer-



(a) Latency. (b) Throughput Case 1. (c) Throughput Case 2. (d) Continuous Batching.

Figure 4: Xinference's performance comparing with other toolkits.

ence inference engines across three environments. Detailed performance data and discussion are presented in Appendix A.3. With this information, users can make informed choices about the right inference engine.

6 Conclusion

In conclusion, Xinference is a user-friendly tool designed for large model serving. This tool eliminates the need for users to write additional code or configurations, allowing users to focus on building AI applications. It can manage the entire lifecycle of large model serving. Xinference can scale serving workloads onto a cluster and achieve high throughput and low latency. At its foundation, Xinference employs the actor framework that we developed to handle the management of inference engines and hardware.

Acknowledgments

We thank all contributors who have committed code to the Xinference project. This work was partly supported by the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under Grant No.24XNKJ22, and partly supported by the National Science Foundation of China under Grant No.62272466. The computing resources were from the Public Computing Cloud of Renmin University of China and Alibaba Cloud.

Ethics Statement

The Xinference system presented in this paper aims to make large model serving as easy as possible, thereby helping people better access AI models. It's worth noting that Xinference does not supply the model itself, hence it cannot be responsible for the content generated by the model. If our system is used in certain circumstances considered sensitive or critical, it should be used with caution, and the generated content may be investigated by domain experts.

References

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, La Jolla, CA, USA. ACM.

Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond.

Georgi Gerganov. 2023. llama.cpp. `https://github.com/ggerganov/llama.cpp`.

Google. 2024. Gemma: Open models based on gemini research and technology. Technical report, Google DeepMind.

Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for

artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245. William Kaufmann.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, Koblenz Germany. ACM.

Jerry Liu. 2022. LlamaIndex. https://github.com/run-llama/llama_index.

Weizheng Lu, Kaisheng He, Xuye Qin, Chengjie Li, Zhong Wang, Tao Yuan, Xia Liao, Feng Zhang, Yueguo Chen, and Xiaoyong Du. 2024. Xorbits: Automating Operator Tiling for Distributed Data Science. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5211–5223.

MetaAI. 2024. The llama 3 herd of models. https://ai.meta.com/research/publications/the-llama-3-herd-of-models/.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. 2023. Towards Efficient Generative Large Language Model Serving: A Survey from Algorithms to Systems.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577, USA. USENIX Association.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Python. 2024. asyncio — Asynchronous I/O. https://docs.python.org/3/library/asyncio.html.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine Mcleavey, and Ilya Sutskever. 2023. Robust speech recognition via large-scale weak supervision. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pages 28492–28518. PMLR.

Ray Team. 2024. Ray Serve: Scalable and programmable serving. https://github.com/ray-project/ray.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-Pack: Packaged Resources To Advance General Chinese Embedding.

Xorbits. 2024. Xoscar: Python actor framework for heterogeneous computing. https://github.com/xorbitsai/xoscar.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai,

Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024a. Qwen2 Technical Report.

Chaoyu Yang, Sean Sheng, Aaron Pham, Shenyang Zhao, Sauyon Lee, Bo Jiang, Fog Dong, Xipeng Guan, and Frost Ming. 2024b. BentoML: The framework for building reliable, scalable and cost-efficient ai application. `https://github.com/bentoml/bentoml`.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA. USENIX Association.

Luyu Zhang. 2023. Dify. `https://github.com/langgenius/dify`.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs.

A Appendix

A.1 Xoscar Actor Framework

The actor programming model is a paradigm for addressing distributed and concurrency (Hewitt et al., 1973). Each actor is a basic computational unit with certain computing resources and can execute actions or behaviors based on given inputs. Ray (Moritz et al., 2018) is a widely used actor programming framework, while our actor framework is more lightweight. Here, we use the `ModelActor` in Listing 1 as an example to illustrate how we build Xinference with the actor framework.

Listing 1: Code snippet of ModelActor.

```
1  import xoscar as xo
2
3  class ModelActor(xo.Actor):
4    def __init__(self, *args, **kwargs):
5      ...
6    async def load(self):
7      # load checkpoints of a model
8      ...
9    async def generate(self, prompt):
10     # generate content using a model
11     ...
12   async def handle_batch_request(self,
         prompt):
13     # call the SchedulerActor to handle
           continuous batching requests
14     ...
15   async def __post_create__(self):
16     # called after the actor instance is
           created
17     ...
18   async def __pre_destroy__(self):
19     # called before the actor instance is
           destroyed
20     ...
```

Actor Class. Each actor class is a standard Python class that inherits from `xoscar.Actor`. Each actor instance requests resources such as CPU or GPU from the actor pool. There are two special methods worth noting. The `__post_create__` is invoked when the actor is created, allowing for necessary initialization. The `__pre_destroy__` is called when the actor is destroyed, allowing for cleanup or finalization.

Define Actor Actions. Each actor needs to define certain actions or behaviors to accomplish specific tasks. For instance, the `ModelActor` class loads the model and performs model inference. The `load` method loads model checkpoints, the `generate` method generates content given a prompt, and the `handle_batch_request` handles continuous batching requests as it would call the `SchedulerActor`.

Reference Actors and Invoke Methods. When an actor is created, it yields a reference so that other actors can reference it. The actor reference can also be referenced with the IP address. Suppose the `ModelActor` is created and the reference variable is `model_ref`, which can be managed by `WorkerActor`. The `load` method of the `ModelActor` can be invoked by calling `model_ref.load()`.

A.2 Register Model

Listing 2 shows an example of how to register the Llama 3.1 instruct model.

Listing 2: Register Llama 3.1 instruct model in JSON.

```
1  {
2   "model_name": "llama-3.1-instruct",
3   "model_ability": ["chat"],
4   "model_specs": [
5     {
6      "model_format": "ggufv2",
7      "model_size_in_billions": 8,
8      "quantization": ["q8_0", ...],
9      "model_id": "lmstudio-community/Meta-
           Llama-3.1-8B-Instruct-GGUF",
10    },
11    ...
```

```
12 ],
13 "prompt_style": {
14   "style_name": "LLAMA3",
15   "system_prompt": "You are a helpful assistant.",
16   "roles": [ "user", "assistant"],
17   "stop_token_ids": [128001, 128009],
18   "stop": ["<|end_of_text|>", "<|eot_id|>"]
19 }
20 }
```

The `model_specs` define the information of the model, as one model family usually comes with various sizes, quantization methods, and file formats. The `model_id` defines the repository of the model hub from which Xinference downloads the checkpoint files. The `prompt_style` specifies how to format prompts for this particular model. The current JSON format also supports registering custom models, as users give the aforementioned fields to Xinference.

A.3 Choose the Right Inference Engine

Table 4 summarizes the different inference engines supported by Xinference, and Table 3 is our benchmark result of these inference engines.

Table 3: Benchmark results of different inference engines when serving Llama 3 8B model. **L** is for latency and **T** is for throughput. For throughput tests, we mimic two cases: the first (C1) is 10 concurrent requests, which is a low concurrency scenario, and the second (C2) is 50 requests, which is a high concurrency scenario.

(a) NVIDIA A800 80GB on-premises cluster.

Engine	L (s)	T@C1 (token/s)	T@C2 (token/s)
PyTorch	3.56	36.69	37.10
vLLM	1.85	487.94	1276.29
SGLang	**1.51**	**627.83**	**2087.81**
llama.cpp	2.07	77.68	77.99

(b) NVIDIA A10 24GB cloud instance.

Engine	L (s)	T@C1 (token/s)	T@C2 (token/s)
PyTorch	9.16	14.56	14.72
vLLM	5.53	190.74	466.37
SGLang	**5.25**	**205.94**	**599.18**
llama.cpp	6.06	24.23	24.56

(c) Apple M3 36GB laptop.

Engine	L (s)	T@C1 (token/s)
PyTorch	19.68	6.41
MLX	15.13	8.12
llama.cpp	**9.00**	**13.81**

In terms of model support, PyTorch has the most, but as shown in Table 3, it exhibits the poorest inference performance. Regarding the model format, llama.cpp has its own unique format, and PyTorch-compatible checkpoints need to be converted into gguf or ggml. The two model formats are often quantized to lower than 8-bit. llama.cpp users may face additional burdens when getting the model, either by downloading from a model hub or by converting from PyTorch checkpoints. According to Table 3, llama.cpp is not adept at handling high concurrent requests and is more commonly used in scenarios with limited memory, such as personal computers or edge devices. vLLM and SGLang offer the strongest performance, with SGLang showing the best latency and throughput. The vLLM has a more active open-source community and supports a greater variety of models. The inference engine with precompiled packages facilitates easier installation. Otherwise, building from source often leads to compilation issues, resulting in poor usability.

Table 4: The models, model formats, hardware, and installation of different inference engines.

Engine	Models	Model Format	Hardware	Precompiled Package
PyTorch	180+	PyTorch	CPU	✓
			CUDA	✓
			ROCm	✓
vLLM	60+	PyTorch	CPU	✓
			CUDA	✓
			ROCm	
SGLang	20+	PyTorch	CUDA	✓
llama.cpp	50+	gguf ggml	CPU	✓
			CUDA	✓
			ROCm	
			Metal	
MLX	20+	mlx	Metal	✓