SERIF: A Scalable, Extendable Real-Time Inference Framework

A PROJECT
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Daniel Frink

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Abhishek Chandra

Jan, 2025

Acknowledgements

I sincerely appreciate all the mentorship and support that I have received during my time in graduate school. I especially want to thank Joel Wolfrath, who has been a great mentor in my budding professional career, and has provided support for me throughout this project. I am also grateful to Professor Chandra, who was willing to support my Plan B project on short notice, and to Professors Jon Weissman and Ulya Karpuzcu for serving on my committee. Finally, thank you to my parents and brothers, who have been instrumental as pillars and role models throughout my time in graduate school.

Abstract

Recent advancements in machine learning models for a wide array of tasks have raised important questions about the most efficient and accurate ways to serve requests for systems which provide access to these models. One recent approach to this challenge is SneakPeek, a paper that explores methods that incorporate data awareness for scheduling within time-sensitive inference serving systems. SERIF is an extended implementation of the real-time serving framework used to evaluate the novel data-aware scheduling approaches proposed in SneakPeek. SERIF can be adapted to any arbitrary distributed infrastructure, model type, and application type through a well-documented configuration process. SERIF also extends upon the concept of "data-aware" scheduling by providing a per-node shared-memory cache implementation that is shown to decrease overall system latency with data intensive applications. This paper covers background in the field of inference-serving, implementation details of the SERIF framework, and an evaluation of SERIF across different infrastructure types, scheduler implementations, and input data loads.

Contents

Chapter 1

Introduction

In recent years, artificial intelligence (AI) has taken over as the hottest and fastest growing subsection of the computer science domain. It is beginning to venture into increasingly complex spaces, and the most recent models have begun to pass long-standing benchmarks for difficult tasks such as ARC-AGI [1], competitive mathematics and programming [2], and the protein folding problem[3]. Along with the growing demand for AI across a wide variety of domains, the number of adjacent challenges that need to be solved to deliver products that work for businesses is also increasing. Some of those challenges are about regulation and ethics, some surround monetization, and some face the technical challenges of building usable systems for customers. The problem that this project focuses on falls into the third category; specifically, the problem of efficiently and accurately serving inference requests in systems with complex infrastructure and numerous types of data and models to choose from.

One recent paper that has approached this problem is SneakPeek[4]. The novel contribution of SneakPeek is the demonstration of methods that improve the performance of standard scheduling algorithms by providing ”data-awareness” to the scheduler. This involves running the input data through a lightweight classifier that can help the scheduler prioritize requests more efficiently. Using a data-aware scheduling approach in a real-time serving environment, the authors were able to increase utility by 62-90% depending on the type of scheduler, models and data used.

SERIF [5] is an extended implementation of the framework used to evaluate SneakPeek.

This extension provides the ability for future work in this area to easily test new schedulers, models, and input data types within an infrastructure that can be configured and scaled at will by the user. The paper is structured as follows;

- Chapter 2 provides a comprehensive summary of the current state of inference-serving research, with a specific deep dive into the contributions of SneakPeek, to provide context for the necessity and requirements of a system that can enable future research in the field of real-time inference serving.
- Chapters 3-5 provides details on the architecture, implementation, and evaluation of SERIF.
- Chapter 6 discusses the outcome of this work along with future directions for work that could be accomplished with the use of and continued extension of SERIF.

Chapter 2

Inference Serving Background

2.1 Origins of Inference Serving Research

The concept of inference serving as a distinct research focus has its roots in the mid-2010's, with its first applications coming in the field of optimizing advertisements for social media applications.

In 2014, researchers at LinkedIn published LASER [6], a platform built to improve advertisement prediction. LASER covers many different aspects of the ML life-cycle, including model training, deployment, serving, and configuration. With regard to model serving, LASER focused mainly on the importance of latency in the system. They found that within the context of production advertising systems, optimizing for prediction latency was generally more valuable than optimizing for prediction accuracy, and developed an approach using caching and lazy evaluation to achieve that goal.

In 2015, researchers at Berkeley published Velox [7], a platform for managing machine learning deployments for large-scale data analytics. Similarly to LASER, it identified serving as a distinct challenge within the ML deployment cycle; however, it also mainly used a cache-based approach to speed up request serving rather than developing novel scheduling or scaling algorithms.

The first notable paper that specifically focused on inference serving was Clipper [8]. The authors of this paper identified three main objectives for which inference-serving systems should optimize: throughput, latency, and accuracy. Most of the research in the inference serving space to date focuses on optimizing for one or more of these objectives.

To accomplish this goal, Clipper included several foundational inference serving optimization techniques that have been continually iterated upon in future papers, such as dynamic request batching, prediction caching, and bandit/ensemble methods for model selection. In addition, Clipper was designed with a layered architecture that allowed for simple user interaction experience - which was in stark contrast with previous inference serving solutions.

2.2 Foundational Approaches

2.2.1 Dynamic Batching

Dynamic batching is a technique that is used to take advantage of the parallelization that modern ML accelerators provide. In short, the idea is that requests can be sent in parallel to the underlying hardware accelerator rather than being executed in a serialized fashion. Although waiting to execute requests in a batch may sacrifice a small amount of latency for the first requests, the total throughput of the system will significantly increase by utilizing batched inference.

As discussed previously, Clipper was the first paper to present a novel batching strategy in the context of inference serving. They identified the latency-throughput trade-off and framed their approach in the context of minimizing the amount of SLA timeouts while maximizing the throughput of the system. To accomplish this, they introduced two core ideas: dynamic batch sizing and delayed batching. Dynamic batch sizing allows the system to increase the batch size (which increases throughput but also increases latency) up to the point at which SLA timeouts are happening, at which point the system can back off. Delayed batching allows the system to intelligently wait to process a batch if there are few incoming requests in the batch, as it may be able to process more requests that arrive during the extended waiting period in the same batch. This optimization can significantly increase throughput for systems with uneven or bursty workloads.

While Clipper's strategies are relevant across a wide variety of systems, different types of serving environments have more specific considerations to take into account. TensorFlow Serving [9] is one of the leading open-source production inference serving solutions. One novel contribution presented in the 2017 paper is the concept of "inter-request" merging - which increases performance by concatenating underlying input tensors. Merging inputs into single tensors rather than just grouping requests enables more efficient hardware utilization through optimized memory allocation and kernel launches. This type of optimization is only possible in an environment where the structure of the incoming requests and underlying models is homogeneous, which may be more common in production use cases for performance or cost purposes.

In recent years, increased capabilities in the cloud computing space have enabled a new paradigm, called serverless computing. Serverless systems have two main attributes: compute resources are managed by the cloud provider, and containers for code execution are stateless. This model allows customers to very quickly scale their compute resources up or down, and allows the customer to only pay for compute they actually use, however, it also introduces some complications with respect to batching in serverless inference serving systems. BATCH [10] addresses these challenges by factoring these constraints into batching approaches. Two of the main constraints are cold start latency, that is, time spent initializing a new model, and resource allocation limits, which are generally managed very tightly in order to decrease cost for the customer. By implementing techniques such as prestarting new containers and managing factors such as batch sizing, timeouts, and memory allocation, BATCH showed for the first time that efficient dynamic batching for inference serving systems can be achieved in a stateless environment.

2.2.2 Resource Management

Another technique that has been researched to improve inference throughput and latency is resource management. In general, resource management refers to taking actions that, in some way, scale the compute capabilities of the system. The most intuitive approach to improving performance through resource management is to increase the amount of compute resources available to the system. In the context of inference-serving systems, there are two main methods for doing this: horizontal and vertical scaling. Horizontal scaling refers to techniques that adjust the number of different executors available, while vertical scaling refers to techniques that adjust the power of existing executors. Both of these approaches increase the overall cost of running the system, so care must be taken when choosing to scale resources in both directions. There are many papers that propose "auto-scaling" algorithms that automatically manage the compute/cost tradeoff with different methods and in different environments. Clipper proposes baseline, general purpose autoscaling techniques, Swayam [11] discusses efficient auto-scaling in the context of highly distributed systems, and FA2 [12] presents a graph-based autoscaler for deep learning systems.

There are also cases where optimizations can be made to better utilize system resources without needing to increase total compute power. An example of this is InferLine [13], which studies inference serving performance in pipelined systems. In a pipelined system, a single inference request may be processed by multiple different models, which can be placed on several different types of hardware accelerators. It uses a low-frequency planner to periodically re-configure model placement based on the general profile of incoming request over long periods of time, along with a high-frequency tuner that can react to bursty workloads similarly to the previously discussed auto-scaling and dynamic batching methods.

2.2.3 Scheduling Policies

The third fundamental type of optimization that has been studied in the context of inference serving is scheduling policy. A scheduling policy defines the rules for when and how requests are executed, including decisions about batching, hardware placement, and request prioritization, with the end goal of meeting latency, throughput, and cost requirements for the system.

One approach that researchers have studied as a scheduling objective is GPU resource scheduling. GPUs are costly for organizations to buy or use in a cloud environment, which means optimizing utilization and fairness is necessary for modern workloads. Nexus [14] proposes novel scheduling techniques that aim to maximize GPU utilization within a system, such as "squishy bin packing", which takes batch sizing on different executors into account when scheduling requests, and partial network batching, which enables similar DNNs to be batched together on the same GPU if they have similar layers. In the same thread, Themis [15] proposes techniques that attempt to maximize GPU fairness via an auction-based scheduler, which may be important in systems with many clients competing for limited compute resources.

Clockwork [16] takes a contrasting approach to scheduling policy. Rather than focusing on utilization or fairness, it uses a central controller that attempts to execute requests in a manner as predictable as possible. This controller accounts for factors such as GPU memory states, timing of both inference and model loading actions, and real-time load balancing decisions. This more holistic approach to scheduling may be valuable in production systems that require stable and consistent performance over long periods of time.

2.3 Modern Innovations

2.3.1 Cost-Aware Scheduling

As inference-serving systems continue to gain more real-world applications, monetary cost has begun to increase as a factor to account for when making scheduling decisions. One paper that addresses this is INFaaS [17], which proposes a modelless serving system that allows users to specify performance and accuracy requirements for applications and aims to minimize cost for the user while meeting those requirements. To accomplish this, INFaaS uses a variety of methods, including a novel model variant search process using ILP and intelligent model placement across heterogeneous hardware resources in a way that maximizes hardware resource sharing between models.

Although INFaaS is applicable in many environments, it does not specifically look at the optimizations that can be made in a cloud environment. Many cloud providers offer what are known as ”spot instances” - resources available at a discounted rate when the provider has unused compute that would otherwise be left idle. Cocktail [18] looks specifically at optimizing for inference accuracy in a cloud environment while leveraging spot instances to minimize the cost of the system. Using an ensembling-based approach to model selection, along with a resource manager that can make use of these spot instances, they were able to match the cost savings of INFaaS while significantly improving latency and accuracy benchmarks.

In contrast to both of these papers, Proteus [19] approaches cost awareness from a different standpoint. Many systems may be constrained with regard to how much compute they have available and thus need to make decisions about how to best make use of lower footprint model variants while still meeting SLO deadlines. Proteus makes use of accuracy scaling, which is the ability to adjust the accuracy of the back-end models based on real-time workload. Using a MILP approach to optimize several subproblems, along with a novel batching approach, Proteus significantly reduces accuracy degradation and latency timeouts in hardware-constrained systems compared to baseline approaches.

2.3.2 Domain Specific Optimizations

The increased complexity of the hardware and infrastructure used by inference-serving systems has also led to the development of more complex approaches to serving specific applications. One example of this is computer vision, since serving requests for computer vision models requires large amounts of data from continuous photo/video input. One framework that attempts to mitigate these challenges is DeepRT [20], a framework designed to best serve computer vision models in an edge computing environment. Using a novel scheduler called "DisBatcher", which intelligently groups video frames that arrive within the same time window and require the same CNN model into batches, along with other key components that helped keep deadline misses to a minimum, they were able to maintain throughput on par with baseline comparisons while also offering latency guarantees better than existing frameworks such as BATCH [10] and AIMD-based approaches like Clipper.

Large language models (LLMs) are another domain in which fine-tuned strategies have been applied to boost system performance. One paper that presents a novel approach to dynamic batching for LLM serving systems is dLoRA [21]. dLoRA specifically looks at serving LoRA models, which are models that have small adapter matrices that are trained to slightly modify the output or expertise of a model. This is particularly useful for LLMs, which may have large base models that are expensive to train but can be made better for specific applications by simply retraining the adapter matrix for a certain task. dLoRA introduces two different modes, a "merged" mode, where the LoRA adapter is combined with the base model, and an "unmerged" mode, where the base model and adapter computations are done separately. The unmerged mode allows for batching across LoRA models with different adapter matrices, but introduces more computational overhead than the merged mode. By efficiently managing these two modes, they were able to significantly improve throughput compared to state-of-the-art LLM inference serving systems.

Another challenge facing LLM serving is the bottleneck of the KV cache. LLMs heavily use KV cache because they are required to maintain previously computed KV pairs for all previous tokens in a request to be able to generate the next token. Multiple different approaches have been taken to alleviate this bottleneck. vLLM [22] devised a PagedAttention mechanism, which divides the KV cache into blocks that can be stored non-contiguously, similar to how operating systems use virtual memory. The benefits are also analogous to virtual memory: less fragmentation of the memory space and the enabling of dynamic allocation of GPU memory for the KV cache. In contrast, DejaVu [23] focuses on intelligently managing the KV cache with respect to the entire request pipeline. This involves techniques such as streaming key-value pairs between prompt processing and token generation phases, data replication, and microbatch-level swapping between CPU and GPU memory. Both frameworks demonstrated a significant improvement over widely used trivial, contiguous KV cache management strategies.

2.4 SneakPeek and Data-Aware Optimization

An area of potential optimization for inference serving that has not been widely studied is the application of data-awareness in request scheduling techniques. The concept of data-awareness is that there may be patterns in the underlying distribution of input data that can be extracted and used as guidance for decisions made in the rest of the system. For inference-serving systems specifically, data-aware scheduling algorithms can be used to increase system performance over their baseline counterparts. This optimization was studied and implemented in SneakPeek. In this framework, input data from different sources is first run through a low-latency Bayesian estimator, which can be configured on a per-application basis to give a weight to the certainty of the estimation. The request scheduler can then include this information in its prioritization process, or even choose to use the estimate to satisfy the inference request in order to meet SLO deadlines.

The evaluation presented in the SneakPeek paper was done on a system with a single GPU and a single executor process. In this configuration, it was shown that a data-aware estimator applied to a novel grouped scheduling algorithm can maintain utility with an increasing number of request arrivals significantly better than data-oblivious scheduling algorithms. This result begs the question of how data awareness translates to a system with a more complicated infrastructure that better reflects the profile of real-world inference serving systems.

With the context of the wide variety of optimization techniques discussed, along with the more specific application of real-time systems in mind, SERIF was built as a flexible, configurable inference serving system that can be used to test new scheduling algorithms, serving infrastructures, optimization techniques, etc., with a minimal and well-documented configuration process for the user.

Chapter 3

Framework Architecture Overview

Figure 3.1 shows the high-level design of SERIF and illustrates the standard flow of data through the system. First, a worker managed by the data processing task reads data from a user-defined file (1). The logic for parsing and managing this file is owned by the user. After reading and formatting the data, the data processing task will optionally run the processed data through a user-defined classifier, which will assign the request with a "data-aware" hint for the scheduler. The data processing task then serializes and uploads the data to a Redis database (2b), and optionally, a local shared memory cache (2a) accessible to executor processes on the same node. Then, it sends the request token, data-aware hints, and other metadata to the scheduler (3). When the scheduler wakes up, it will pop all of the currently waiting requests off of its' queue, call into a user-defined scheduler which will determine how to map requests to executors (4). Once the schedule has been created, the scheduler sends requests to the executor nodes assigned to each request in the schedule (5). The executor nodes will then forward each request to an executor worker process (6). When the executor processes the request, it will read the data for the request either in the local shared memory cache (7a), or from Redis (7b) if it is not available in shared memory, and forward the request to the model specified by the scheduler (8).

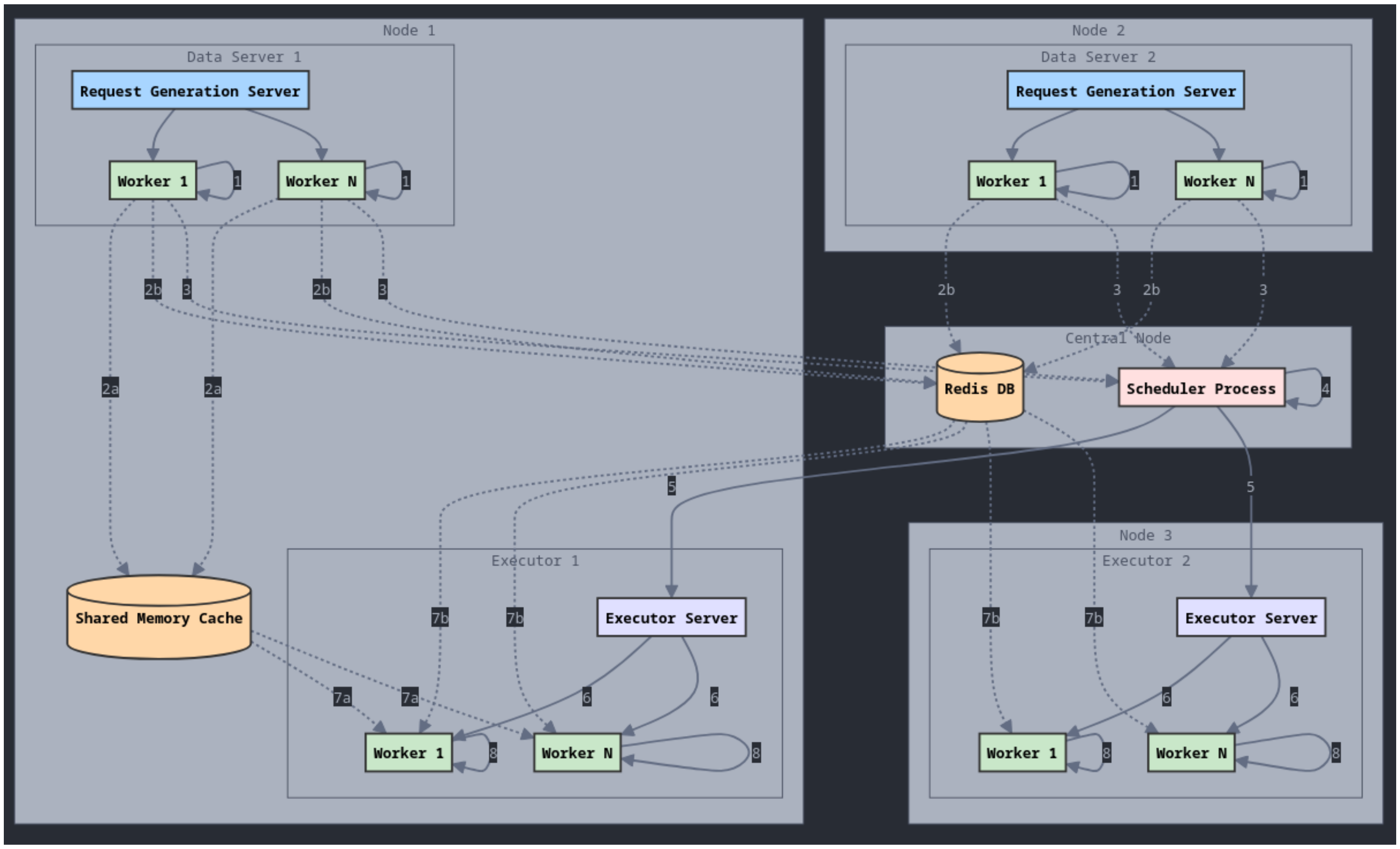

Figure 3.1: One potential configuration for SERIF.

Chapter 4

Implementation and Design

4.1 Framework Configuration

One of the overarching goals for SERIF is the ability to use it for a wide variety of applications within a distributed, heterogeneous infrastructure. To accomplish this, it was necessary to design a well-defined configuration that can manage/scale almost every aspect of the serving process. There are three different configuration components defined for this project: system configuration, application configuration, and data configuration.

The system configuration contains variables that are needed to configure components that are generally used by multiple different tasks within the framework. This involves defining the scheduler implementation to be used along with other scheduler metadata, defining the Redis node endpoints, and defining the executor nodes within the system. A complete list of variables and their usage can be found on GitHub[5].

The application configuration contains variables that are used to define each individual application that SERIF will serve. An application is defined as a task that requires model inference requests that need to be serviced across multiple data streams. Some examples of applications that were set up to test SERIF are defined in Table 5.2. Each application must contain information on things such as the required latency for requests, a utility matrix for model outcomes, and optionally, SneakPeek-specific data such as prior distribution of the data. In addition, each application has a list of model profiles that contain data about the accuracy of the model, latency metrics for each executor in the system, and a file containing weights for the trained model that will be pulled into executor memory during startup. A complete list of variables and their usage can be found on GitHub[5].

The data configuration contains variables that define the behavior of the real-time request generating nodes. The framework introduces the concept of a "modality". The user configures each modality with a name, the expected shape of the input data generated by the data processing task, and a maximum byte size to be used by the local KV store discussed in Section 4.2. In addition, the user defines a list of servers that generate inference requests. Each server is associated with a name, IP, and port, a list of data streams it will process on a per-modality basis, and information to facilitate its use of a local KV store. A complete list of variables and their usage can be found on GitHub[5].

Together, these configurations allow the user to tailor SERIF to serve requests on essentially any distributed infrastructure layout with minimal code changes to the framework, which will be discussed more in Section 4.3. For use cases where the user wants to ignore the effects of network latency, SERIF can also be configured to run fully on a single system, while also providing the ability to test schedulers, models, and parsing implementations for a system with multiple different request generation nodes and compute endpoints, which may better reflect the architecture of a real-world serving system.

4.2 Data Management

One challenge that needs to be solved in a distributed inference serving system is getting request data to executor processes as quickly as possible. In a single-node system, this is trivial to do efficiently, as all data can be passed between the request generator, scheduler, and executor via shared memory. In a distributed architecture, however, this gets trickier, as it is unknown which executor the request data will need to end up on. This framework solves this problem by sending all generated data requests to Redis before notifying the scheduler of their existence. The Redis database is accessible by all executor nodes via the system configuration described in Section 4.1.

While Redis is enough to functionally solve this problem, introducing network latency into the system can cause congestion at the executor nodes, as they can only execute requests as quickly as they can read data from Redis. In a cluster computing environment with fast LAN speeds, high bandwidth, and small request sizes, this latency can be negligible. In slower environments, however, this network latency can cause congestion that results in executors not being able to satisfy SLO timeouts due to an ever-increasing pile-up of requests. To help alleviate this, a configurable shared-memory KV cache implementation was designed to speed up requests that are assigned to an executor process on the same system from which the request originated.

4.2.1 Local KV Cache Design

The design of this KV cache is shown in Figures 4.1, 4.2, and 4.3. There are two main components of this cache: the data region and the metadata region. The data region is a large chunk of memory that contains only request data. It is divided into individual slots reserved for each modality, each of which is managed by the metadata region. The metadata region is a smaller chunk of memory that contains a serialized dictionary, where each of the keys is a slot number, and their corresponding values are another dictionary containing information to track the usage of the slot.

This cache was not designed to be able to store data for every request that is processed by it's corresponding request generator tasks, however, care is taken within the system to ensure that unused requests do not stick around for too long taking up space. If an executor finds the data for a request in shared memory, it will immediately remove it from the cache, and a garbage collector task also routinely removes in-use slots that have been idle for a user-defined waiting period. Depending on the system infrastructure and the scheduling implementation being used, this optimization can significantly decrease the effects of network latency, as demonstrated in Chapter 5.

4.3 Pluggable Components

Although the configuration discussed in Section 4.1 allows the user to mold SERIF to their architecture and use case, it is also necessary to provide a mechanism in code for the user to add custom logic for input data parsing, scheduling, and model forwarding. SERIF accomplishes this by providing high-level APIs that the user implements for each of these components. The implementation details for these APIS are provided in the

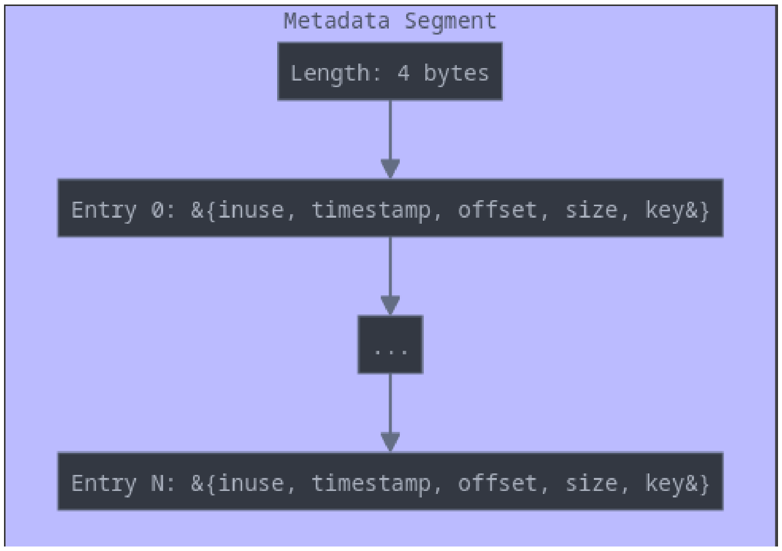
Metadata Segment
Length: 4 bytes
Entry 0: &{inuse, timestamp, offset, size, key&}
...
Entry N: &{inuse, timestamp, offset, size, key&}

Figure 4.1: Shared Memory Metadata Structure

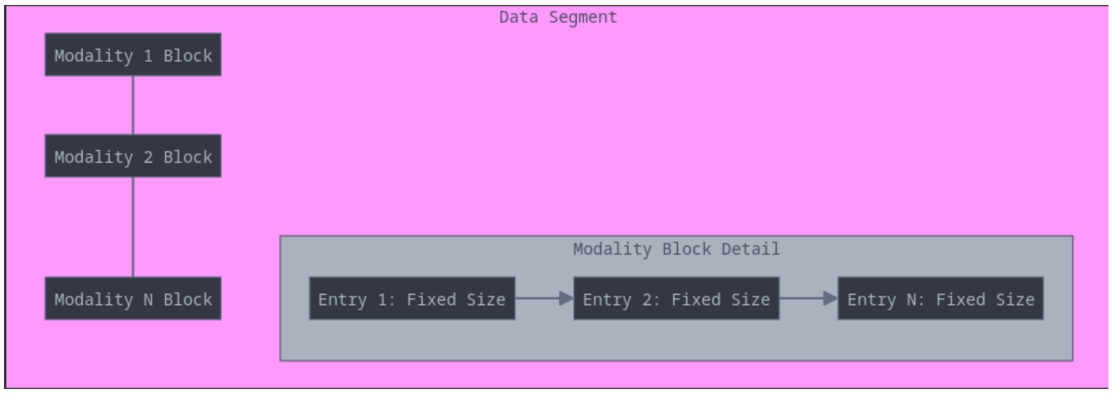

Figure 4.2: Shared Memory Data Structure

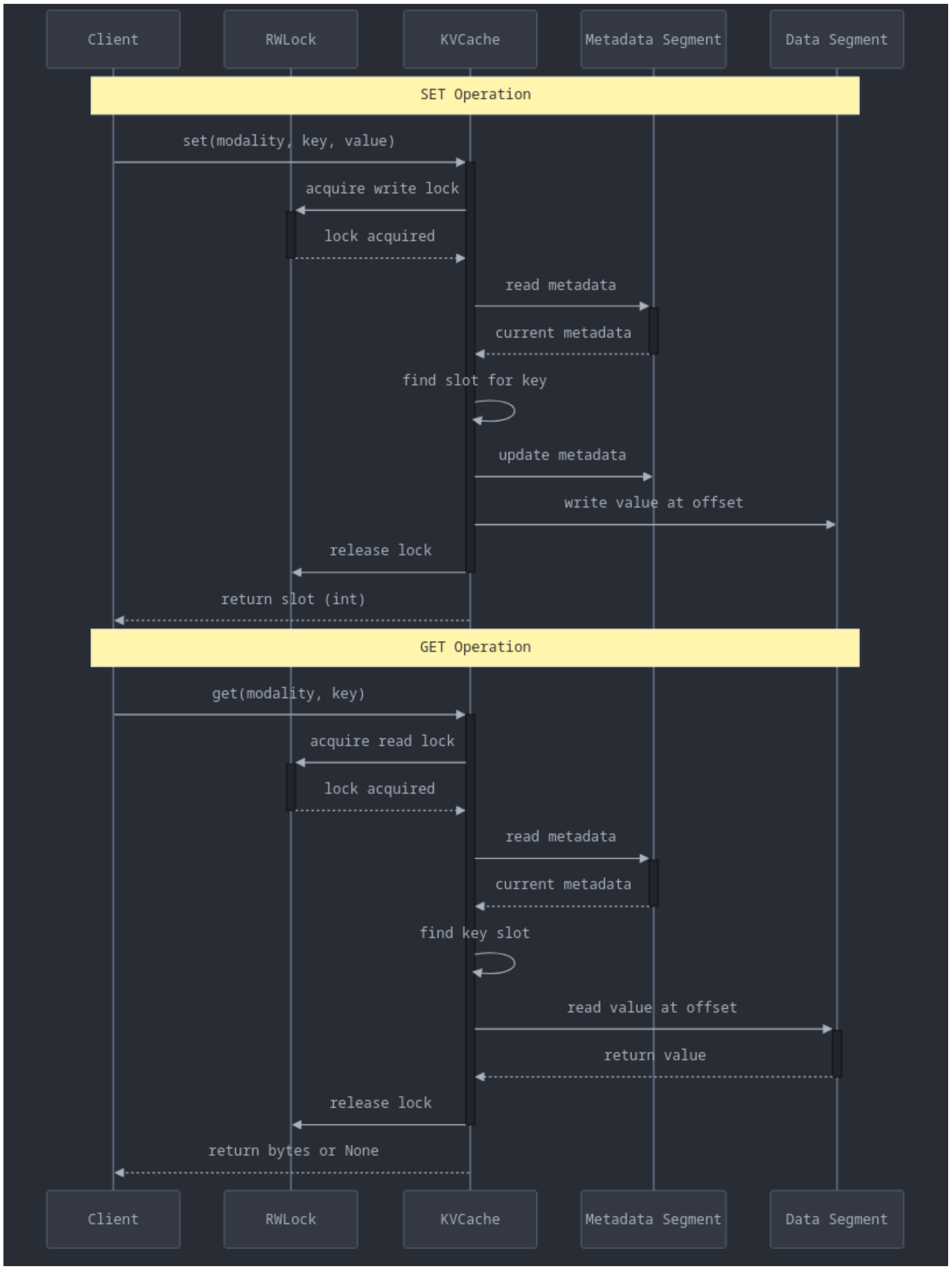

Figure 4.3: Sequence Diagram for GET and SET operations

SERIF documentation on GitHub[5].

4.4 Scheduler Implementations

In order to demonstrate the advantage of the KV cache presented in the previous sections, three different scheduling algorithms were tested to demonstrate the different qualities of the SERIF framework in varying distributed environments: a random scheduler, a locally optimal EDF scheduler, and the grouped scheduler presented in the SneakPeek paper. Each of these schedulers had to be extended to take network latency into account when calculating the utility for a certain schedule configuration. Network latency was penalized in two ways:

- For a request that was calculated to be processed before a deadline taking network latency into account, a small utility penalty is added if the request incurs additional network latency. This penalty scales with the amount of network latency, as well as how close the request is to the deadline.
- For a request that was calculated to be processed after the deadline for the request, a quadratically scaling penalty was added based on the additional network latency incurred. This harshly penalizes schedules that continue to pile on network-intensive assignments when executors are already backed up on requests they are processing.

In addition to extending each of these schedulers to take network latency into account, cache-prioritizing variants for the random and locally optimal EDF were also created. The only change these cache-prioritizing versions make is that if data for a request has been cached, the scheduler is forced to place the request on an executor node that has access to the cached version of that data. This optimization significantly aids with network-latency concerns; however, it is not easily compatible with approaches like the grouped scheduler, which aims to place similar requests, regardless of their origin, on the same backend executor nodes to be able to perform batched inference.

Chapter 5

Experimental Setup and Evaluation

In order to demonstrate the flexibility of the SERIF framework, different combinations of the following variables were tested for functionality and evaluated with respect to accuracy and latency in each executor node in the system.

- Input Data Load: Data points were taken with the system configured to use varying numbers of input data streams.
- Scheduling Algorithm: 5 different scheduling algorithms were tested in each configuration. These algorithms are detailed in Section 4.4.
- Shared Memory Cache Size: Experiments were conducted to demonstrate the difference in performance with larger/smaller cache sizes.
- Modalities: All experiments were done using video, sensor, and audio modalities. The video modality is extremely data intensive, with each request being about 60MB in size.

Each test was conducted with two contrived applications that could be applicable in a healthcare setting - wake-word monitoring and fall detection monitoring. Due to the complexity of setting up model forwarding and parsing code, only data from the video modality/video models for fall detection were set up, with the rest generating random data and simulating delay in the executor based on the profile of the model assigned to the request.

Table 5.1: System Specifications

System Name	**CPU Type**	**GPU Type**
cs-geeta	Intel Core i5-4590	NVIDIA RTX 3060
csel-cuda-01	Intel Xeon Gold 6148	NVIDIA Tesla T4
csel-cuda-02	Intel Xeon Gold 6148	NVIDIA Tesla T4

Table 5.2: Applications and Models

Application	**Description**	**Models and Modalities**
Fall Detection	Determine if someone has fallen	Video Modality: X3D models (small, medium, and large), Sensor Modality: MiniRocket
Wake Word Detection	Detect keywords spoken	Audio Modality: HOWL with LSTM and MobileNet

5.1 Experiment 1: Multi-Node Infrastructure with Data Intensive Input

5.1.1 Infrastructure Description

The infrastructure used in this experiment was set up as follows:

- Request Generation Nodes: Two request generation nodes were set up with 5 workers each, one on cs-geeta and one on csel-cuda-01.
- Executor Nodes: Three executor nodes were set up, one on cs-geeta, one on csel-cuda-01, and one on csel-cuda-02.
- Scheduler Node: The scheduler process was hosted on csel-cuda-01.
- Redis Node: cs-geeta hosted all of the Redis servers used in the Redis cluster configured for the experiment.

5.1.2 Experiment Variables

- Number of Input Groups: 10, 20, 30, and 40 groups were tested for each configuration.

- Cache Sizes: The smaller input cache contained, for each modality, cache entries for 50% of the groups being processed on each request generation node. The larger input cache contained entries for 80% of the groups that were processed at each generation node.

- Deadline: A 5 second deadline was imposed for both applications across all tests. This means that estimated request latencies above 5 seconds were punished more significantly scaling with network latency.

5.1.3 Evaluation

Figures 5.1, 5.2, 5.3, and 5.4 show the results for this experiment. There are a few conclusions that can be drawn from the data.

- All algorithms that did something intelligent with the request scheduling/placement significantly outperformed the random scheduler from a latency perspective.

- In general, the cache-prioritization variants of the random and EDF schedulers outperformed their counterparts when looking at both latency/accuracy when a larger cache size was used. With a smaller cache, they performed fairly comparably, with the variants that did not especially prioritize cache performing slightly better with larger input data loads.

- The grouped scheduler generally performed better from a latency perspective than the EDF schedulers, with the trade-off of a slight accuracy disadvantage. However, the grouped scheduler places similar requests together to allow for batched inference, so it may have the potential to perform even better if the executor had batched inference capability, as discussed in Section 6.1.

- Table 5.3 shows that the evaluated scheduling algorithms do add somewhat significant overhead to the system. However, the overhead per request is still relatively low - only about 0.03-0.04 seconds at the very high end. This extra overhead is justified by the performance gained from placing requests in an efficient manner. One observation made was that the variance of the scheduling overhead was fairly high, however, across all tests, the random schedulers generally incurred less overhead than the EDF and grouped schedulers.

- Overall, SERIF performs better with high shared memory cache sizes when data-intensive modalities are in use. This experiment examined scenarios in a memory-restricted environment, however, nodes with access to large amounts of RAM could choose to significantly increase these cache sizes to ensure that the percentage of requests that cannot be cached stays consistently low.

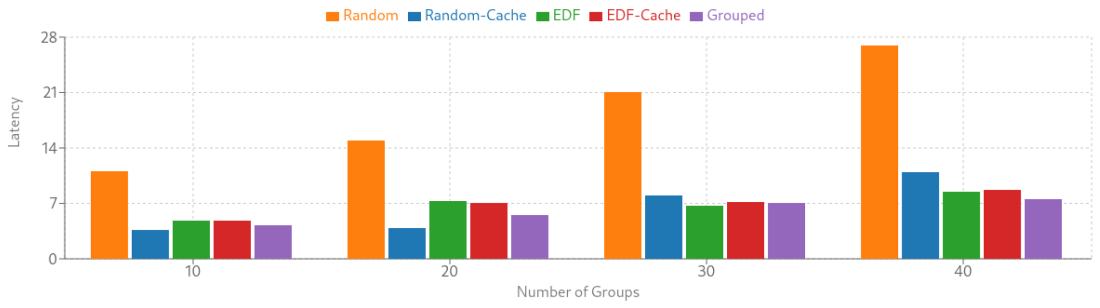

Figure 5.1: Latencies across different schedulers using a small cache size.

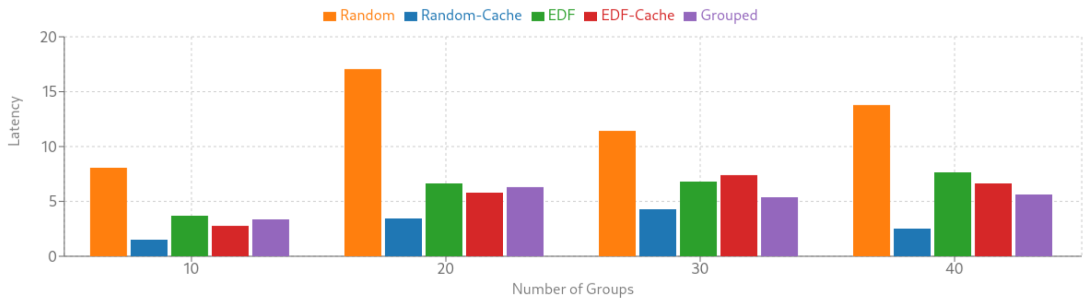

Figure 5.2: Latencies across different schedulers using a large cache size.

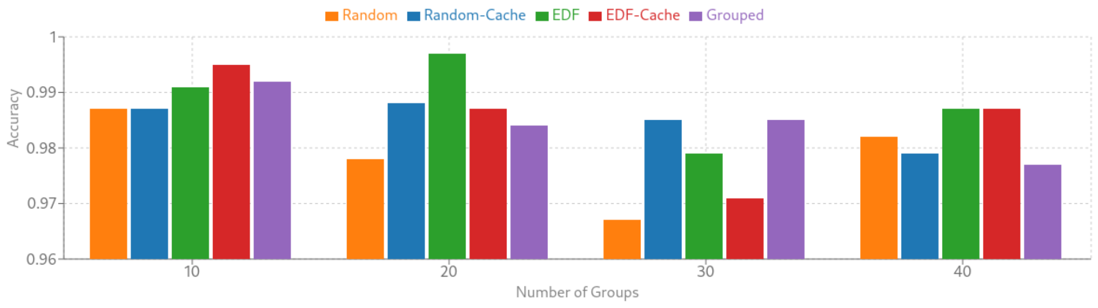

Figure 5.3: Accuracies across different schedulers using a small cache size.

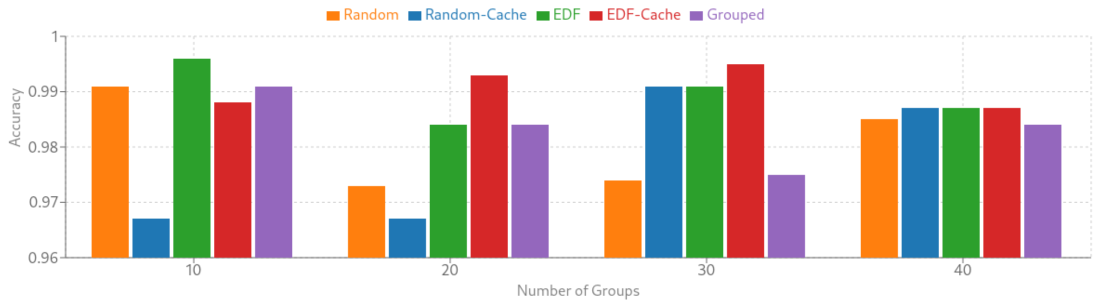

Figure 5.4: Accuracies across different schedulers using a large cache size.

5.2 Experiment 2: Reducing Network Latency

Some assorted tests were also run with the system configured in ways that reduce network latency. The two approaches to accomplish this were:

- Using the same multi-node architecture but removing the video modality from the system.
- Using a single-node architecture to eliminate all internode communication

The results of these tests demonstrated that the scheduler algorithms used in this evaluation introduce enough overhead with a high number of requests that the accuracy/latency gained from using them does not particularly outweigh their costs. However, there are future optimizations, discussed in Section 6.1, that could be made to

Table 5.3: Scheduling Overhead

Algorithm / Groups	**10**	**40**
Random	3.639	5.460
Random-Cache	4.09	6.699
EDF	7.698	12.694
EDF-Cache	5.282	11.707
Grouped	7.514	8.851

reduce scheduling/system overhead to allow for more accurate experiments in environments with lower overall network latency.

Chapter 6

Conclusion and Future Direction

6.1 Future Direction

The existing implementation of SERIF accomplishes the main goals described in the paper. It provides a configurable, extensible inference serving implementation that can be used for future research in this area. That being said, there are still a few things that could be modified to enhance the usability of the tool.

- Right now, while the user has the ability to group requests together that can be batch processed on an executor, SERIF does not have any sort of batching support baked in. It may be possible for the user to make use of the pluggable components described in Section 4.3 to do this, but it may be easier if this support is implemented directly in the SERIF framework.

- While the system is highly configurable and has a well-defined process for setting up configuration files, it can still be time consuming to set up the files, especially if many input data streams need to be specified. A frontend configuration tool that could be used to generate configuration files could be helpful for improving usability of the system.

- For use in a production environment, it may be helpful to have the ability to modify the configuration of the system while it is running. This could mean adding new executor or request generating nodes, adding new model types, etc.

- As discussed in Chapter 5, modalities with large input data sizes tend to significantly increase lag within a real-time inference serving environment. Beyond utilizing shared-memory caching for executors that are co-located with request-generation nodes, it may be possible to make other datapath optimizations, such as decoupling the data retrieval path from the execution path, pre-forwarding data to executor nodes likely to end up using it, or utilizing methods other than shared memory to allow for faster data streaming between nodes such as RDMA.

- More work needs to be done to create a mechanism for SERIF to send prediction results somewhere. This could be done with another pluggable component/API, or SERIF could have a built-in solution that the user could configure.

- SERIF currently uses XMLRPC for inter-node communication. This library is easy to work with; however, it also incurs substantially more overhead than using the gRPC library. Switching to gRPC could improve communication between different nodes in the system, especially if the system is used with a high number of input streams or data modalities.

- SERIF currently does not have a plug-in for the data-aware estimation component described in SneakPeek, but this will likely be added in the near future.

6.2 Conclusion

In conclusion, the SERIF implementation demonstrates the potential of an extendable, scalable framework for inference serving in the context of real-time inference serving systems. The results in this paper show that with data-intensive applications, the shared memory caching solution paired with network latency aware schedulers can significantly reduce overall request latency in the system. In addition, the design of SERIF makes it possible for future work to easily test other backend scheduling implementations, models, and applications. Overall, SERIF is a good first step towards a usable research tool for real-time inference serving systems.

References

[1] Francois Chollet. OpenAI o3 Breakthrough High Score on ARC-AGI-Pub, 2024. Accessed: January 6, 2025.

[2] Brett Young. OpenAI Introduces o3: Pushing the Boundaries of AI Reasoning, 2024. Accessed: January 6, 2025.

[3] J. et al. Jumper. Highly accurate protein structure prediction with alphafold. *Nature*, 596:583–589, 2021.

[4] Joel Wolfrath. *Data-Aware Optimizations for Efficient Analytics over Geo-Distributed Data.* PhD thesis, University of Minnesota, 2024.

[5] Daniel Frink. Serif: A scalable, extendable real-time inference framework. `https://github.com/dndfrink/SERIF`. Accessed: 2025-01-27.

[6] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: a scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, page 173–182, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *CoRR*, abs/1409.3809, 2014, 1409.3809.

[8] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: a low-latency online prediction serving system. In

Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, NSDI'17, page 613–627, USA, 2017. USENIX Association.

[9] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ML serving. *CoRR*, abs/1712.06139, 2017, 1712.06139.

[10] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[11] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, page 109–120, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. Fa2: Fast, accurate autoscaling for serving deep learning inference with sla guarantees. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 146–159, 2022.

[13] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, February 2020. USENIX Association.

[16] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: performance predictability from the bottom up. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[17] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Managed & model-less inference serving. *CoRR*, abs/1905.13348, 2019, 1905.13348.

[18] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: Leveraging ensemble learning for optimized model serving in public cloud. *CoRR*, abs/2106.05345, 2021, 2106.05345.

[19] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 318–334, New York, NY, USA, 2024. Association for Computing Machinery.

[20] Zhe Yang, Klara Nahrstedt, Hongpeng Guo, and Qian Zhou. Deeprt: A soft real time scheduler for computer vision applications on the edge. *CoRR*, abs/2105.01803, 2021, 2105.01803.

[21] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 911–927, Santa Clara, CA, July 2024. USENIX Association.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Ré, and Beidi Chen. Deja vu: contextual sparsity for efficient llms at inference time. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Appendix A

Glossary

- **AIMD** – Additive Increase/Multiplicative Decrease
- **ARC-AGI** – Abstract and Reasoning Corpus for Artificial General Intelligence
- **Data Awareness** – Refers to methods that take some part of the underlying distribution of data into account to improve system performance.
- **DNN** – Deep Neural Network - A machine learning model built with multiple layers of interconnected ”neurons” between the input and output layers.
- **GPU** – Graphics Processing Unit - A hardware accelerator designed initially for rendering graphics, but in machine learning contexts, it can accelerate neural network training and inference by processing multiple matrix operations in parallel.
- **Inference Request** – A query sent to a machine learning model to get a prediction or output based on the input data.
- **KV Cache** – A key/value store in GPU memory that stores intermediate output tensors computed during inference to avoid redundant computation.
- **LoRA** – Low Rank Adaptation - A technique used to fine-tune large language models by adding small ”adapter” matrices to the model's existing layers.
- **(M)ILP** – (Mixed) Integer Linear Programming - An optimization technique used for problems where either some of the variables are integers (MILP), or all of the variables are integers (ILP).

- **Modelless System** – An inference architecture that executes machine learning models without maintaining persistent model copies in memory, instead loading model parameters on-demand for each inference request.

- **RDMA** – Remote Direct Memory Access - A method to allow two separate computers to directly access each other's memory spaces without incurring the overhead of involving the operating system in the memory access protocol.

- **SLA or SLO** – Service Level Agreement/Objective - the advertised latency that the inference serving system promises to fulfill as many requests as possible within.

- **Utility** – A measure of how well a system is performing on a variety of metrics important to the user, such as latency, throughput, and accuracy.