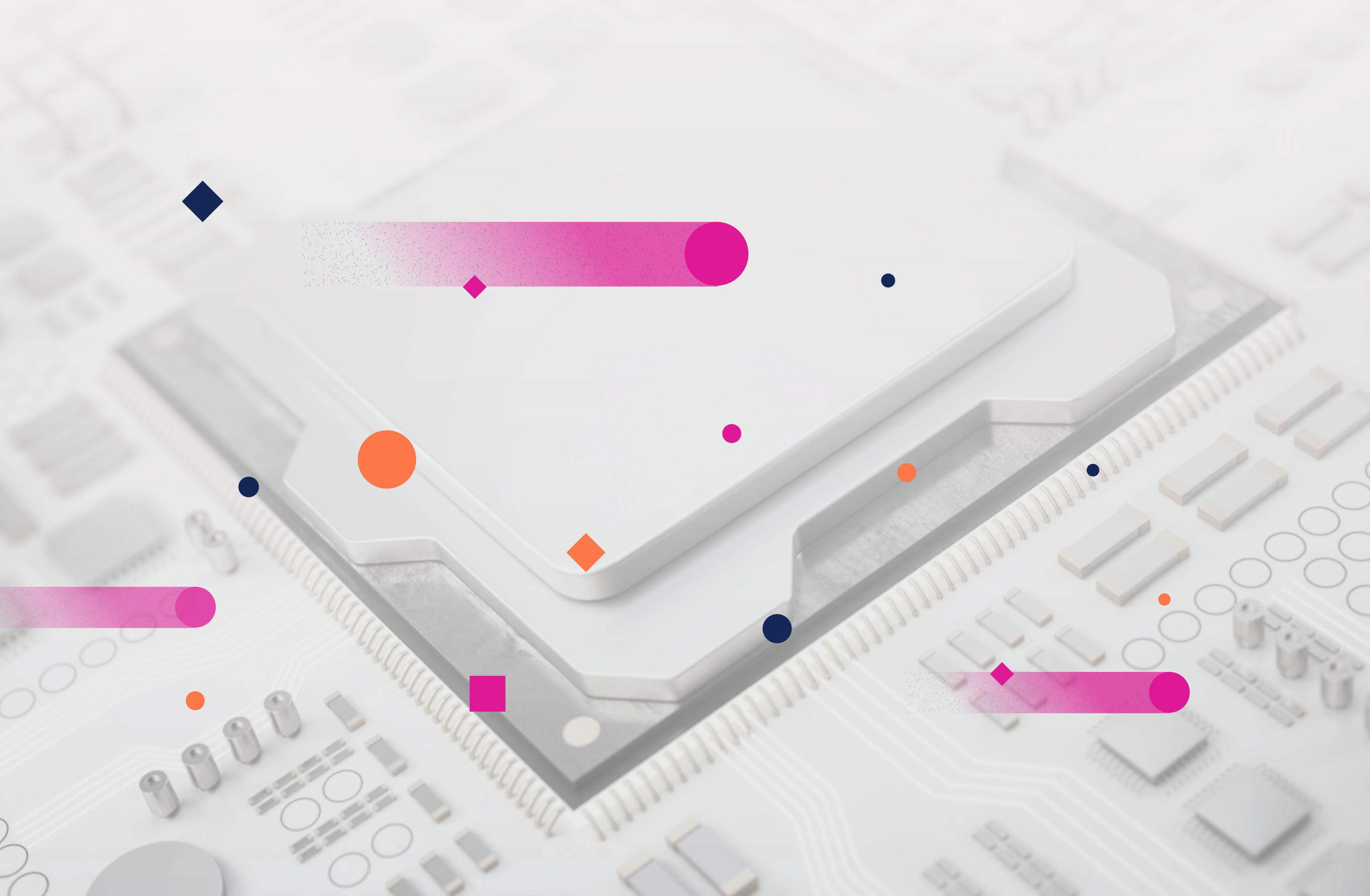
OCTOBER 2024

RESEARCH

Run:ai Model Streamer: Performance Benchmarks

Authored by:
Ekin Karabulut, Noa Neria, Omer Dayan



The deployment of large language models (LLMs) presents a critical challenge in optimizing inference efficiency, particularly due to the cold start problem—where models take substantial time to load into GPU memory, thus impacting both user experience and operational scalability. This whitepaper introduces the Run:ai Model Streamer, an open-source solution designed to mitigate these latency issues by enabling concurrent reading of model weights from the storage while directly streaming them into GPU memory. We benchmark the Run:ai Model Streamer against the default loaders of vLLM as a baseline (Safetensors Loader), as well as against Tensorizer, across different storage types, including local SSDs and Amazon S3. The experiments demonstrate that the Run:ai Model Streamer significantly reduces model loading times, thus reducing the cold start times when starting the inference engine, even in cloud-based environments, while maintaining compatibility with widely used safetensor format without the need for weight format conversion. Our results highlight the importance of storage selection independently of any model loading library and concurrent weight streaming in achieving efficient model deployment at scale, offering key insights for machine learning practitioners looking into enhancing LLM deployment performance.

Table of contents

Introduction

The increasing complexity of deploying large language models (LLMs) due to their dynamic nature in production environments has magnified the importance of efficient model loading strategies. These models, often requiring tens to hundreds of gigabytes of memory, pose significant challenges in terms of both latency and resource utilization, especially when scaling services to meet unpredictable user demand. This is where the cold start problem becomes particularly pronounced: the time it takes for a model to be loaded into GPU memory can introduce substantial delays, severely impacting both the end-user experience and the operational dynamics of machine learning systems.

Traditional approaches to loading models—sequentially transferring large tensor files from storage to CPU memory and then to GPU—is inefficient and costly, particularly in auto-scaling environments where fast scale up from idle states is critical. Therefore, a lot of corporations keep a lot of idle replicas most of the time not to degrade the user experience. But they end up paying more in compute costs. To address these challenges, the Run:ai Model Streamer was developed. This tool leverages concurrent reading of model weights from storage while streaming them to GPU, offering a marked improvement over existing methods.

In this whitepaper, we present an empirical performance analysis of the Run:ai Model Streamer, highlighting its effectiveness across different storage types (local SSDs and cloud-based S3) and in combination with the vLLM inference engine. We compare the Run:ai Model Streamer's performance against other tools, such as the Hugging Face safetensors loader and Tensorizer, providing insights into its advantages and limitations.

Methodology

Model Loading Overview

Before jumping into the overview of tools, let's have a look at what it means to load a model. Loading a machine learning model to a GPU for inference involves two main steps:

1. **Read Weights from Storage to CPU Memory**: Load the model's weights (which can be in various formats such as .pt, .h5, .safetensors, or custom formats) from storage (local, cluster-wide, or cloud) into CPU memory.
2. **Move Model to GPU**: Transfer the model's parameters and relevant tensors to GPU memory.

Notably, when loading models from cloud-based storage like S3, the process involves an additional step of loading the model to local disk as an intermediate stage before transferring it to CPU and GPU memory.

Traditionally, these steps are taking place sequentially, which makes the model loading times one of the biggest bottlenecks when scaling up

Side Note: In this whitepaper, we will use the .safetensors model format as the default due to its wide adoption in the ecosystem. However, be aware that different formats may be used in other resources.

Run:ai Model Streamer

How does it work?

The Run:ai Model Streamer is a Python SDK with a high performant C++ implementation, designed to accelerate the model loading times onto GPUs from various storage types (network file systems, S3, Disk, etc.). It achieves this by using multiple threads to read tensors concurrently from a file in object or file storage, to a dedicated buffer in the CPU memory. Each tensor is assigned an identifier, which allows the application to manage which tensors are loaded into GPU memory, enabling simultaneous tensor reading and transferring. This way the application can load tensors from the CPU memory to the GPU memory while other tensors are being read from storage to the CPU memory.

Moreover the tool takes full advantage of the fact that GPU and CPU have separate subsystems. The GPU is connected to the system via PCIe, allowing it to access CPU memory directly without needing CPU interventions. This means that CPU-side storage reads and GPU transfers happen in parallel in real time - leading to highly efficient and fast model loading across both subsystems.

What are the key features?

- **Concurrency**: The tool uses multiple threads to read model weight files in parallel, reducing storage bottlenecks and increasing GPU utilization. Multiple threads are capable of reading a single tensor in parallel.
- **Balanced Workload for Reading**: Model tensors come in different sizes. Model Streamer divides the work in a balanced way regardless of the various sizes of the models so that the storage read bandwidth can be saturated.
- **Support for Multiple Storage Types**: Compatible with various storage solutions, including local file systems (e.g., SSD) and cloud-based object stores (e.g., S3).
- **No Tensor Format Conversion**: Directly supports safetensors format, eliminating conversion overhead and ensuring fast model loading.
- **Easy Integration**: Streamer offers an iterator similar to the Safetensors's native iterator but with added benefit of concurrent reading, which is performed in the background. Its Python API simplifies integration with inference engines like vLLM and TGI while benefiting this highly performant C++ layer.

What is unique about it?

Model Streamer sets itself apart with its highly optimized C++ layer and support for concurrent tensor reads. Unlike other tools, which are designed for sequential access, the Model Streamer allows multiple threads to read from the same tensor simultaneously and streams tensors from CPU to GPU while still reading the tensors concurrently from storage to CPU. The model streaming utilizes OS-level concurrency to read data from local file systems, remote file systems, or object stores. In addition to performance, a Python wrapper provides simple APIs and easy integration into an existing codebase. It allows users to fine-tune the level of concurrency (RUNAI_STREAMER_CONCURRENCY), data chunk size for each thread (RUNAI_STREAMER_BLOCK_BYTESIZE), and CPU memory usage (RUNAI_STREAMER_MEMORY_LIMIT), making it adaptable to systems with limited resources.

Further details on setup and usage can be found in the documentation.

HuggingFace (HF) Safetensors Loader

How does it work?

The HuggingFace (HF) Safetensors Loader is an open-source utility that provides a safe and fast format for saving and loading multiple tensors. The tool uses a memory-mapped file system to avoid unnecessary data copying. For CPU operations, tensors are mapped directly into memory from the file. On the GPU, the tool creates an empty tensor with PyTorch, then moves the tensor data using cudaMemcpy, facilitating a zero-copy loading process.

What are the key features?

- **Zero-Copy Loading**: Bypasses unnecessary copies, directly mapping tensor data from storage to memory, resulting in faster read operations.
- **Support for CPU and GPU**: Allows efficient tensor loading both on CPU (through direct file mapping) and GPU (via memory-mapped loading).
- **Shared Tensors Support**: Avoids duplicating tensor data that is referenced by multiple layers in a model, leading to memory optimization.

What is unique about it?

HF Safetensors Loader's ability to perform zero-copy operations makes it significantly faster than traditional methods like pickle-based loading. It follows the traditional model loading steps (see here). Its native support for safetensors format, combined with efficient loading to both CPU and GPU, offers distinct advantages in terms of safety, speed, and memory efficiency, especially for models with shared layers.

Tensorizer

How does it work?

Tensorizer is an open-source tool developed by CoreWeave that serializes model weights and their corresponding tensors into a single file. Instead of loading an entire model into RAM before moving it to the GPU, Tensorizer streams the model data tensor by tensor from an HTTP/HTTPS or S3 source. The serialization format (model.tensors) contains all metadata at the beginning of the file, allowing for efficient, on-demand model weight loading. During this process, each read operation is handled by a Python reader, which is associated with a thread. Each thread is assigned to load specific tensors, but while each tensor is assigned to only one thread, each thread can manage multiple tensors. These threads can fetch random tensors as needed.

What are the key features?

- **Efficient Tensor Streaming:** Loads model weights tensor by tensor instead of pre-loading everything into RAM, reducing memory overhead.
- **Single File Serialization**: Encodes model weights and tensor metadata into a single file for efficient access during model loading.
- **Threaded Reading**: Tensorizer allows multiple Python readers (threads), where each reader fetches tensors independently. Although only one thread is responsible for reading any particular tensor, each thread can process multiple tensors, providing a level of parallelism.
- **HTTP/S3 Endpoint Support**: Supports loading models over the web or cloud storage, making it flexible for cloud-based deployments.
- **Parallel Streaming Encryption/Decryption and Authentication:** Supports encryption-at-rest and cryptographic authentication of model weights during streaming, adding security alongside standard TLS-based encryption-in-transit.

What is unique about it?

Tensorizer's ability to stream model weights directly from HTTP/S3 endpoints allows it to handle large models without overwhelming system RAM. By assigning threads to individual tensors, it achieves a degree of parallelism in model loading, though each tensor can only be read by one thread. Additionally, Tensorizer provides encryption-at-rest and cryptographic verification of model weights, ensuring secure and efficient handling of models across different environments.

Where Loading Meets Inference Engines: Loading Weights with vLLM

Model serving is not complete without an inference engine. There are many inference engines and servers that one can utilize. If you are new to the topic, we recommend checking out our previous blog about introduction to inference engines.

In our previous benchmarking, vLLM showed remarkable performance. Therefore, we will consider vLLM and its model loading offerings in this whitepaper. The vLLM framework uses the HF safetensors model loading as default. Additionally, it supports Tensorizer by CoreWeave to load models from S3 endpoints. However, note that the Tensorizer library requires converting weights from safetensors format to tensorizer format.

For more information about inference engines, please refer to the previous comparison blog.

Side Note: For this benchmarking study, we only utilized vLLM. However, in the future, we are planning to benchmark different inference engines as well.

Experiment Setup

This section details the technical configuration and experiment design used to evaluate the performance of different model loading techniques across various storage types.

Technical Configuration

The experiments were conducted using the following setup:

- **Model:** Meta-Llama-3-8B, a large-scale language model weighing 15 GB, stored in a single Safetensors format.
- **Hardware:** AWS g5.12xlarge instance featuring 4 NVIDIA A10G GPUs (only one GPU was used for all tests to maintain consistency).
- **Software Stack:**
  - CUDA 12.4
  - vLLM 0.5.5 (Transformers 4.44.2)
  - Run:ai Model Streamer 0.6.0
  - Tensorizer 2.9.0
  - Transformers 4.45.0.dev0
  - Accelerate 0.34.2

For the experiments involving Tensorizer, the same model was serialized into Tensorizer's proprietary tensor format using the recipe provided by the Tensorizer framework.

Storage Types

To assess the loaders' performance under different storage conditions, we conducted experiments using three distinct storage configurations:

- **Local SSDs (GP3 and IO2 SSDs):** High-performance local storage types with different IOPS and throughput limits.
  - **GP3 SSD**
    - Capacity: 750 GB
    - IOPS: 16,000
    - Throughput: 1,000 MiB/s
  - **IO2 SSD**
    - Capacity: 500 GB
    - IOPS: 100,000
    - Throughput: up to 4,000 MiB/s
- **Amazon S3:** A cloud-based storage option where the latency and bandwidth constraints of the cloud environment were expected to affect performance. We used S3 buckets located in the same AWS region as the instance to minimize inter-region latency.

For pricing details and additional information on AWS storage types, please refer to the AWS page.

Experiment Design

The experiments were structured to compare the performance of different model loaders (Run:ai Model Streamer, Tensorizer, and HuggingFace Safetensors Loader) across the three storage types:

Experiment #1: GP3 SSD

We measured model loading times using different loaders on the GP3 SSD configuration. (Results as Tables in Appendix A.)

Experiment #2: IO2 SSD

The same loaders were tested on IO2 SSD to evaluate the impact of higher IOPS and throughput. (Results as Tables in Appendix B.)

Experiment #3: Amazon S3

This experiment focused on comparing loaders in a cloud storage scenario. Safetensors Loader was excluded as it does not support S3. (Results as Tables in Appendix C.)

Experiment #4: vLLM with different loaders[1]

We integrated Run:ai Model Streamer into vLLM to measure the complete time required to load the model for all the storage types above and make it ready for inference. We compare it with the default loader of vLLM (HF Safetensors Loader) for SSD experiments and Tensorizer integration of vLLM for S3 experiments (Safetensors Loader does not support loading from S3). This experiment allowed us to test the overall impact of the loaders on end-to-end model serving times. (Results as Tables in Appendix D.)

Each experiment was conducted under cold-start conditions to ensure consistency and eliminate the effects of cached data. For the cloud-based Amazon S3 tests, at least two-minute wait between tests was introduced to avoid any caching effects on AWS side and maintain accuracy in the results.

Specifically for Tensorizer experiments, we serialized the same model following the Tensorizer recipe to the required tensor format[2]. For the benchmarking experiments for standalone Tensorizer, the benchmarking recipe in their repository is utilized. We performed these experiments without the optional hashing.

[1]For vLLM experiments, we used eager mode and disabled swap-size due to the fact that we are not implementing any beam-search.
[2]During our research, we learned that serialization of the weights for the vLLM engine using Tensorizer should be done via the provided serialization script in the vLLM repository . Therefore, for experiments, where the Tensorizer is coupled with a vLLM engine, we used the vLLM script to serialize the model weights. For the standalone loader experiments, we followed the serialization recipe in the Tensorizer repository. Further performance investigation of these scripts are not in the scope of this whitepaper.

Experiment Results

Experiment #1: GP3 SSD

In this initial experiment, we compared the loading performance of different model loaders using GP3 SSD storage. First, we evaluated the impact of concurrency on the performance of the Run:ai Model Streamer (see Figure 1) and examined how the number of workers affected Tensorizer (see Appendix A). For Run:ai Model Streamer, increasing the concurrency—the number of concurrent threads reading from storage into CPU memory—led to a notable decrease in model loading time.

At concurrency 1, Run:ai Model Streamer loaded the model in 47.56 seconds, slightly slower than HuggingFace Safetensors Loader at 47.99 seconds. However, as we increased concurrency, we observed significant improvements. With a concurrency of 16, the loading time dropped to 14.34 seconds, while maintaining a steady throughput of approximately 1 GiB/s, which is the maximum throughput of GP3 SSD. Beyond this concurrency level, performance gains were constrained by the storage's throughput limit.

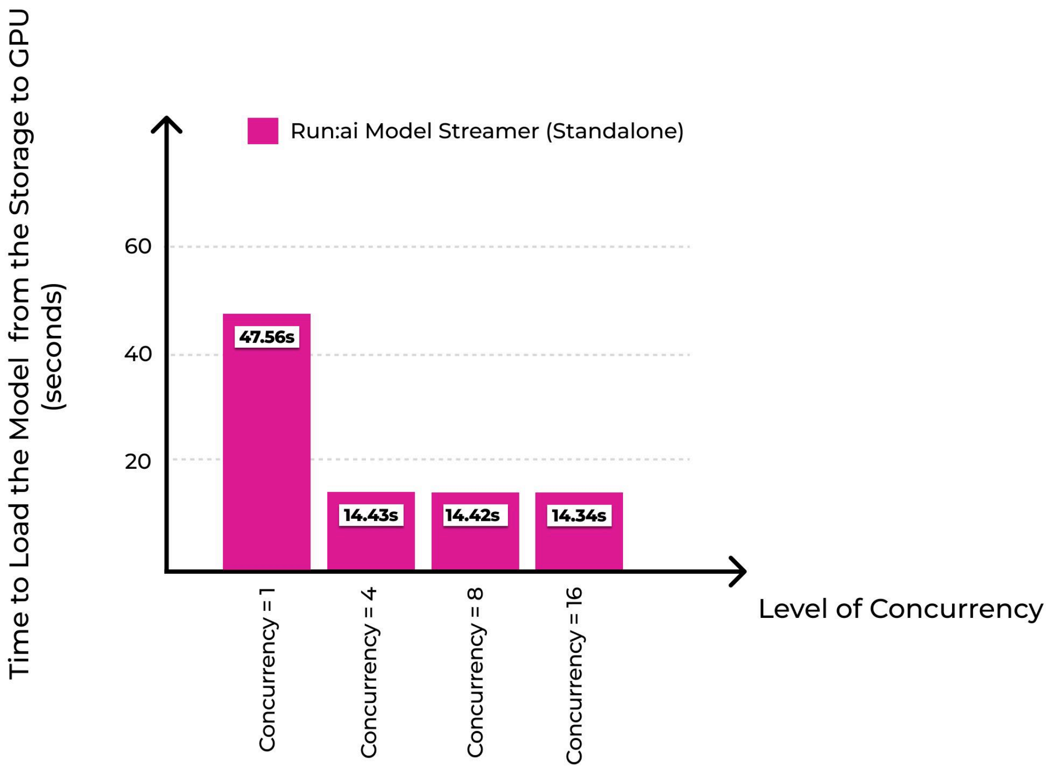


Figure 1: The Effect of Concurrency on Model Loading Performance with Run:ai Model Streamer on GP3 SSD. This figure shows the impact of different concurrency levels (1, 4, 8, and 16) on model loading time using the Run:ai Model Streamer. As concurrency increases, load times decrease significantly, dropping from 47.56 seconds (at concurrency 1) to 14.34 seconds (at concurrency 16). At this point, the streamer achieves the maximum possible throughput of 1 GiB/s, which is the limit of the GP3 SSD.

Tensorizer showed similar scaling behavior. With a single worker, the loading time was close to that of the Safetensors Loader at 50.74 seconds. As the number of workers increased, Tensorizer reached its best performance with 16 workers, achieving a loading time of 16.11 seconds and a throughput of 984.4 MiB/s, again nearing the maximum bandwidth of GP3 SSD.

The storage throughput limit of GP3 SSD was the bottleneck for both Run:ai Model Streamer and Tensorizer, limiting further performance improvements. This led us to test a higher throughput storage solution in Experiment #2.

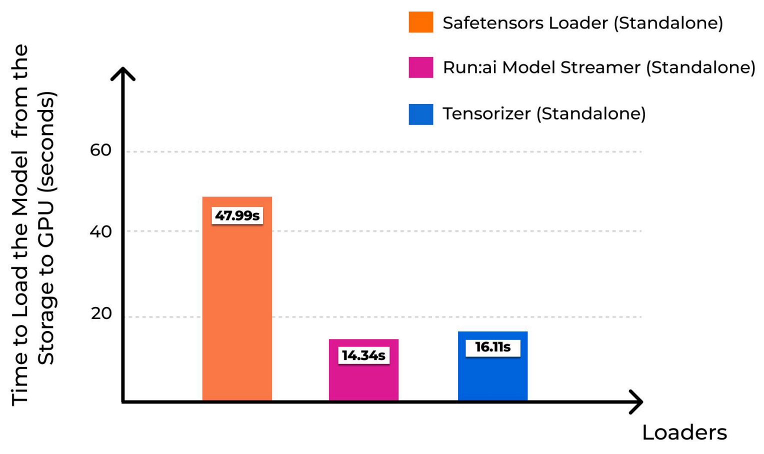


Figure 2: Model Loading Performance on AWS GP3 SSD with Safetensors Loader, Run:ai Model Streamer, and Tensorizer. This figure compares the model loading times of Safetensors Loader, Run:ai Model Streamer, and Tensorizer on AWS GP3 SSD. The best observed performance for each loader is shown. For the Run:ai Model Streamer, the optimal result was achieved with a concurrency level of 16. For Tensorizer, the best performance was recorded using 16 workers as well (see Appendix A).

Experiment #2: IO2 SSD

For the second experiment, we used IO2 SSD, which offers significantly higher throughput than GP3 SSD. As before, we analyzed the effect of concurrency on Run:ai Model Streamer (see Figure 3) and the number of workers on Tensorizer (see Appendix B).

At concurrency 1, Run:ai Model Streamer and HuggingFace Safetensors Loader showed similar loading times of 43.71 seconds and 47 seconds, respectively. However, as we increased concurrency, Run:ai Model Streamer showed much more pronounced gains compared to GP3 SSD. With concurrency 8, the model was loaded in just 7.53 seconds, making it around 6x faster than the HuggingFace Safetensors Loader, which took 47 seconds.

For Tensorizer, the performance also improved significantly. The optimal result was observed with 8 workers, achieving a model loading time of 10.36 seconds (see Figure 4 and Appendix B). Beyond that, adding more workers did not yield further performance improvements, likely due to storage throughput limitations.

Despite the theoretical maximum throughput of 4 GiB/s for IO2 SSD, our experiments consistently hit a ceiling at around 2 GiB/s with Run:ai Streamer and 1.6 GiB/s with Tensorizer. This suggests practical throughput limitations on the AWS infrastructure, rather than the loaders themselves.

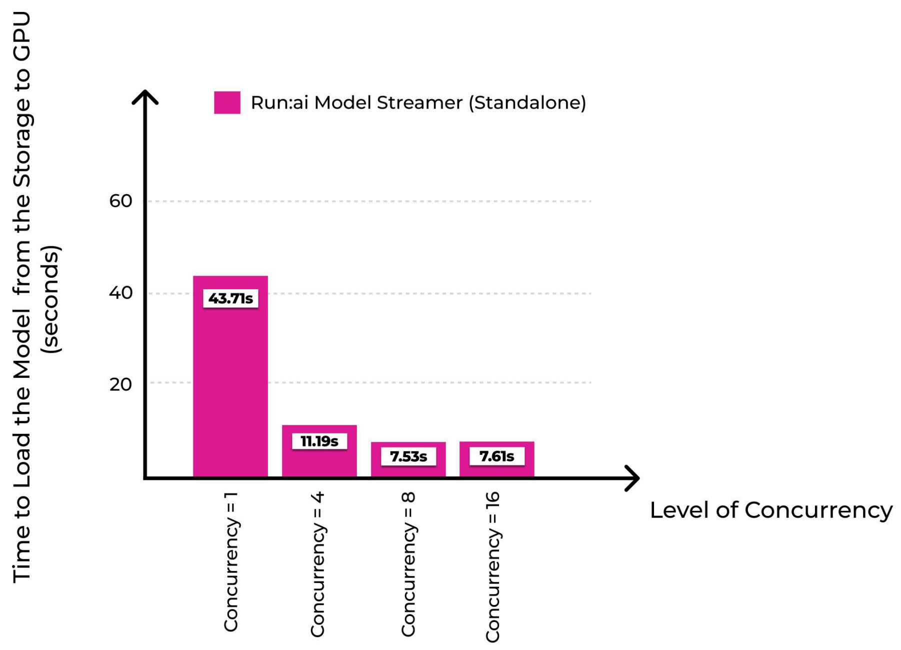


Figure 3: The Effect of Concurrency on Model Loading Performance with Run:ai Model Streamer on IO2 SSD. This figure shows the impact of different concurrency levels (1, 4, 8, and 16) on model loading time using the Run:ai Model Streamer. As concurrency increases, load times decrease significantly, dropping from 43.71 seconds (at concurrency 1) to 7.53 seconds (at concurrency 8).

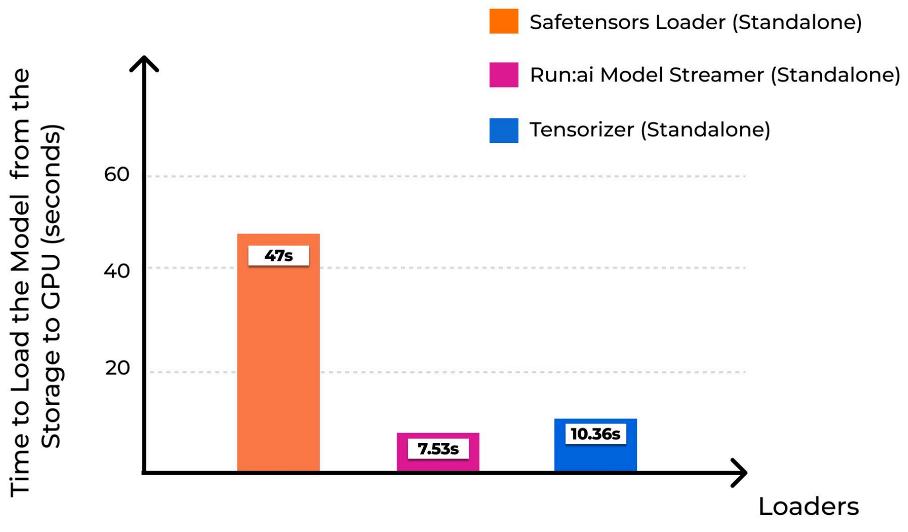


Figure 4: Model Loading Performance on AWS IO2 SSD with Safetensors Loader, Run:ai Model Streamer, and Tensorizer. This figure compares the model loading times of Safetensors Loader, Run:ai Model Streamer, and Tensorizer on AWS IO2 SSD. The best observed performance for each loader is shown. For the Run:ai Model Streamer, the optimal result was achieved with a concurrency level of 8. For Tensorizer, the best performance is recorded using 8 workers as well (see Appendix B).

Cloud Storage Experiments (S3)

Experiment #3: S3 Bucket

In this experiment, we compared the performance of Run:ai Model Streamer and Tensorizer using Amazon S3 as the storage medium. Since HuggingFace Safetensors Loader does not support S3, it was not included in this benchmarking experiment. For the Tensorizer experiments, we used different numbers of workers and chose the best result for Figure 6, which was achieved with 16 workers in this case (see Appendix C).
The results showed that Run:ai Model Streamer outperformed Tensorizer at all tested concurrency levels. At concurrency 4, Run:ai Model Streamer loaded the model in 28.24 seconds. As concurrency increased, Run:ai Model Streamer continued to improve, reaching a load time of 4.88 seconds at concurrency 32, compared to 37.36 seconds for Tensorizer's best result with 16 workers. This demonstrates Run:ai Model Streamer's superior efficiency in loading from cloud-based storage.

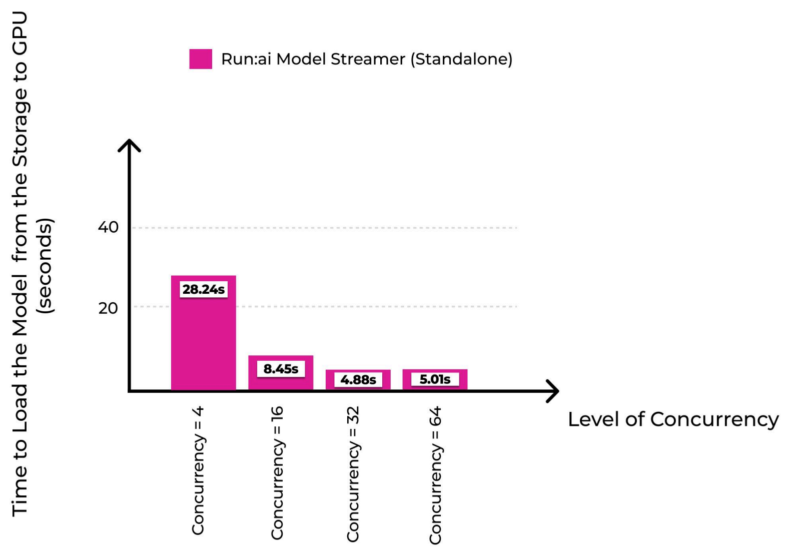


Figure 5: The Effect of Concurrency on Model Loading Performance with Run:ai Model Streamer on S3 Bucket. This figure shows the impact of different concurrency levels (4, 16, 32 and 64) on model loading time using the Run:ai Model Streamer. As concurrency increases, load times decrease significantly, dropping from 28.24 seconds (at concurrency 4) to 4.88 seconds (at concurrency 32).

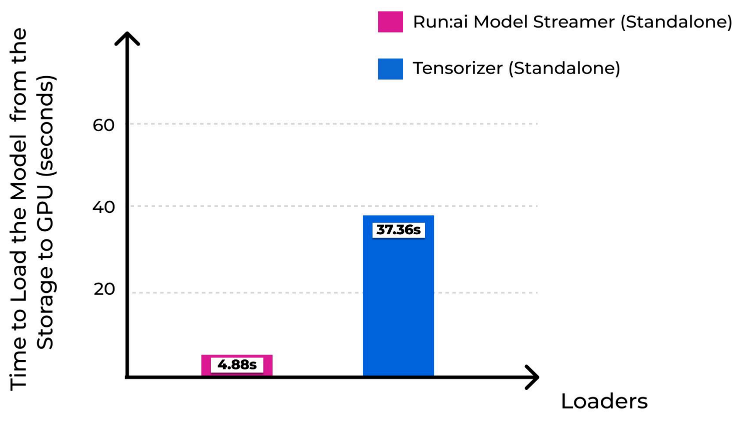


Figure 6: Model Loading Performance from AWS S3 with Run:ai Model Streamer, and Tensorizer. This figure compares the model loading times of Run:ai Model Streamer and Tensorizer from S3 bucket. The best observed performance for each loader is shown. For the Run:ai Model Streamer, the optimal result was achieved with a concurrency level of 32 (4.88 seconds), while for Tensorizer, the best performance was recorded using 16 workers (37.36 seconds).

Side note: During our experiments, we observed unexpected caching behavior on AWS S3. When experiments were repeated in quick succession, the model load times significantly improved, likely due to some form of S3 caching mechanism. To ensure consistency and avoid benefiting from this "warm cache," we introduced at least a 3-minute wait between each test run. The results presented here reflect the times recorded after these intervals, ensuring they represent cold-start conditions.

Experiment #4: vLLM with All Loaders

In this experiment, we integrated the different model loaders into vLLM to assess the total time required for the model to be ready for inference when an inference engine is used. This includes the entire process from loading the model to the point where it can handle user requests. While all loaders Run:ai Model Streamer, HuggingFace Safetensors Loader and Tensorizer were tested with local storage (GP3 SSD and IO2 SSD), HuggingFace Safetensors was excluded from S3 storage configurations since it does not support loading from S3 bucket. We tested Tensorizer for S3 storage together with vLLM and compared it with Run:ai Model Streamer.

For each vLLM + Run:ai Model Streamer experiment, we used the most optimal concurrency levels determined from earlier experiments. Specifically:

- For GP3 SSD, a concurrency level of 16 was used (see Figure 1).
- For IO2 SSD, the concurrency level was also 8 (see Figure 3).
- For S3 storage, a higher concurrency level of 32 was employed (see Figure 5).

These optimal concurrency levels were selected based on the findings from our standalone loader experiments (such as in Experiment #1 with GP3 SSD), where different concurrency levels were tested to determine the most efficient configuration for Run:ai Model Streamer. Once the best-performing concurrency levels were identified, they were applied in the corresponding vLLM experiments to ensure the loaders performed optimally within vLLM.

Similarly, for the Tensorizer + vLLM integration, we used the most optimal number of workers determined in previous experiments. Specifically:

- For GP3 SSD, 16 workers were used (see Appendix A)
- For IO2 SSD, 8 workers were used (see Appendix B)
- For S3, 16 workers were used (see Appendix C)

For GP3 SSD, Run:ai Model Streamer required 35.08 seconds in total, while HuggingFace Safetensors Loader took 66.13 seconds. Similarly, with IO2 SSD, Run:ai Model Streamer reduced the time to 28.28 seconds, while HuggingFace Safetensors Loader required 62.69 seconds. The same trend was observed with Tensorizer as well. For GP3 SSD experiments, Tensorizer took 36.19 seconds while this number lowered down to 30.88 with IO2 SSD.
In both cases, Tensorizer and Run:ai Model Streamer cut down the readiness time of the engine by around half in comparison to Safetensors Loader.

For the S3 storage configuration, Run:ai Model Streamer achieved a total readiness time of 23.18 seconds, while Tensorizer took 65.18 seconds to achieve the same outcome.

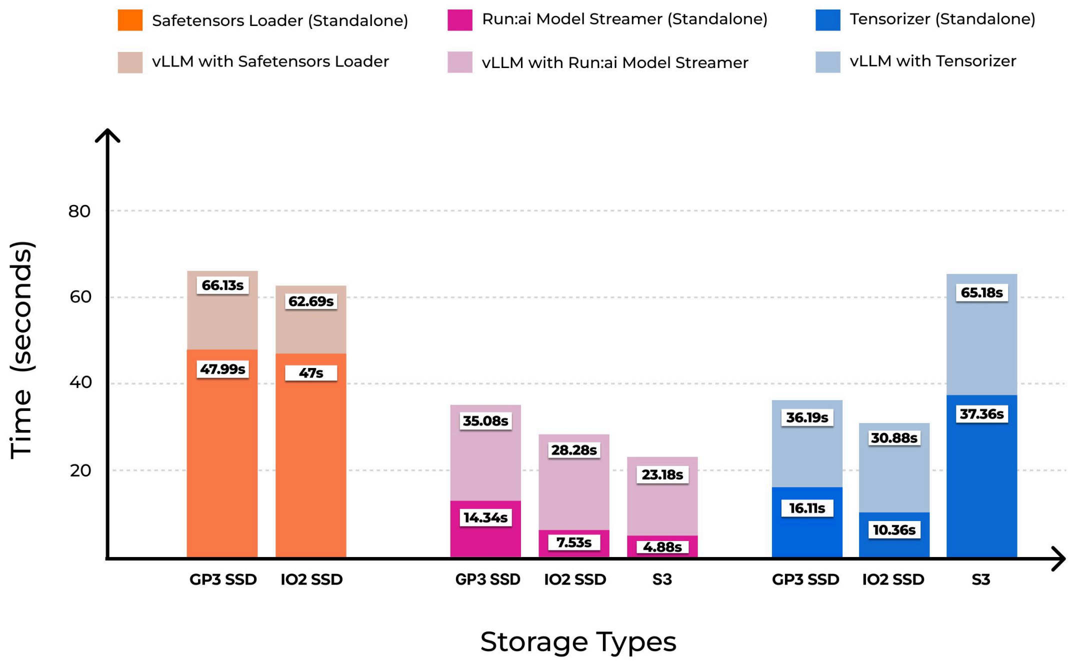


Figure 7: This figure presents the total time required for the vLLM engine to be ready for inference across different storage types (GP3 SSD, IO2 SSD, and S3) when using Run:ai Model Streamer, HuggingFace Safetensors Loader, and Tensorizer. The dark-colored bars show the time it takes to load the model from storage to GPU while the light-colored bars show the total time for the vLLM engine to load and get ready to serve requests (time to load the model plus the time to warm the inference engine up). For local storage options (GP3 and IO2 SSD), the Run:ai Model Streamer and Tensorizer consistently outperformed the Safetensors Loader, cutting readiness times nearly in half. On S3, both Run:ai Model Streamer and Tensorizer were tested, with Run:ai Model Streamer delivering significantly faster readiness times.

Discussion & Conclusion

Run:ai Model Streamer: Consistent High Performance Across Storage Types

In each experiment, Run:ai Model Streamer demonstrated a significant performance advantage across all storage solutions. On GP3 SSD and IO2 SSD, we observed that the Run:ai Model Streamer reached its best performance at concurrency 16 and 8 respectively, where it consistently approached the maximum throughput limits of the storage devices—1 GiB/s for GP3 and 2 GiB/s for IO2—dramatically reducing model load times in comparison to HuggingFace Safetensors Loader. This was particularly evident in Experiment #2, where Run:ai Model Streamer loaded the model in just 7.53 seconds on IO2 SSD, around 6x reduction in load time compared to HuggingFace Safetensors Loader.

When moving to Experiment #3 on Amazon S3, the Run:ai Model Streamer achieved 4.88 seconds at concurrency 32, vastly outperforming Tensorizer, which took 37.36 seconds with its best configuration of 16 workers. This is due to the fact that Run:ai Model Streamer creates an AWS S3 client per thread, with each thread sending multiple asynchronous requests to the S3 backend. As a result, the effective concurrency level is much higher than just the thread count. For example here, with a configured concurrency of 32, the effective concurrency level is multiplied by the AWS max concurrency limit (default is 50), resulting in an effective concurrency level 50 times larger. The 7.6x performance difference highlighted not only the Streamer's superior efficiency in handling cloud-based storage but also its ability to scale with utilizing concurrency fully, a key factor in minimizing latency in large-scale deployments.

Evenly Distributed Workload Across Threads in S3 Environments

One of the standout features of the Run:ai Model Streamer that we also proved with Experiment #3 is its ability to evenly distribute the workload across all threads, independent of the specific model or the size of the tensors being loaded. The streamer is designed in layers, where the storage layer reads the file in equal blocks. This way, the Streamer ensures that all threads are fully utilized, leading to better scalability and higher throughput at higher concurrency levels as we can see in S3 experiment results (see Experiment #3).

Concurrency and Storage Throughput as Key Performance Drivers

One of the most striking findings from the experiments is the critical role that concurrency and storage throughput play in determining model loading performance. In Experiment #1 (GP3 SSD), increasing concurrency significantly improved loading times for Run:ai Model Streamer until the storage throughput became the limiting factor. This is consistent with the Streamer's design, which leverages multi-threading to maximize data throughput from storage to the GPU. For GP3 SSD and IO2 SSD, concurrency 16 and 8 proved to be the sweet spot respectively, allowing the streamer to achieve maximum throughput without being bottlenecked by the storage bandwidth. Beyond this point, further increases in concurrency did not result in substantial improvements, as the storage bandwidth had already been saturated. The GP3 SSD's throughput limit of 1 GiB/s is clearly reflected in the data, where the performance plateaued at higher concurrency levels. This is an important observation, as it demonstrates that loading efficiency is ultimately constrained by the storage medium's ability to provide data.

vLLM Integration and 2x Faster Time to Model Readiness for End Users

Our final experiment, Experiment #4, which integrated these loaders with vLLM, further supports the versatility of the Run:ai Model Streamer. In both local storage configurations—GP3 SSD and IO2 SSD—Run:ai Model Streamer outperformed the HuggingFace Safetensors Loader by a significant margin. On GP3 SSD, Run:ai Model Streamer required only 35.08 seconds to prepare the model for inference, compared to 66.13 seconds for Safetensors Loader. The same pattern held for IO2 SSD, with Run:ai Model Streamer reducing readiness time to 28.28 seconds, a stark contrast to the 62.69 seconds needed by the default loader.

In the cloud-based scenario of S3 storage, Run:ai Model Streamer showcased its efficiency by achieving readiness in 23.18 seconds, while Tensorizer required more double that time (65.18 seconds). The high performance of Run:ai Model Streamer under both local and cloud storage conditions underlines its broad applicability in various machine learning deployment contexts and setups.

Future Work

As we conclude this benchmarking study, several topics present themselves for future exploration and refinement of our findings. The following areas stand out as promising directions for advancing our understanding of the Run:ai Model Streamer's performance when it comes to serving large language models (LLMs):

Expanding beyond vLLM: While our benchmarking focuses on vLLM, other high-performance inference engines, such as Hugging Face's Text Generation Inference (TGI) can be evaluated for integration with the Run:ai Model Streamer. Different engines may have different optimizations that could further reduce loading times or reveal new bottlenecks.

Multi-GPU Model Parallelism: While we used a single GPU in our experiments, a deeper exploration into multi-GPU inference use cases could offer a more comprehensive view.

Testing on Kubernetes Clusters for Auto-scaling purposes: A further investigation on Kubernetes-based environments can give a better overview on the performance improvement of the Run:ai Model Streamer in comparison to other loaders for autoscaling use cases of LLMs. Evaluation on real production environments with unpredictable workloads can give more insights on its impact under load spikes, failover scenarios, and auto-scaling events.

Investigating Faster Storage Solutions: Given the speed limitations of even high-end SSDs like IO2, exploring the impact of integrating faster storage technologies for ultra-fast data transfers can be investigated further for reducing cold start times even further. We also expect to see highly optimized results for reading from storage types that are designed for multiple clients, such as distributed storage types (e.g. NAS and object stores) due to the design of Run:ai Model Streamer.

About Run:ai

Run:ai is revolutionizing the AI infrastructure landscape with its platform, designed to optimize the efficiency, scalability, and accessibility of AI and machine learning operations. By addressing the challenges of AI infrastructure, Run:ai empowers enterprises to accelerate their AI initiatives and foster innovation.

To learn more about Run.ai visit:
www.run.ai

Appendix A

Experiment #1: GP3 SSD Results as Table

Run:ai Model Streamer		HuggingFace Safetensors Loader
Concurrency	**Time to Load the Model to GPU (s)**	**Time to Load the Model to GPU (s)**
1	47.56	47.99
4	14.43	
8	14.42	
16	14.34	

Tensorizer	
Number of Readers	**Time to Load the Model to GPU (s)**
1	50.74
4	17.38
8	16.49
16	16.11
32	17.18
64	16.44
100	16.81

Appendix B

Experiment #2: IO2 SSD Results as Table

Run:ai Model Streamer		HuggingFace Safetensors Loader
Concurrency	**Time to Load the Model to GPU (s)**	**Time to Load the Model to GPU (s)**
1	43.71	47
4	11.19	
8	7.53	
16	7.61	
20	7.62	

Tensorizer	
Number of Readers	**Time to Load the Model to GPU (s)**
1	43.85
4	14.44
8	10.36
16	10.61
32	10.95

Appendix C

Experiment #3: S3 Bucket Results as Table

Run:ai Model Streamer	
Concurrency	**Time to Load the Model to GPU (s)**
4	28.24
16	8.45
32	4.88
64	5.01

Tensorizer	
Number of Readers	**Time to Load the Model to GPU (s)**
8	86.05
16	37.36
32	48.67
64	41.49
80	41.43

Appendix D

Experiment #4: vLLM Results as Table

For GP3 SSD Storage

vLLM with Different Loaders	
Loader	**Total time until vLLM engine is ready for requests (s)**
Safetensors Loader	66.13
Run:ai Model Streamer	35.08
Tensorizer	36.19

For IO2 SSD Storage

vLLM with Different Loaders	
Loader	**Total time until vLLM engine is ready for requests (s)**
Safetensors Loader	62.69
Run:ai Model Streamer	28.28
Tensorizer	30.88

For S3 Storage

vLLM with Different Loaders	
Loader	**Total time until vLLM engine is ready for requests (s)**
Run:ai Model Streamer	23.18
Tensorizer	65.18