Enabling Efficient ML Inference in SigmaOS with Model-Aware Scheduling

by

Katie Liu

S.B. Computer Science & Engineering, MIT, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Katie Liu
Department of Electrical Engineering and Computer Science
May 16, 2025

Certified by: Ariel Szekely
Doctoral Candidate, Thesis Supervisor

Certified by: M. Frans Kaashoek
Charles Piper Professor of EECS, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Enabling Efficient ML Inference in SigmaOS with Model-Aware Scheduling

by

Katie Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Machine learning inference in multi-tenant cloud environments leads to significant challenges when it comes to minimizing latency and resource contention, especially as models grow in size and complexity. This thesis addresses the cold start overhead and scheduling inefficiencies of multi-tenant ML serving by integrating the RayServe distributed model-serving framework into σOS, a cloud operating system that unifies container and serverless paradigms. The thesis also proposes two model-aware schedulers within σOS that intelligently routes inference requests to reduce the number of cold starts: Model Colocation, which prioritizes placing requests on machines where the required model is already loaded, and Centralized Model Registry, which tracks globally available models to inform scheduling decisions. These policies proactively reduce model load times by reusing cached models. Experimental results on language translation workloads in an 8-node cluster show that these schedulers achieve a $\approx 50\%$ reduction in average inference latency and eliminates roughly 4–5 cold starts per workload, compared to σOS's default scheduler. Through this model-aware approach to scheduling, our work enables more efficient, scalable, and low-latency ML inference serving in multi-tenant cloud settings.

Thesis supervisor: Ariel Szekely
Title: Doctoral Candidate

Thesis supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of EECS

Acknowledgments

Thank you to Ariel and Frans for being so generous with your time and answering my silly questions over the past year. This work would not have been possible without your patience, support, and mentorship. It's been inspiring to witness your passion for hacking, and your guidance has truly been invaluable in helping me navigate the daunting research world.

Thank you to my friends for the laughs, conversations, and unforgettable memories over the past four years, and for helping me maintain my sanity through all the ups and downs. The friendships and communities I've been fortunate to find here have shaped me into who I am today, and are what I'll look back on the most fondly during this time. I'm incredibly lucky to have met each of you.

Finally, thank you to my family – my parents, Kathy and Bill, and my brother, Patrick – for your unconditional love and unwavering support throughout my entire life. None of this is possible without you, and I don't think I could ever express in words how grateful I am for you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning, which is becoming an integral part of almost every industry, requires scalable, efficient, and low-latency infrastructures to handle increasingly complex real-time workloads. Serving ML models presents significant challenges, including managing large model sizes and memory requirements, efficiently scheduling workloads across multiple nodes, and handling resource contention in distributed systems.

RayServe [1] is a framework designed to address these challenges, providing a scalable solution for deploying ML models in a distributed system. By abstracting away the complexities of infrastructure management, RayServe enables developers to focus on deploying models without worrying about system orchestration or resource allocation [2]. However, while ML serving is an important building block for cloud applications, RayServe does not orchestrate multi-tenant microservice applications.

σOS ([3], [4]) is a multi-tenant cloud operating system implemented in Go [5] that combines container orchestration and serverless under a unified platform. In doing so, it provides fast startup times and support for long-running stateful microservices to all tasks, enabling them to benefit from the best properties of both serverless and container platforms. σOS uses a centralized scheduler and resource requests to schedule long-running tasks, and uses real-time resource utilization to inform scheduling decisions. Effective scheduling is crucial for efficiently allocating resources, prioritizing tasks, and preventing bottlenecks, especially in multi-tenant environments, where fairness, isolation, and scalability must be maintained for good performance.

Traditional schedulers do not account for the unique requirements of ML workloads such as the size, initialization time, and computational time. This thesis presents the design, implementation, and evaluation of two model-aware schedulers that intelligently orchestrate inference requests to minimize average latency by reducing the number of cold starts, and therefore the overhead of model loading and configuration.

1.1 Contributions

We first integrate GPU support and RayServe into σOS, enabling scalable, distributed inference serving on top of σOS's flexible programming model. This integration highlighted the performance bottlenecks caused by model loading and cold starts, which informed the design

of the model-aware scheduler.

We introduce two novel scheduling policies:

1. **Model Colocation Scheduling**: Prioritizes colocating inference requests with models already available in memory on the same machine. It aims to minimize the number of cold starts by routing requests to machines where the models are cached. When warm proxies are unavailable, this policy either queues requests at the proxy (*Colocation With Queuing*) or processes the request immediately on another machine (*Colocation Without Queuing*).
2. **Centralized Model Registry Scheduling**: Maintains a global registry of available models across the system, allowing for routing of requests to machines that have the models available, even if they are not colocated with the request.

We evaluate these policies against the default σOS scheduling policy to demonstrate an overall reduction in the number of cold starts and end-to-end workload latency. In particular, we observe a 50% average reduction in latency and 4-5 fewer cold-starts on an 8-node cluster for three sample workloads. Finally, we summarize these results with an analytical model to determine the best policy for a given workload.

1.2 Roadmap

Chapter 2 provides an overview of RayServe, σOS, and their relevant features. Chapter 3 describes the motivation behind the cold start problem that this thesis targets. Chapters 4 and 5 detail the design and implementation of the model-aware scheduler. Chapter 6 evaluates the performance of various scheduling policies using a range of translation workloads and Chapter 7 concludes by discussing future directions of research.

Chapter 2

Background

2.1 Inference Servers

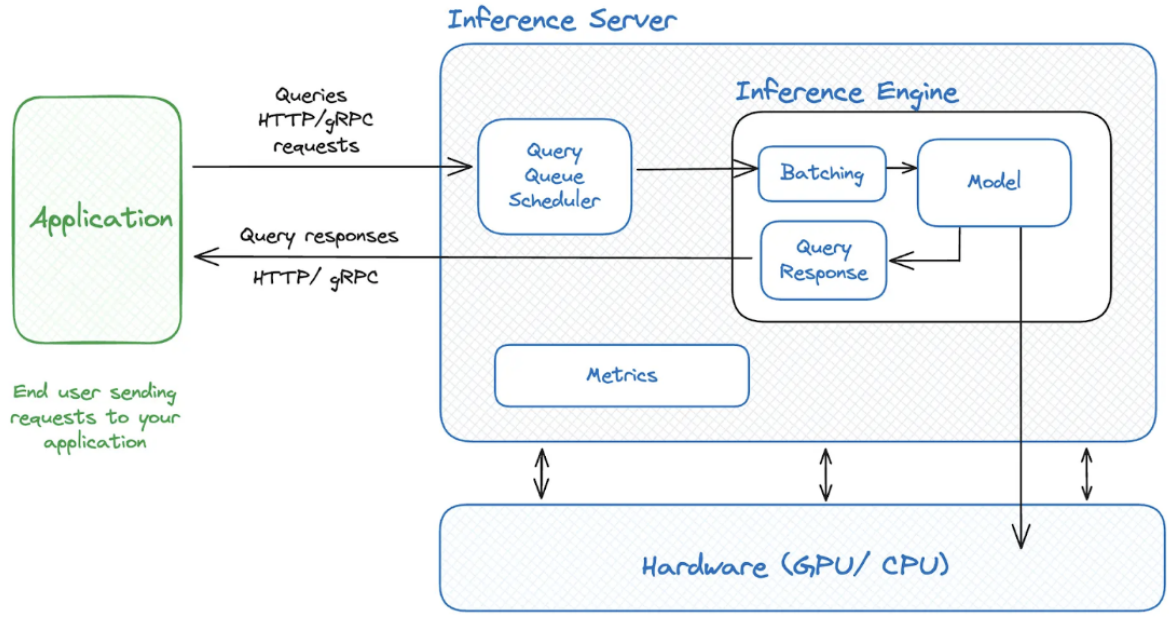

Figure 2.1: Typical architecture of LLM inference and serving [6].

An ML inference server is a system designed to efficiently make predictions on unseen data. Unlike training, which involves adjusting model parameters to improve model accuracy, inference involves using trained models to generate outputs, such as predictions or classifications, based on the model's learned patterns.

Inference servers expose API or service endpoints that allow external applications or users to submit data for processing. Each server consists of an inference engine responsible for running the model and generating predictions. Each server also manages the computational resources (CPUs, GPUs, TPUs) needed to execute the model and ensures scalability by dynamically adjusting the number of workers to handle concurrent requests.

Common performance metrics include throughput, typically measured in the number of requests processed per second, and latency, the time it takes from when an input request is received by the server until when the output is returned, which is particularly important in real-time streaming applications.

2.2 RayServe

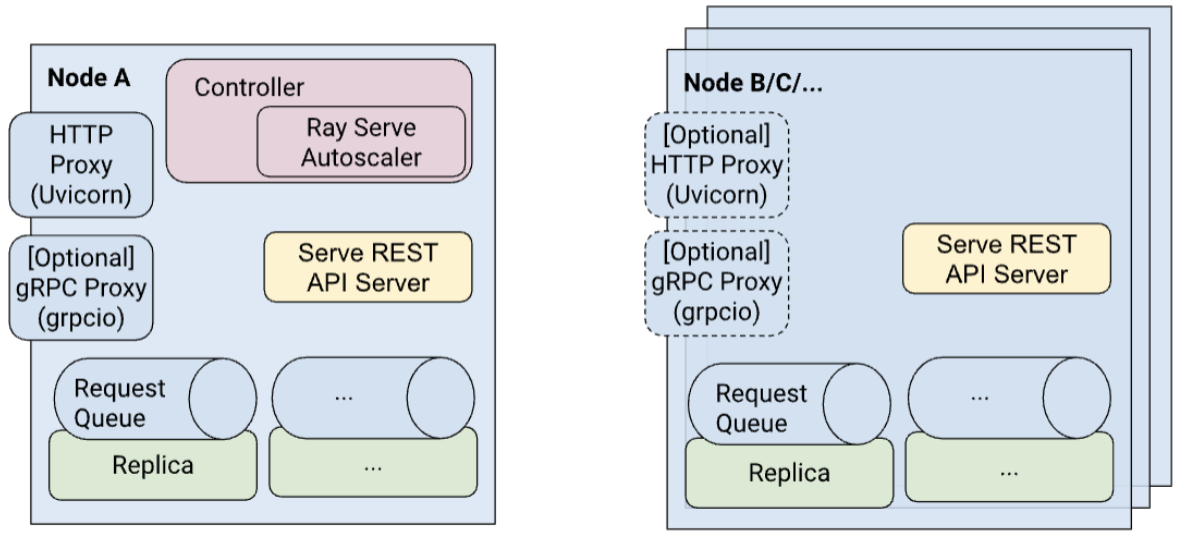

Figure 2.2: RayServe architecture [7].

Ray [8] is a distributed computing framework designed for parallel and distributed execution. It utilizes tasks and actors for parallel execution: tasks are stateless, remote function calls executed asynchronously, while actors are stateful objects that maintain internal state across method calls. Remote functions are invoked with `.remote()` and return futures that represent the result of asynchronous computations. Ray's abstractions simplify the development of scalable distributed systems by handling resource allocation and execution within a cluster.

RayServe, which is built on top of Ray, is a framework-agnostic Python model serving library for building online inference APIs. It allows for building a complex inference service consisting of multiple models and application logic, and offers scheduling support, resource sharing, and easy scaling within and across machines. Compared to Slurm's [9] static resource allocation and Triton's [10] reliance on external systems like Kubernetes [11], RayServe provides a simpler, more scalable solution for serving models at scale with low-latency needs.

A Serve instance consists of three types of actors: a global Controller, an HTTP proxy, and replicas. The Controller manages all other actors, including an autoscaler, which will try to increase worker nodes when resource demand grows and remove worker nodes when they sit idle. The HTTP proxy, built on Uvicorn, handles incoming traffic and routes requests to replicas. Each replica utilizes its own queue to process requests.

When a request enters RayServe, it is handled asynchronously via the ASGI protocol, which allows multiple requests to be processed concurrently. The HTTP server listens for events in the event loop, and requests are sent to replicas based on a simple scheduling policy that considers the length of the request queue. Each request triggers the `__call__` method (Section 5.1) in the user-defined class, which processes the request and returns the appropriate response.

Meanwhile, each replica regularly reports autoscaling metrics to the Controller, which monitors the system's overall load and ensures an even distribution of requests across the replicas.

2.3 σOS

2.3.1 Procs

Each execution in σOS is called a `proc`. σOS's cloud-centric API allows procs to interact through a network addressing scheme – service discovery occurs through a per-tenant naming service, named, which is used to register endpoints and store state. In this way, communication is only allowed among procs of the same tenant. The lightweight σcontainers that procs are spawned in limit their access to system calls and allow for fast start-up times.

2.3.2 Default Scheduler

Each proc is a task that is marked as either latency-critical (LC) or best effort (BE), which reflects the key distinction between the two types of tasks that are unified under σOS. LC procs are configured with reserved CPU time and RAM while BE procs are scheduled as CPU and RAM become available. BE procs allow σOS to fill in the gaps in resource reservations, especially when resources are overprovisioned and result in low average utilization.

σOS uses a centralized scheduler (`LCSched`, similar to the Kubernetes scheduler [12], to manage latency-critical tasks. Each machine has a scheduler, `MSched`, identified by a unique kernel ID, that coordinates with the global `LCSched` for scheduling LC procs and several BEScheds for scheduling BE procs. `LCSched` and `BESched` both maintain a queue of procs as they become ready to execute. An in-memory table tracks the available CPU and memory on each machine running σOS, which is continuously updated as procs start and stop. In `LCSched`, a background thread iterates over every realm (per-tenant namespace) in a round-robin fashion, dequeuing procs to execute as they meet resource eligibility. `MSched` interacts with the host OS to discover runnable processes, assign resources, and more. Once a proc is scheduled to a machine, the associated `MSched` will spawn and execute the proc.

2.3.3 Proxies

In σOS, a proxy is a long-running provider-managed service that projects an external resource into the per-tenant global namespace. From the perspective of any other σOS proc, the proxy appears as an ordinary σOS namespace, and its contents can be accessed using the standard σOS APIs.

Behind the scenes, the pathname is a string that represents the location or address of a resource or service within the global namespace. When a client queries a proxy, the pathname is translated into the remote protocol by whichever proxy received the request. Each proxy registers itself under a well-known path within the namespace (e.g., `/s3/s3srv_0`, `/s3/s3srv_1`). σOS resolves the pathname to route the request to the correct proxy. For example, a pathname with $\sim$`any` allows σOS to locate any available proxy, while $\sim$`local` instructs σOS to locate a proxy on the same machine. The latter can help to avoid network transfers and enable local resource access without the proc needing to know what machine they are physically run on. The following is an example of a directory structure within σOS, where different services are organized under a global namespace.

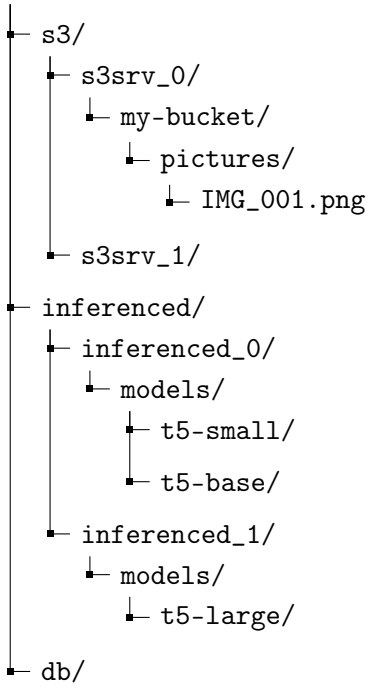

Figure 2.3: Example σOS directory tree.

The `s3/` directory contains multiple S3 server proxies (`s3srv_0`, `s3srv_1`), each exposing a tenant's Amazon S3 buckets and files. The `db/` directory provides access to databases like MongoDB. The `inferenced/` directory holds inference server proxies (`inferenced_0`, `inferenced_1`), each maintaining a directory of cached models. For example, t5-small and t5-base are cached on `inferenced_0`, while t5-large is cached on `inferenced_1`.

Chapter 3

Motivation

3.1 RayServe Performance Observations

The initial performance experiments are conducted on a t3.xlarge AWS EC2 instance with 4 vCPUs, 16.0 GiB memory, and a 5 Gbps network bandwidth [13]. Consider a simple translation request of "quick brown fox jumps over" from English to French using a t5-small model. Upon making a `POST` request to the translate endpoint, the request spends 2.788 seconds initializing t5-small (Section 3.2) and 0.923 seconds performing inference for a total wall clock time of 3.711 seconds. After completion of prediction, RayServe generates an HTTP response to send to the client.

The overall request latency suggests that the most significant delay of a RayServe request occurs during model loading, since model inference itself is relatively fast. As further investigation, we run a workload for one hundred concurrent translation requests of the same five word phrase, with a default RayServe autoscaling policy (minimum of 1 replica, maximum of 4 replicas, and a target of 2 ongoing requests per replica):

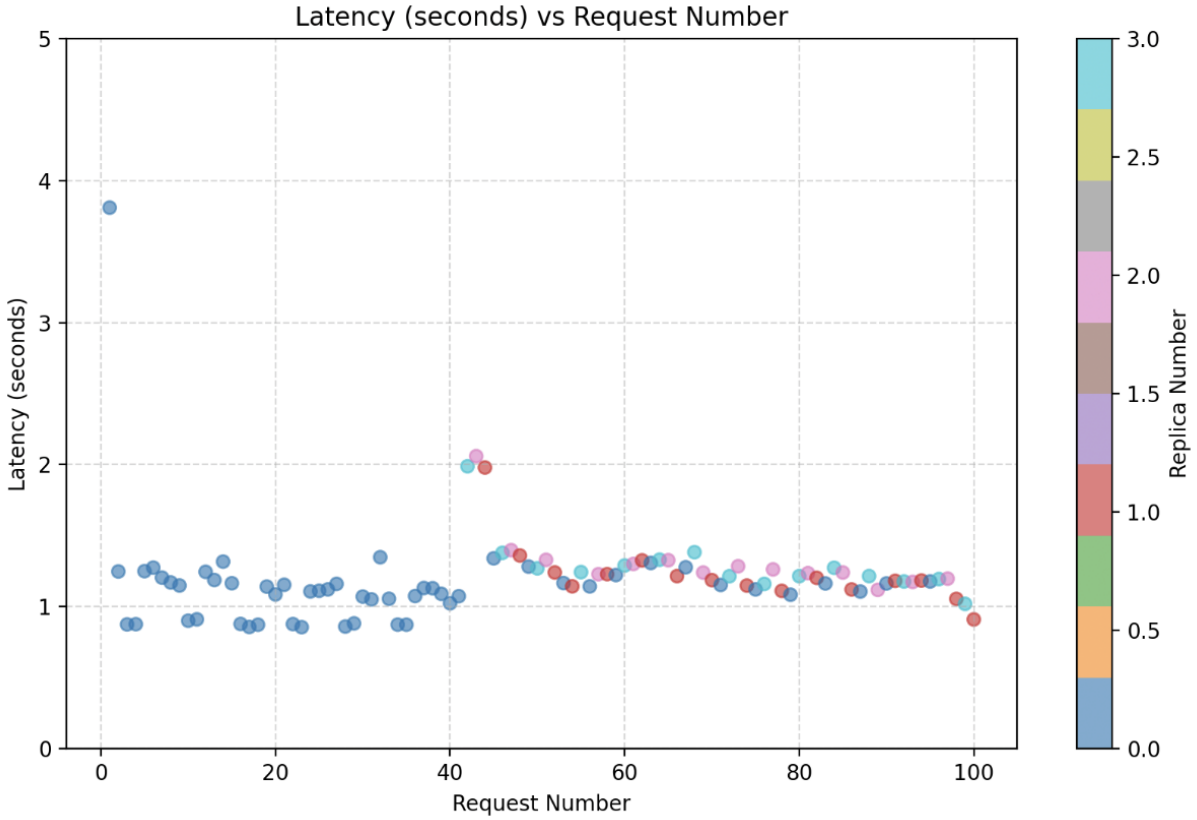

Figure 3.1: Latency of requests over time for a workload of 100 concurrent translate requests.

We make two observations:

1. It takes a long time for additional replicas to spin up – the same replica handes the first 40 requests before the other 3 replicas start to. This is configurable with RayServe parameters.

2. The initial request for each replica is by far the most expensive operation. Replica 0's first request requires nearly 4 seconds while subsequent requests hover around 1 second. The first request for replicas 1-3 requires 2 seconds, after which latency also drops to around 1 second.

3.2 Transformers Pipeline

We first examine the process of model initialization and explore potential strategies for reducing cold start time. The RayServe application instantiates models using pipelines from the HuggingFace transformers library [14], which wraps pre-processing components, the model that generates predictions from the inputs, and post-processing components into one. We break down the steps to set up a t5-small translation pipeline: `pipeline(translation_en_to_fr, model=model_name)`.

The bulk of the total 3.35 second initialization time is loading the pretrained model and tokenizer:

Step	**Time**
Loading pretrained model	2.73s
Download `config.json`	0.1s
Download `model.safetensors`	2.27s
Download `generation_config.json`	0.07s
Get `T5ForConditionalGeneration` model class	0.06s
Finalize model weight initialization	0.23s
Load pretrained tokenizer	0.49s
Download `tokenizer_config.json`	0.06s
Download cached vocab files	0.3s
`spiece.model`	0.08s
`tokenizer.json`	0.12s
`added_tokens.json`	0.04s
`special_tokens_map.json`	0.03s
`chat_template.jinja`	0.03s
Instantiate tokenizer	0.13s

Table 3.1: Model Loading and Tokenizer Setup Breakdown

Many initialization steps involve downloading the necessary files from the HuggingFace Hub and caching them locally. The first time the pipeline is set up, the model files are fetched and stored in the local cache. On subsequent pipeline initializations, the system first checks the cache to see if the required model and files are already available. If so, the pipeline directly loads the files from the cache, significantly reducing the setup time to just the final step (approximately 0.4 seconds in this case).

3.3 Model Cold Start and Inference Times

The initial model loading process not only takes a significant amount of time, but also becomes progressively more costly as the model grows in complexity. We isolate the initialization time of each model by taking the average over five separate cold starts, each the result of sending one translation request to RayServe, for each model type.

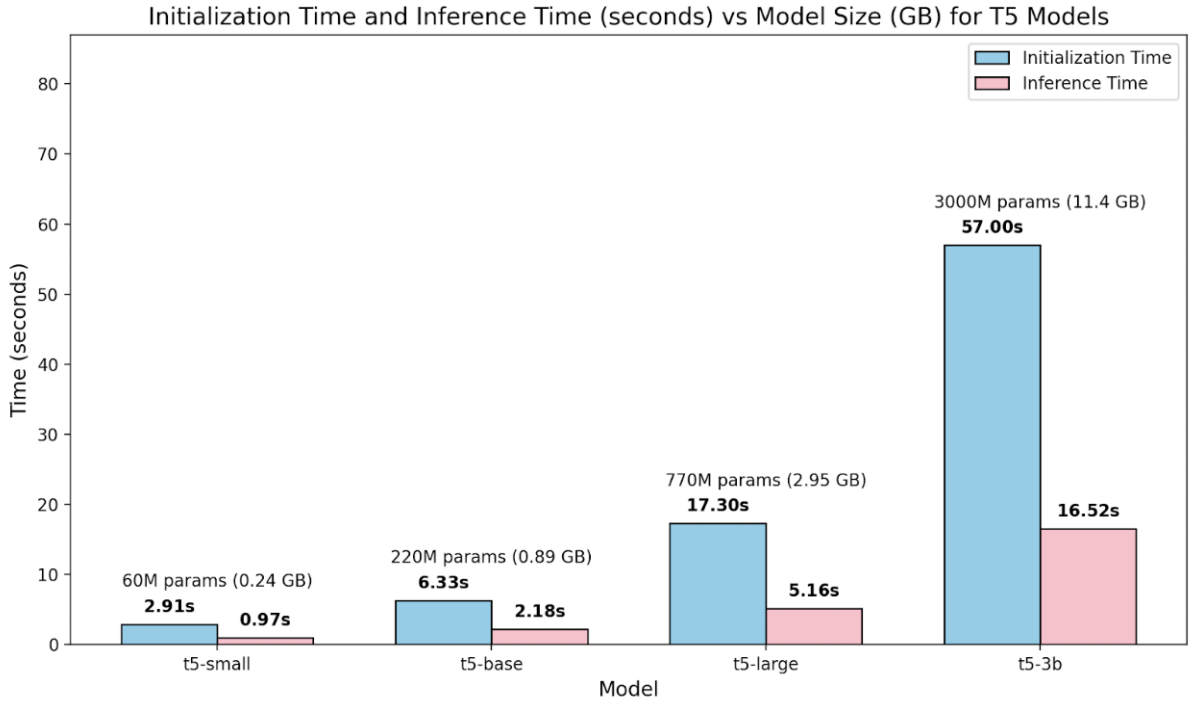

Figure 3.2: Average initialization (3.2) and model inference time for four T5 models on a c220g5 node (machine spec in Section 6.1). Each set of bars is labeled with the number of parameters and size of `model.safetensors` [15], which contains the model weights.

The figure demonstrates that inference is significantly faster than initialization time, with the difference becoming more pronounced as the model size increases. There is also a linear relationship between the model size (or number of parameters) and the average initialization time, meaning initialization time increases proportionally as the model size grows. In a distributed system, where multiple machines may need to load and serve large models, these initialization costs can accumulate, significantly impacting overall system performance. For example, t5-3b, with 3 billion parameters, takes nearly a minute to initialize.

Note that the primary bottleneck in initialization occurs during the download of pretrained models in the Transformers pipeline (Section 3.2), which accounts for 2.73 seconds (81.5%) of the 3.35-second initialization time. Given this, we focus on strategies to avoid cold starts.

Chapter 4

Design

The purpose of being model-aware is to minimize the end-to-end latency of RayServe workloads by tracking and utilizing models cached on each machine to reduce the number of times models need to be loaded from remote storage. This chapter discusses the design of two schedulers towards this end.

4.1 Overview

We introduce two primary scheduling policies: Model Colocation and Centralized Model Registry. Both aim to route inference requests to machines where models are already cached.

1. **Model Colocation**: Places inference tasks on the same machine where the required model is cached to avoid remote access latency. This policy is divided into two subpolicies:
    i. *With Queuing*: If no warm proxy is available, the scheduler queues the incoming requests and waits for the proxy to become available.
    ii. *Without Queuing*: If no warm proxy is available, the scheduler spawns the proc on a cold machine and sends the request to that proxy.
2. **Centralized Model Registry**: Spawns procs on machines that either queries local proxies or sends requests to remote warm proxies over the network.

4.2 RayServe Application

We build a RayServe application in Python that provides three scalable services that are representative of common AI workloads: translation, image classification, and text-to-image generation.

Translation is a classic NLP task used across industries, image classification is a fundamental task in computer vision and widely applied in medical diagnostics, autonomous vehicles, etc., and text-to-image generation has become popular in the art and design industries. These tasks each have different inference time characteristics. Image classification

often involves large input sizes, and text-to-image generation models often require extensive GPU computation. This variability in task type allows for flexible exploration of inference characteristics such as autoscaling, load balancing, and resource utilization.

Each service is deployed as a Ray actor with autoscaling configurations, ensuring efficient resource utilization based on demand. The services also leverage application-side caching to reduce model loading times.

4.3 Extending σOS Procs

To represent and schedule inference workloads, we make several additions to the σOS `ProcProto` and `ProcEnvProto` (Section 5.2). The former is responsible for representing and managing the basic structure of a proc in the system, while the latter represents the environment in which a proc is running.

Table 4.1: Additions to σOS `ProcProto` and `ProcEnvProto`

Field	Proto	Description
`bool useMASched`	`ProcEnv`	Determines whether the proc will utilize the model-aware scheduler or the default scheduler.
`string remoteProxyID`	`ProcEnvProto`	The ID of the remote machine that the proc sends its inference request to.
`string model`	`ProcProto`	The language model (e.g., t5-small, facebook/bart-large-mnli) that the proc utilizes to run.
`repeated string preferredKernelID`	`ProcProto`	The ID(s) of the machine(s) that the proc prefers, if any.

Although inference procs are latency-critical – minimizing response time and making real-time decisions in ML applications is crucial – we introduce a third type of proc: model-aware (MA) procs. Inference tasks have distinct characteristics that distinguish them from general LC and BE tasks, so MA procs allow σOS to consider not only CPU and memory availability, but also additional factors like GPU usage, model size, and proxy request queue lengths.

4.4 Inference Proxy

The σOS inference client is responsible for handling different types of inference requests, enabling the system to perform ML tasks such as translation and image classification. The service, like most other σOS services, is defined using protocol buffers (proto3), and exposes a single RPC method, `Query`, which accepts an `InferenceRequest` and returns an `InferenceResult` (Section 5.3). Inference requests originate from a client that sends requests using the σOS RPC library (which uses gRPC).

`InferenceRequest` contains a `task_type` field that specifies the kind of inference task being performed. For example:

1. `TranslationRequest`: Contains the text to be translated along with the source and target language codes.

2. `ImageClassificationRequest`: Contains the image data to be processed for classification.

This design allows the system to extend easily to additional task types and provides flexibility for handling different models and inference workloads in a structured manner.

Since the Ray/RayServe code base is fairly large, it is quite infeasible to port it entirely to the σOS API. As such, the σOS inference server acts as a proxy to RayServe, which is run with its own container image. The inference server is initialized with an endpoint URL and utilizes an HTTP client to send requests to an external running RayServe instance. σOS configures the inference server with the necessary client and server connections, initializing it as a privileged kernel proc alongside others, such as `dialyproxyd`, `binfs`, `ux`, and `db`.

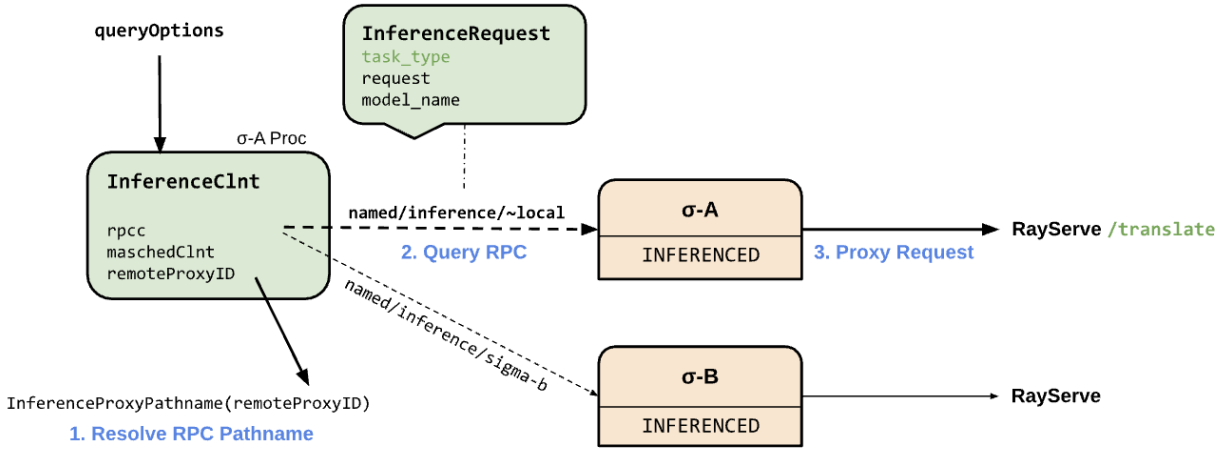

Figure 4.1: Internals of an inference proc that constructs an `InferenceClnt` and sends a local translation `Query` to the RayServe `/translate` endpoint.

A σOS inference proc constructs an `InferenceClnt`, which consists of fields like `RPCClnt`, `MASchedClnt`, and `RemoteProxyID`. Upon initialization, the inference client resolves the RPC pathname (1). If the proc's `RemoteProxyID` is set, the pathname resolves to `named/inference/<RemoteProxyID>`. Otherwise, the client sends requests locally via `named/inference/~local`. When the proc makes a `Query`, `InferenceClnt` constructs the appropriate gRPC proto to send based on input parameters, determines the correct RayServe endpoint (e.g. `/translate` for translation tasks, `/classify-image` for image classification tasks), and sends the request via RPC to the corresponding `InferenceServer` (2). The `InferenceServer` then proxies the request to the hosted RayServe instance via HTTP (3).

4.5 Model Colocation Scheduling

The first policy aims to colocate models and procs, routing requests to RayServe proxies that already have the necessary model cached. We make use of locality, spawning procs on the same machine where the desired proxy is located.

When a machine enters the cluster, it registers itself with both `LCSched` and `MASched`. Although only `LCSched` tracks each machine's available resources, `MASched` maintains a ledger of machines in the cluster to track other important metadata. The primary scheduling decisions reside within `MASched`, but important modifications are also made to the logic of enqueuing a request with `LCSched`.

`MASched` is responsible for efficiently managing model-related tasks and determining where inference tasks should be sent by maintaining the following structures (full structure in Section 5.4):

Field	Description
`modelToProxies`	A map that associates each model name with a list of kernel IDs that have the model cached.
`numOngoingRequests`	A map that associates each kernel ID with the number of ongoing requests at that machine's RayServe instance.
`qs`	A map that associates each realm with its queue, allowing for proper management of procs based on tenant isolation.
`lcschedclnt / mschedclnt`	Clients that interact with `LCSched` / `MSched`, respectively, to request resource allocations and scheduling actions.

Table 4.2: Select `MASched` fields

4.5.1 Lifecycle of a σOS Inference Request

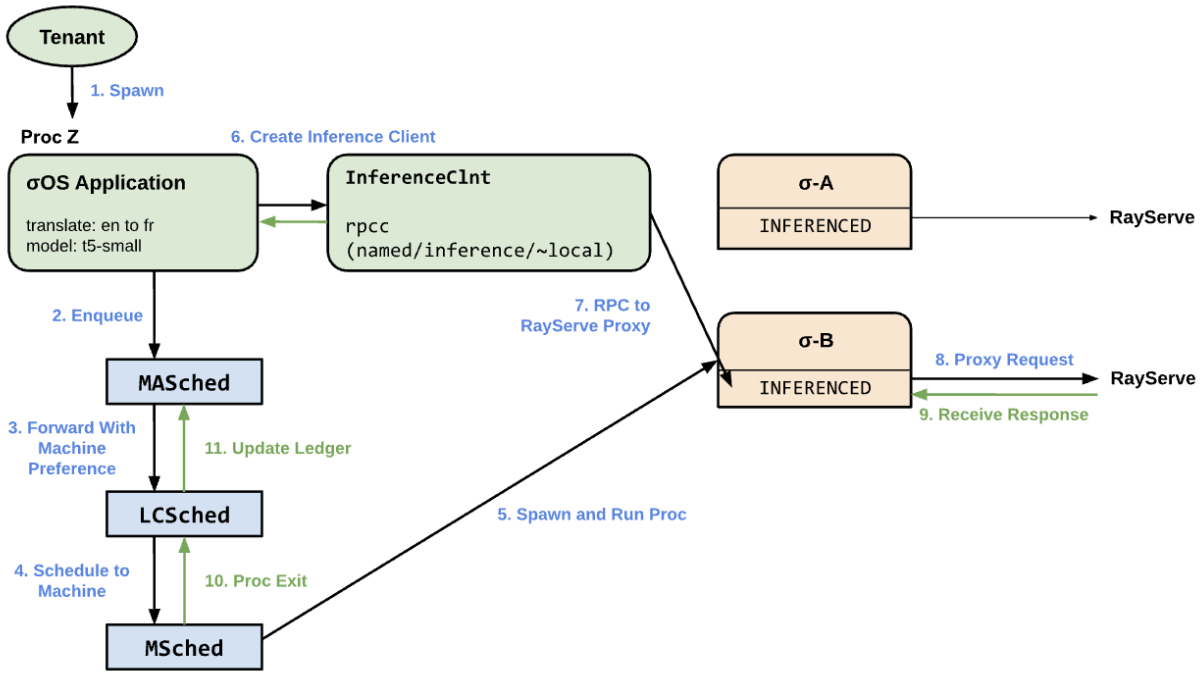

Figure 4.2: Model colocation policy: Lifecycle of a proc queued to `MASched`.

Inference requests in σOS begin with the parent proc calling `SetModel` on the inference proc. This method sets the proc's model field and classifies it as a `T_MA` proc, indicating that it requires specialized scheduling considerations for model colocation and resource management.

After the tenant spawns the proc (1), it is enqueued to `MASched` via an `Enqueue` RPC, remaining there until it is assigned a kernel ID via a channel (2). During this waiting period, `MASched` continuously tries to dequeue procs in the background. Once a proc reaches the front of the queue, the background scheduling goroutine dequeues it and forwards the proc's model requirements along with the kernel IDs of machines that the proc prefers to `LCSched` (3). A machine preference is defined as one that already has model already cached.

The scheduling flow within `LCSched` follows these steps:

1. Preferred Machine Check: If the proc has a set of preferred machines, `LCSched` tries to route the request to one of those first.
2. Resource Availability: In colocation without queueing, if there are either no preferred machines or none of the preferred machines can accommodate the request due to resource constraints (e.g. insufficient CPU, memory), `LCSched` attempts to schedule the proc on other machines in the cluster. In colocation with queueing, `LCSched` repeatedly attempts to schedule the proc on a preferred machine.
3. Routing: After determining the target machine, `LCSched` proceeds with sending the proc to the corresponding `MSched` for execution (4).

`MASched` spawns and runs the proc (5). As with Figure 4.1, the proc creates an inference client (6), which RPCs an inference server (7). The server sends an HTTP request to the corresponding RayServe instance (8). RayServe will either always access the model from its cache (with queuing), or potentially download and cache the new model (without queuing). The response makes its way back to the proc and original tenant (9), and system clean-up after execution remains unchanged. The proc eventually exits and frees up the resources that it was using (10), and `MASched` updates `modelsToProxies` with the kernel ID that the proc was executed on (11).

4.6 Centralized Model Registry Scheduling

Many real-world AI workloads consist of CPU-bound pre-processing and post-processing tasks (e.g. data pre-processing, feature extraction, output conversion). In these scenarios, it may not always be optimal to wait to spawn a proc on a machine that has a model cached, especially if the machine is bottlenecked by CPU pre/post-processing.

An alternative policy is to spawn a proc on any machine, locate a RayServe instance that has the correct model cached, and send a request over the network to that instance.

Consider the following scenario (Figure 4.3) with a 3-machine cluster: *Sigma-A*, *Sigma-B*, and *Sigma-C*. Each running RayServe instance has 1 replica, meaning it can process 1 request at a time. *Sigma-A* is unutilized and does not have any models cached. *Sigma-B* and *Sigma-C* are fully utilized by procs spawned on those machines and both have the t5-small model cached. There is 1 queued request at *Sigma-B* and 3 queued requests at *Sigma-C*.

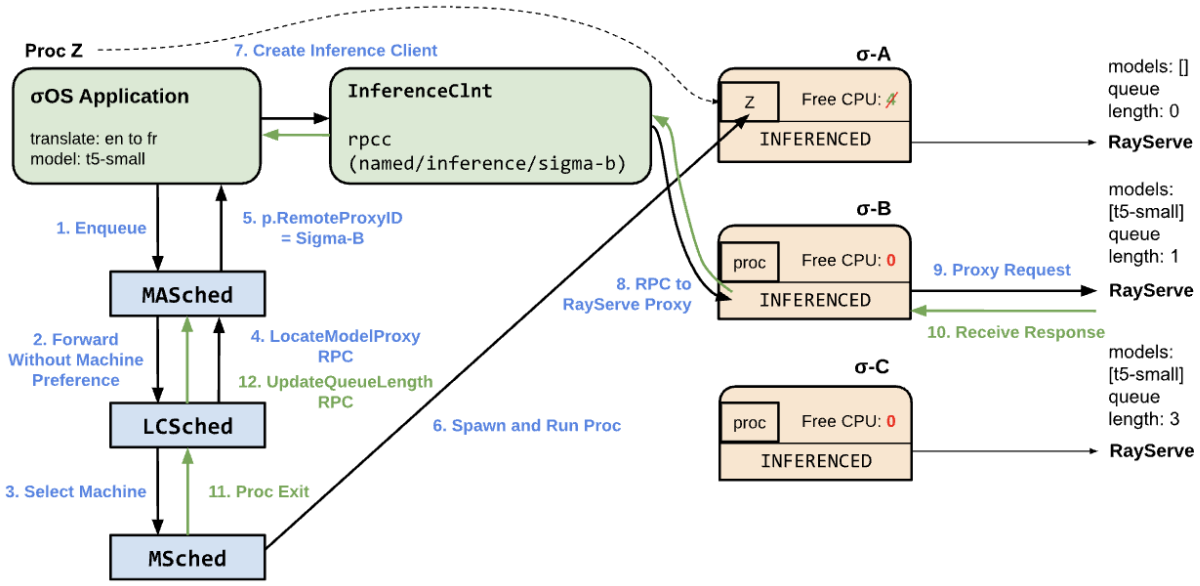

Figure 4.3: Centralized model registry policy: Lifecycle of a proc queued to `MASched`.

Proc Z is a t5-small translation query that performs 5 seconds of post-processing. When the tenant tries to spawn Z, the following steps occur:

- Z gets enqueued in `MASched` (1), which forwards the request without a machine preference to `LCSched` to allow a randomly available machine to be selected (2).
- `LCSched` selects the only available kernel, σ-A (3), and makes a `LocateModelProxy` RPC call to `MASched` (4).
- `MASched` looks up t5-small in its model registry and finds that σ-B and σ-C both have the model cached. It sets the proc's `RemoteProxyID` to σ-B (5) after seeing that σ-C's queue has exceeded `targetOngoingRequests` (which defaults to 3); it also increments σ-B's number of ongoing requests to account for the new incoming query from proc Z.
- σ-A's `MSched` will now spawn and run the proc on σ-A (6), at which point Z creates an inference client with an RPC client referring to σ-B's inference server (7).
- When the inference client makes the `Query` RPC call to the server, the RPC client will resolve the correct pathname to the remote proxy located on σ-B (8), which sends the request to the RayServe instance (9).
- After Z waits its turn to get processed, RayServe sends the response back to the inference client (10), at which point post-processing begins on σ-A.
- After Z finishes, the proc exits (11) and makes an `UpdateQueueLength` RPC (12) to `MASched`, decrementing `numOngoingRequests`[1] to reflect that σ-B has one less request in its queue.

In the case where `MASched` is unable to locate a remote proxy, if either the model has never been seen or all proxies have too many ongoing requests, the client will query its local RayServe proxy to download the model and process the request. The machine that the proc is spawned to will then be added to `modeltoProxies` in the `LocateModelProxy` RPC.

[1]Drawing inspiration from RayServe's built-in autoscaling policy, we track the number of ongoing requests at each instance in order to distribute load as necessary.

Chapter 5

Implementation

This chapter details several protos, structs, and RPCs that are most critical to the implementation of the σOS model-aware scheduler.

5.1 RayServe Application

The RayServe application is deployed in a Docker container, with a shell script handling the deployment of the cluster. Models are also cached application-side so that newly spun up replicas do not need to reinitialize each pipeline.

The `Translator` class is defined as a RayServe deployment and specifies autoscaling parameters such as a minimum of 1 replica, a maximum of 4 replicas, and a target of 2 ongoing requests per replica.

```
@serve.deployment(
    autoscaling_config={
        "min_replicas": 1,
        "max_replicas": 4,
        "target_num_ongoing_requests_per_replica": 2,
    })
class Translator:
```

Each endpoint further defines an asynchronous user-defined method `__call__`, which processes incoming HTTP requests and returns the response, that is invoked in RayServe to handle specific tasks like translation.

```
async def __call__(self, http_request: Request) -> str:
    text: str = await http_request.json()
    result = self.translate(text)
    return result
```

5.2 Proc

The `ProcProto` message defines the structure of a proc within the system, including various parameters like the environment configuration (`ProcEnvProto`), command arguments,

environment variables, resource requirements (CPU, GPU, memory), and model-specific information.

```
message ProcEnvProto {
  // omitted fields, unchanged from before
  bool useMASched = 31;
  string remoteProxyID = 32;
}

message ProcProto {
  ProcEnvProto procEnvProto = 1;
  repeated string args = 2;
  map<string, string> env = 3;
  uint32 typeInt = 4;
  uint32 gpuTypeInt = 5;
  uint32 mcpuInt = 6;
  uint32 memInt = 7;
  uint32 gpuInt = 8;
  string model = 9;
  repeated string preferredKernelID = 10;
}
```

5.3 Inference Proxy

5.3.1 Proto

The `InferenceRequest` message defines the structure of a request sent to the inference proxy, specifying the task type, corresponding request data, and model name to identify which model should be used for the task.

```
enum TaskType {
  UNSPECIFIED = 0;
  TRANSLATION = 1;            // Translation task
  IMAGE_CLASSIFICATION = 2;   // Image classification task
}

message InferenceRequest {
  TaskType task_type = 1;
  oneof request {
    TranslationRequest translation_request = 2;
    ImageClassificationRequest image_classification_request = 3;
  }
  string model_name = 4;
}
```

5.3.2 RPC API

The inference proxy RPC method, `Query`, is essential for interacting with RayServe to perform inference.

RPC	Description
Query(InferenceRequest) returns (InferenceResult)	Sends an HTTP request to a RayServe endpoint determined by the type of input task; returns the response.

Table 5.1: Inference Proxy RPC Methods and Definitions

5.4 MASched (Model-Aware Scheduler)

5.4.1 Struct

The `MASched` struct defines the state and resources to manage the scheduling logic and coordination of inference tasks. It includes a mutex for synchronization, a condition variable, mappings for model-to-proxy relationships, ongoing request counts, queues for task scheduling, and clients for interfacing with the scheduling systems.

```
type MASched struct {
  mu                  sync.Mutex
  cond                *sync.Cond
  modelToProxies      map[string][]string
  numOngoingRequests  map[string]int
  qs                  map[sp.Trealm]*queue.Queue[string, chan string]
  lcschedclnt         *lcschedclnt.LCSchedClnt
  mschedclnt          *mschedclnt.MSchedClnt
}
```

5.4.2 RPC API

The `MASched` RPC methods handle various operations related to task scheduling, helping manage resources and efficiently assign tasks across available machines.

RPC	Description
Enqueue(EnqueueRequest) returns (EnqueueResponse)	Enqueues a proc to `MASched`; returns the ID of the machine that the proc is spawned on.
RegisterMSched(RegisterMSchedRequest) returns (RegisterMSchedResponse)	Registers a machine with `MASched` by adding its ID to `numOngoingRequests`; returns nothing.
LocateModelProxy(LocateModelProxyRequest) returns (LocateModelProxyResponse)	Locates the first machine with the given proc's model cached that has less than `targetOngoingRequests` ongoing requests; returns ID of that machine.
UpdateQueueLength(UpdateQueueLengthRequest) returns (UpdateQueueLengthResponse)	Decrements the number of ongoing requests of the given machine ID; returns nothing.

Table 5.2: `MASched` RPC Methods and Definitions

Chapter 6

Evaluation

This chapter evaluates the performance of three scheduling policies: *Model Colocation Without Queuing*, *Model Colocation With Queuing*, and *Centralized Model Registry* across various workloads. The primary objective is to identify the workload characteristics that determine which policy minimizes end-to-end latency. Throughout this chapter, we refer to the average initialization time and model inference time of each T5 model size as outlined in Figure 3.2.

Note that we focus on T5, an encoder-decoder transformer model known for its versatility in text-to-text NLP tasks such as translation, summarization, and question answering. T5 is also important in modern AI systems – it is used as a text encoder in state-of-the-art multimodal models like Stable Diffusion 3 [16]. Importantly, T5's wide range of model sizes, from 60M to 11B parameters, allows us to explore tradeoffs between model size and performance in diverse workloads.

Specifically, this chapter seeks to answer the following questions:

1. How does model colocation scheduling compare to σOS's default scheduling (Section 2.3.2) in terms of reducing end-to-end workload latency and minimizing cold starts? (Section 6.2)
2. After how many requests does the effect of cold starts become amortized? (Section 6.3)
3. What is a workload for which each scheduling policy results in lower end-to-end latency than the others? (Section 6.4)
4. Given factors such as model inference time, cold start time, network latency, and the combined pre/post processing time, which scheduling policy is most appropriate for a given workload? (Section 6.5)

6.1 Experimental Setup

In each of the following experiments, σOS is deployed on a cluster of 8 c220g5 nodes running Ubuntu 22.04.2 LTS. Each machine is equipped with two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz, 192GB of memory, and 2 10Gb Intel X520-DA2 NICs [17]. To induce

contention for resources and force σOS to distribute microservices across multiple hosts, all but 4 CPUs are disabled on each machine.

6.2 Simple Inference Workload

The simple benchmark workload consists of 10 English-to-French translation requests, each translating the 5-word phrase "quick brown fox jumps over" using the same model. Each request is spawned sequentially as a σOS proc. We allocate one core to each proc to match RayServe's default CPU allocation per replica. We run three versions of this workload, that use t5-small, t5-base, and t5-large, respectively.

We expect model colocation to yield a significant improvement in latency and number of cold starts over the default σOS scheduler, since it is able to route each request after the first to the RayServe instance that already has the model cached. Without being model-aware, the σOS default scheduler routes requests randomly and forces several machines to download the same model, incurring the cold start cost.

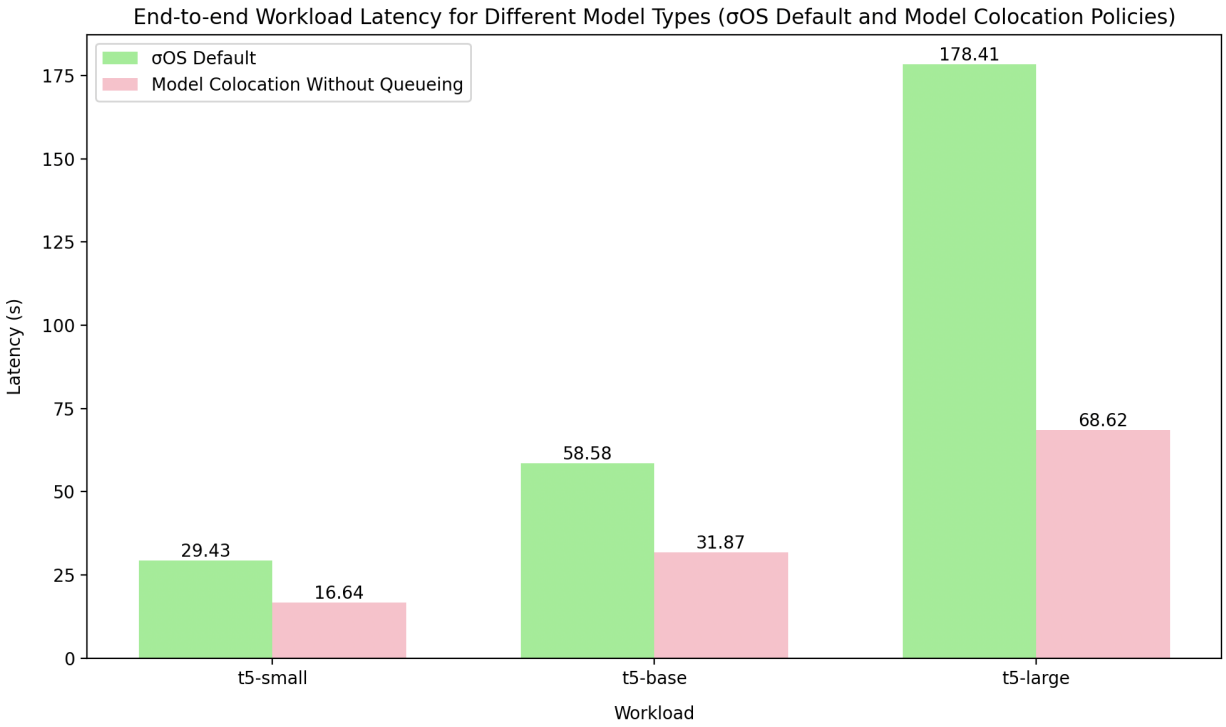

Figure 6.1: End-to-end latency of the 10 translation request workload for each of three model types (t5-small, t5-base, t5-large), comparing the default σOS scheduler to model colocation without queuing.

Although the number of requests processed per second slows for larger models in both policies, the difference between the end-to-end latencies of the two schedulers increases. Model colocation for the t5-large workload becomes nearly two minutes faster than the σOS default scheduler, which can be attributed to the impact of each cold start being larger felt. T5-large

has a ≈ 40 second difference between initialization and inference time compared to ≈ 2 seconds for t5-small, as detailed in Section 3.3.

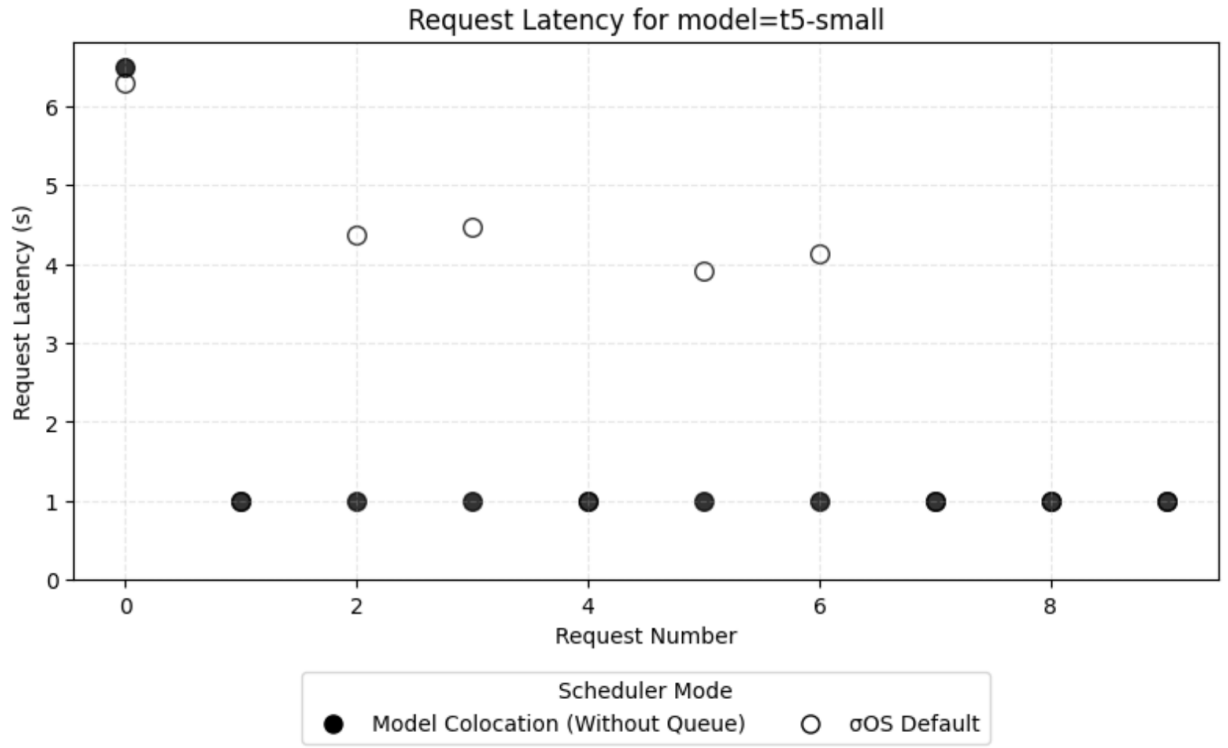

Figure 6.2: End-to-end latency of each σOS proc for model colocation (without queue) and default σOS schedulers. All procs use the t5-small model. *Any default σOS point that is not visible indicates overlap between the two schedulers.

To visualize the number of cold starts, we plot the end-to-end latency of each σOS proc, from when it is first enqueued to the σOS scheduler to when it finishes execution and exits, using the t5-small workload as an example, in Figure 6.2. In both cases, there is a noticeable spike in the first request to the t5-small model of ≈ 6 seconds due to the initial cold start. With model colocation without queuing, every subsequent proc benefits from warm start latency (≈ 1 second) being routed to the RayServe proxy with an already cached t5-small model. In contrast, procs scheduled in the default scheduling policy are routed randomly, leading to spikes each time a new RayServe instance downloads the same model (requests 2, 3, 5, and 6).

In total, 5 machines download t5-small, meaning the default scheduling policy leads to 5 cold starts (as well as 5 cold starts for t5-base and 7 cold starts for t5-large) compared to 1 cold start in each workload with colocation. Each cold start leads to a ≈ 3 second increase in latency for an overall increase of ≈ 12 seconds, which accounts for the difference between the end-to-end t5-small latencies (29.43 - 16.64 seconds) from Figure 6.1.

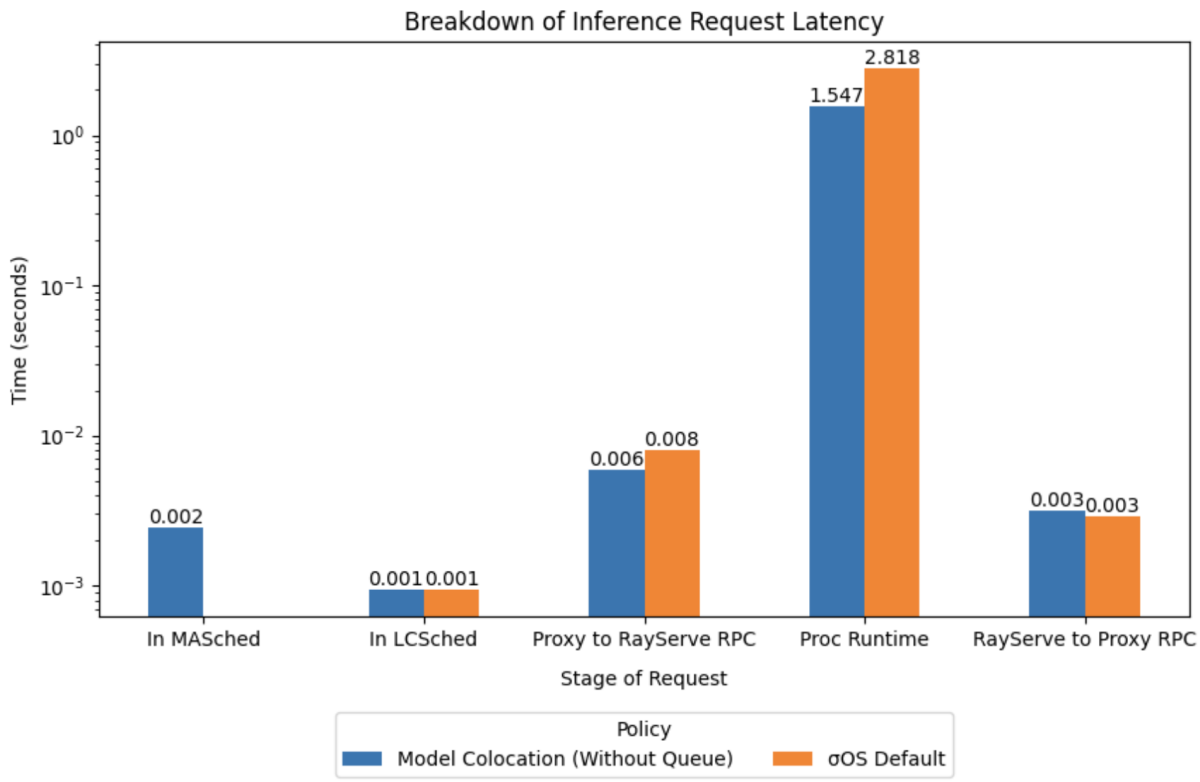

Figure 6.3: Simple Inference Workload: Average latency across all 10 requests, broken down by stage, for model colocation (without queue) and default σOS schedulers. Plotted on a log scale.

The above figure breaks down inference requests into the following stages:

1. In `MASched`: time from initially being enqueued to `MASched` to when it receives the ID of the machine the proc was spawned on, excluding the time spent in `LCSched`
2. In `LCSched`: time from initially being enqueued to `LCSched` to when the proc is successfully scheduled
3. Proxy to RayServe RPC: time from when the inference client sends the HTTP request to RayServe to when RayServe first calls the user-defined method (Section 5.1) and starts processing the request, which includes any time spent queuing at the RayServe replica
4. Proc Runtime: time from when proc is first started to when it exits
5. RayServe to Proxy RPC: time from when RayServe sends the request back to the proxy to when the proxy receives the response

Figure 6.3 illustrates that Proc Runtime, which encompasses total request processing time, accounts for most of the performance difference. The default scheduler overall runtime is higher because of additional cold starts, when the same model is downloaded by multiple

RayServe instances. Importantly, the 0.002 second overhead incurred per request by `MASched` is negligible compared to the resulting difference in proc runtime. Since model colocation scheduling reduces both end-to-end workload latency and the number of cold starts, we conclude that it is worthwhile to be model-aware.

6.3 Cold Start Amortization

Utilizing the same workload from Section 6.2, we can calculate the break-even/cold start amortization point, where the average latency per request for our two policies become almost identical. For a cluster of 8 machines, we experience 1 unavoidable cold start and at most 7 additional cold starts, in the worst case. Each t5-small initialization incurs an additional cost of ≈ 3 seconds – ≈ 21 seconds in total. If we claim that the break-even point occurs when the difference in average latency between the two policies is less than 0.01 seconds, then it should occur around $21/0.01 = 2100$ requests. We verify this by running the same workload from before but with 2100 sequential requests for t5-small, which yield latencies of 2097.30 (Model Colocation Without Queuing) and 2115.19 seconds (Default Scheduler), or average latencies of ≈ 1 second.

For t5-base, each cold start incurs an additional cost of ≈ 6 seconds – 42 seconds in total, meaning the break-even point hovers around 4200 requests. For t5-large, each cold start incurs an additional cost of ≈ 17 seconds – 119 seconds in total, and a breakeven point of 11900 requests. This trend is expected, since larger models require longer initialization times and make cold starts much more expensive to incur (Section 3.3).

6.4 Individual Policy Wins

6.4.1 Model Colocation Without Queuing Win

This workload consists of 2 English-to-French translation requests, each translating the 5-word phrase "quick brown fox jumps over" using t5-small. After performing the translation, each proc sleeps for 5 seconds to simulate how pre/post-processing tasks can block CPU resources and prevent them from being allocated to other tasks, even after the main inference task completes. The requests are spawned concurrently as σOS procs, each allocated the full 4 cores (to simulate full CPU reservation). To simulate high network latency, we also sleep for 5 seconds before querying any remote RayServe proxy.

We expect model colocation without queuing to perform the best. The centralized model registry scheduler would be bottlenecked by the high network latency, and model colocation with queuing would be bottlenecked by the pre/post-processing time, both of which are higher than the cost of cold start. Model colocation without queuing would excel – it would spawn the first proc on a random machine and immediately spawn the second proc to a second machine since the first is fully reserved, and the procs would be executed in parallel.

Workload	Model Colocation (Without Queue)	Model Colocation (With Queue)	Centralized Model Registry
2 concurrent requests to t5-small	**12.38s**	18.80s	14.41s

Table 6.1: Model Colocation Without Queuing Win: End-to-end Latencies

The results indicate that centralized model registry is ≈ 2 seconds slower than model colocation without queuing, which can be explained by the 5 second network latency and ≈ 3 second t5-small cold start time. Instead of waiting 5 seconds to send the request over the network, the scheduler can save 2 seconds by immediately spawning the proc on a cold machine and incurring the cold start time of 3 seconds. We also see that model colocation with queuing is ≈ 6 seconds slower than its counterpart without queuing, which can be explained by Figure 6.4.

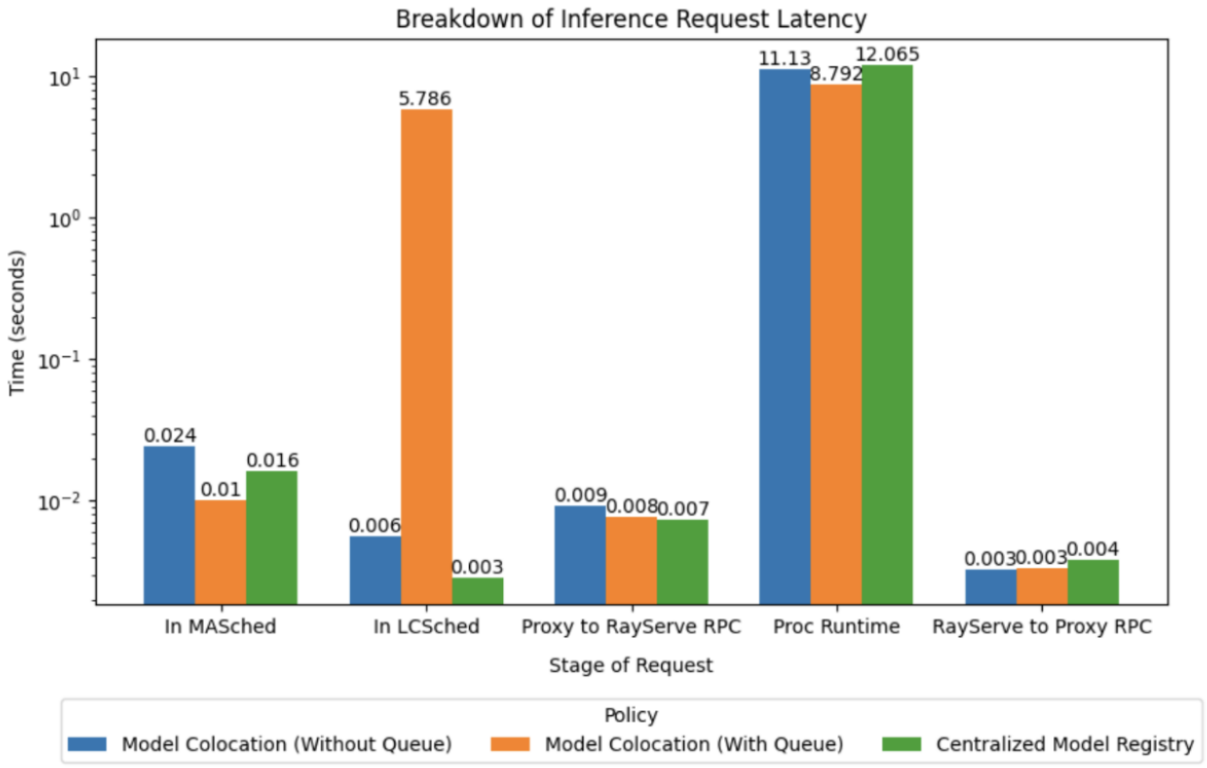

Figure 6.4: Model Colocation Without Queuing Win: Average latency across the two requests, broken down by stage, for model colocation (without queue), model colocation (with queue), and centralized model registry schedulers. Plotted on a log scale.

Since the second proc must wait for the first proc to finish executing entirely in colocation without queuing, it spends an extra ≈ 5 seconds (post-processing time) waiting in `LCSched` to be scheduled. Model colocation with queuing does have a shorter average proc runtime,

since its counterpart without queuing leads to an extra model download, and centralized model registry's expensive network data transfer is included within proc runtime.

Since model inference time remains constant for all three policies but CPU-bound pre/post-processing is lengthy and network transfer is expensive, it becomes more beneficial to incur the extra cold start time and spawn the proc without queuing.

6.4.2 Model Colocation With Queuing Win

This workload consists of 3 English-to-French translation requests, each translating the 5-word phrase "quick brown fox jumps over" using t5-small, with no pre/post-processing. We utilize the first request to pre-warm one RayServe proxy before processing the next two requests. Since the first cold start time is unavoidable, this allows us to realistically simulate the best way to utilize model-awareness in the middle of a stream of requests. The next two requests are spawned concurrently as σOS procs, each allocated the full 4 cores in order to simulate full CPU reservation and ensure that a second proc cannot immediately be spawned on the same machine. To simulate high network latency, we again sleep for 5 seconds right before querying a remote RayServe proxy.

We expect model colocation with queuing to perform the best. The centralized model registry scheduler would again be bottlenecked by the high network latency, and model colocation without queuing would be bottlenecked by the extra cold start time. Model colocation with queuing would spawn the first proc on the warm machine and try to spawn the second proc on the same machine. Since the machine is fully utilized, the second proc would wait the duration of the model inference time to be scheduled, which is faster than both the cold start and network latency of the other two policies.

Workload	**Model Colocation (Without Queue)**	**Model Colocation (With Queue)**	**Centralized Model Registry**
1 warming request, 2 concurrent requests to t5-small	11.18s	**9.65s**	15.19s

Table 6.2: Model Colocation With Queuing Win: End-to-end Latencies

The results indicate that model colocation without queuing is ≈ 2 seconds slower, which is approximately the difference between the t5-small cold start and model inference time. Similarly, centralized model registry is ≈ 5.5 seconds slower, which can be attributed to the difference between network latency and model inference time.

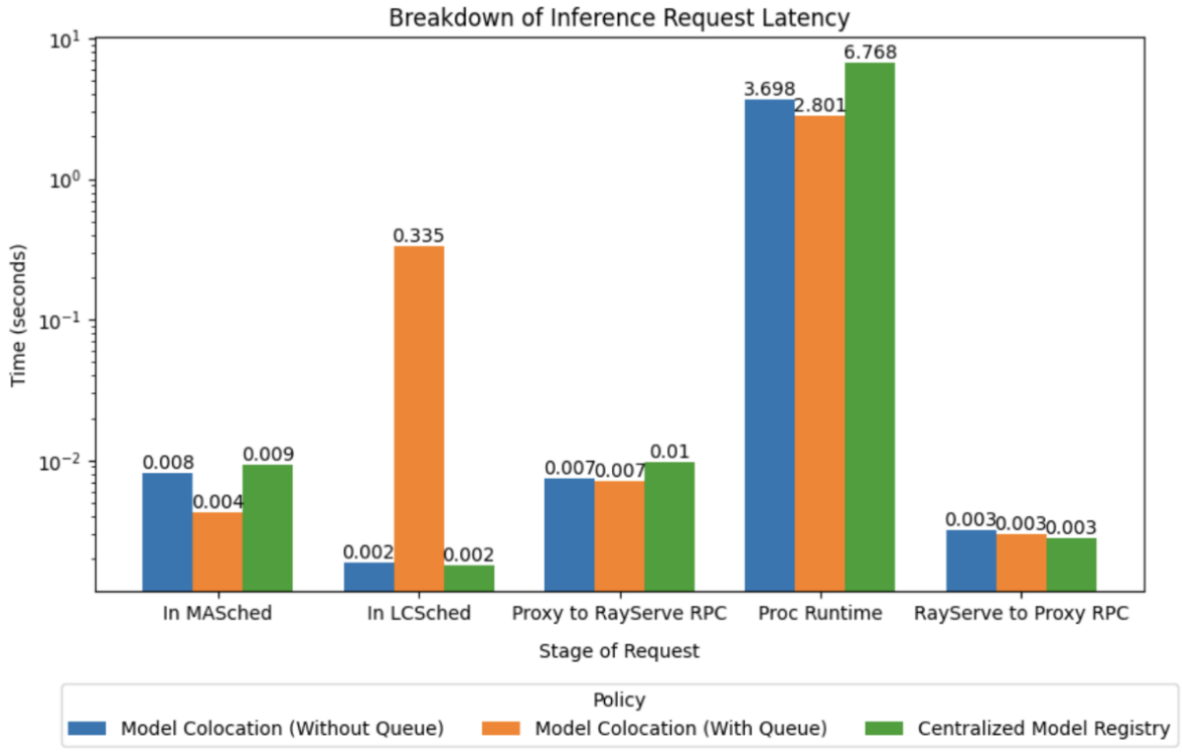

Figure 6.5: Model Colocation With Queuing Win: Average latency across the three requests, broken down by stage, for model colocation (without queue), model colocation (with queue), and centralized model registry schedulers. Plotted on a log scale.

Although colocation with queuing spends longer in `LCSched` due to needing to wait before being spawned to the same machine, its proc runtime is shorter by far. Note that another workload for which we would expect model colocation with queuing to win is one with no pre/post-processing, but using a model with a cold start time much longer than inference time (i.e. t5-large).

When model inference time is short compared to cold start and network transfer time, it becomes more beneficial to incur the queuing time and wait to spawn the proc on a warm machine.

6.4.3 Centralized Model Registry Win

This workload similarly consists of 3 English-to-French translation requests, each translating the 5-word phrase "quick brown fox jumps over" using t5-small, with 5 seconds of pre/post-processing. We again utilize the first request to pre-warm one RayServe proxy before processing the next two requests. Since each request is only a few bytes, the network transfer time is low.

We expect the centralized model registry to perform the best since it is both able to take advantage of the low network latency and avoid waiting for pre/post-processing. Both procs will be spawned on different machines and query the same RayServe that already has t5-small cached, so that the only delay comes from the second request waiting for the replica

to process the first. Model colocation without queuing would again be bottlenecked by the extra cold start time, and model colocation with queuing would be bottlenecked by time spent waiting for the first proc to complete its CPU-bound processing.

Workload	**Model Colocation (Without Queue)**	**Model Colocation (With Queue)**	**Centralized Model Registry**
1 warming request, 2 concurrent requests to t5-small	23.09s	25.27s	**20.34s**

Table 6.3: Centralized Model Registry Win: End-to-end Latencies

Model colocation without queuing is ≈ 3 seconds slower, which is approximately the difference between cold start (3 seconds) and model inference (1 second) time. Model colocation with queuing is ≈ 5 seconds slower, which is approximately the pre/post-processing time (5 seconds).

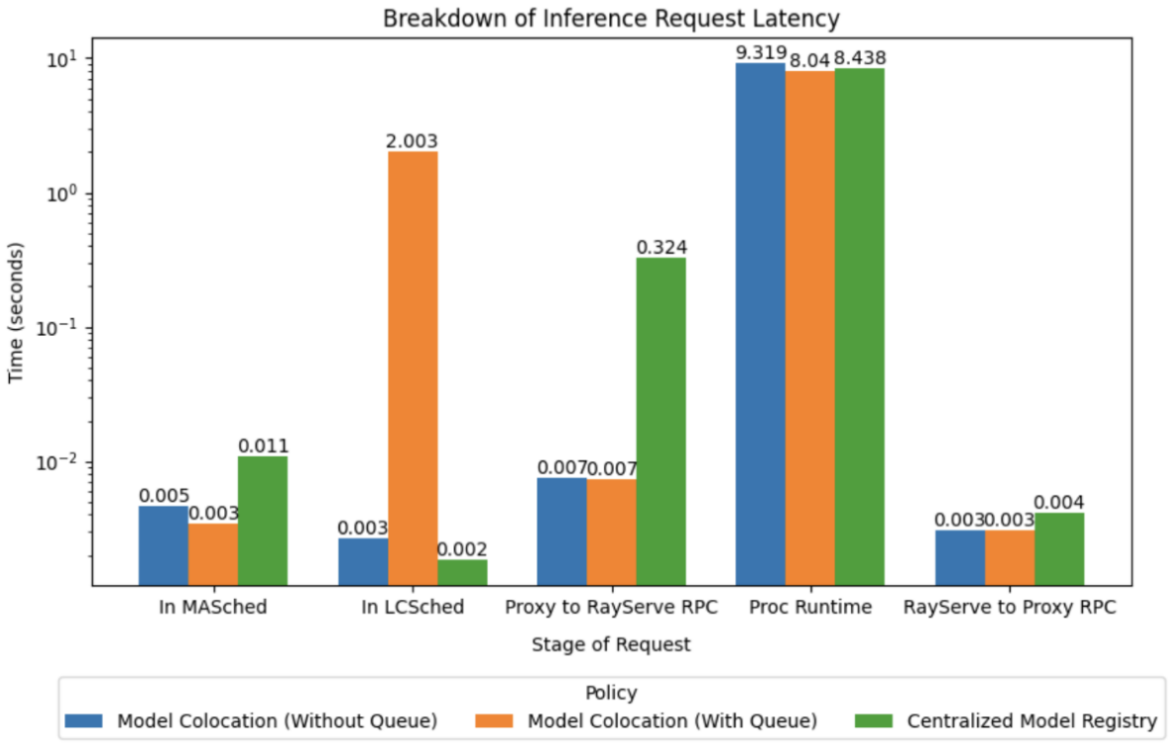

Figure 6.6: Centralized Model Registry Win: Average latency across the three requests, broken down by stage, for model colocation (without queue), model colocation (with queue), and the centralized model registry schedulers. Plotted on a log scale.

We again see that colocation with queuing spends more time in `LCSched`. Centralized Model Registry spends more time in Proxy to RayServe RPC due to the RayServe replica only being able to process one request at a time. Due to fast network transfer in comparison

to cold start and pre/post-processing time, it is more beneficial to send requests over the network to warm RayServe instances.

6.5 Analytical Model

Experiment 6.2 shows that model-aware schedulers grant procs an advantage in being able to utilize already cached models and avoiding expensive cold starts that become even more costly as models and workloads grow in complexity.

Experiments 6.4.1, 6.4.2, and 6.4.3 each demonstrate a workload in which a different policy should be used above the others, and the importance of the following factors: cold start time T_{CS}, model inference time T_I, pre/post-processing time T_P, and network transfer time T_N.

Assume that each RayServe instance consists of 1 replica, there exists an available machine on which to spawn a proc that would incur a cold start cost, $T_{I_{avg}}$ is the average model inference time for an existing request to that model, $T_{P_{avg}}$ is the average pre/post-processing time for an existing request to that model, and n is the maximum number of ongoing requests at any warm proxy.

Given a request, the end-to-end latency can be approximated as follows:

1. $T_{COLOCATION} = \max(T_{CS} + T_I + T_P, n \cdot (T_{I_{avg}} + T_{P_{avg}}))$ where the first term represents the total cost of processing the request on a cold machine and the second term represents the total time it takes to process n existing requests at one proxy.

2. $T_{COLOCATION\_WITH\_QUEUING} = (n+1) \cdot (T_{I_{avg}} + T_{P_{avg}})$ for n existing requests and the newest request.

3. $T_{REGISTRY} = (T_N + T_I + T_P) + n \cdot T_{I_{avg}}$ where the first term represents the total cost of processing the newest request and the second term represents the total time it takes to perform inference on the n existing requests at one proxy.

In conclusion, given that the optimal policy is highly dependent on workload and model characteristics, dynamically selecting the scheduling policy based on cold start time, model inference time, pre/post-processing time, and network transfer time is crucial.

Chapter 7

Conclusion and Future Work

7.1 Limitations

While the model-aware scheduling policies reduce both latency and the number of cold starts for many workloads, there are some limitations to the current implementation. Notably, cold start time introduces an unavoidable overhead for the very first request to a new model, though this overhead becomes amortized after a certain number of requests (Section 6.3). For systems that consistently use the same models, a more effective approach might involve simply caching all models in memory.

Additionally, the current scheduler implementations assume a static set of available resources for model execution, without considering fluctuating resource demands or large-scale dynamic workloads. This limiation could lead to performance degredation under high load, particularly when there is contention between tenants with varying computational needs. Furthermore, the current design only allows procs to use a single model. In cases where a proc requires multiple models cached on different machines, the model-aware scheduler could schedule the proc on the machine with the largest model.

7.2 Future Work

Future work could focus on enhancing the system's scalability and performance through more efficient memory management. Techniques such as dynamic memory reclamation and garbage collection for models could help improve resource utilization. Additionally, exploring alternative scheduling policies, such as priority-based or hybrid approaches, could address resource contention and improve fairness in multi-tenant scenarios. GPU-specific optimizations, including model eviction strategies and GPU-aware scheduling, would also be valuable to explore, especially in mitigating CPU-GPU bottlenecks and improving the efficiency of data transfers between devices.

7.3 Conclusion

This work extends σOS by integrating GPU support and RayServe, introducing the first ML workloads, and developing two novel model-aware scheduling policies to optimize end-to-end latency and reduce the number of cold starts in multi-tenant environments.

An evaluation of σOS revealed a 50% average reduction in latency and 4-5 fewer cold starts on an 8-machine cluster for several translation workloads. Model-awareness helps reduce the number of cold starts by leveraging cached models and exploiting locality for certain workloads. The schedulers also demonstrate flexibility in proxying requests remotely especially when scheduling CPU-bound pre/post-processing tasks to be parallelized with inference. Importantly, these experiments provide insight into the best policy to use based on factors such as model inference time, cold start time, and network latency.

Through the design, implementation, and evaluation of two model-aware schedulers, this thesis takes a step towards improving the efficiency and scalability of multi-tenant ML inference systems.

References

[1] Ray Developers. *Ray Serve Documentation*. 2025. URL: https://docs.ray.io/en/latest/serve/index.html.

[2] Vannevar Labs. *From Data to Defense: Real-Time Inference with Ray Serve*. 2025. URL: https://blog.vannevarlabs.com/from-data-to-defense-realtime-inference-with-ray-serve-3fa3fefc460a.

[3] A. Szekely, A. Belay, R. Morris, and M. F. Kaashoek. "Unifying serverless and microservice tasks with SigmaOS". *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2024). URL: https://pdos.csail.mit.edu/papers/sigmaos:sosp24.pdf.

[4] MIT PDOS. *SigmaOS GitHub Repository*. 2025. URL: https://github.com/mit-pdos/sigmaos.

[5] Go Developers. *Go Programming Language*. 2025. URL: https://go.dev/.

[6] G. Agarwal. *Ten Ways to Serve Large Language Models: A Comprehensive Guide*. 2025. URL: https://gautam75.medium.com/ten-ways-to-serve-large-language-models-a-comprehensive-guide-292250b02c11.

[7] Ray Developers. *Ray Serve Architecture*. 2025. URL: https://docs.ray.io/en/latest/serve/architecture.html.

[8] P. Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2018). URL: https://www.usenix.org/system/files/osdi18-moritz.pdf.

[9] NebiUs. *Slurm vs Kubernetes*. 2023. URL: https://nebius.com/blog/posts/model-pre-training/slurm-vs-k8s.

[10] NVIDIA. *Triton Inference Server User Guide*. 2023. URL: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html.

[11] Kubernetes Authors. *Kubernetes Documentation*. 2023. URL: https://kubernetes.io/.

[12] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. "Borg, Omega, and Kubernetes". *ACM Queue* 14.1 (2016).

[13] Amazon Web Services. *T3 Instances - Amazon EC2*. 2025. URL: https://aws.amazon.com/ec2/instance-types/t3/.

[14] HuggingFace. *Transformers Pipeline Documentation*. 2025. URL: https://huggingface.co/docs/transformers/en/main_classes/pipelines.

[15] HuggingFace. *SafeTensors Documentation*. 2025. URL: https://huggingface.co/docs/safetensors/en/index.

[16] Stability AI. *Stable Diffusion 3 Research Paper*. 2025. URL: https://stability.ai/news/stable-diffusion-3-research-paper.

[17] CloudLab Developers. *CloudLab Hardware Specifications*. 2025. URL: https://docs.cloudlab.us/hardware.html.