Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency

Arpan Gujarati  
MPI-SWS  
arpanbg@mpi-sws.org

Björn Brandenburg  
MPI-SWS  
bbb@mpi-sws.org

Sameh Elnikety  
Microsoft  
samehe@microsoft.com

Yuxiiong He  
Microsoft  
yuxhe@microsoft.com

Kathryn S McKinley  
Google  
ksmckinley@google.com

Abstract

Machine Learning (ML) service platforms allow clients to train and deploy ML models as web services. A key challenge for providers is to guarantee response time Service Level Agreements (SLAs) for inference workloads while minimizing resource waste. Swayam is a fully distributed autoscaling framework that exploits characteristics of production ML inference workloads, to deliver on the dual challenge of resource efficiency and SLA compliance. Our key contributions over prior work are (1) model-based autoscaling that takes into account SLAs and ML inference workload characteristics, (2) a distributed protocol that uses partial load information and prediction at frontends to provision new service instances, and (3) a backend self-decommissioning protocol for service instances. We evaluate Swayam on 15 popular services that were hosted on a production ML as a service platform, for the following service-specific SLAs: for each service, at least 99% of requests must complete within the response time threshold. Compared to a clairvoyant autoscaler that always satisfies the SLAs (i.e. even if there is a burst in the request rates), Swayam improves resource utilization by up to 27%, while meeting the service-specific SLAs over 96% of the time during a three hour window. The production platform has been using a Swayam-based framework for months to host over 100,000 services.

1 Introduction

The increasing effectiveness of Machine Learning (ML) and the advent of cloud services is producing rapid growth of Machine Learning as a Service (MLaaS) platforms such as IBM Watson, Google Cloud Prediction, Amazon ML, and Microsoft Azure ML [1, 2, 5, 7]. Clients of these platforms create and train ML models, and publish them as web services. End-users and client applications then query these trained models with new inputs and the services perform inference (prediction) [24]. For instance, a mobile fitness application collects sensor data and sends a request to an inference service to predict the distance (prediction) [24]. For instance, a mobile fitness application collects sensor data and sends a request to an inference service to predict the distance run by a user given the time spent running. ML inference services are stateless and bound by Service Level Agreements (SLAs) such as “at least 99% of requests must complete within 500ms.” SLA violations typically carry an immediate financial penalty (e.g., clients are not billed for non-SLA-compliant responses) and must thus be avoided as much as possible.

To meet the service-specific SLAs, MLaaS providers may thus be tempted to take a conservative approach and over-provision services with ample hardware infrastructure, but this approach is impractical. Since the services and the overall system exhibits highly fluctuating load, such over-provisioning is economically not viable at cloud scale. For example, while tens of thousands of services may be deployed, many fewer are active simultaneously, and the load on active services often varies diurnally.

Resources must thus be allocated dynamically in proportion to shifting demand. This, however, represents a major challenge for MLaaS providers because the processing time of each request is typically in the range of tens to hundreds of milliseconds, whereas the time required to deploy a fresh instance of an ML service to handle increasing load is significantly larger (e.g., a few seconds)—a slow reaction to load changes can cause massive SLA violations.

In summary, naive over-provisioning approaches are economically not viable in practice, whereas naive, purely reactive resource provisioning heuristics affect SLA compliance, which decreases customer satisfaction and thus also poses an unacceptable commercial risk. A smarter, predictive autoscaling solution is thus required to hide the provisioning latency inherent in MLaaS workloads.

Prior autoscaling and prediction solutions for stateless web services [10, 12, 16, 19, 31] typically perform centralized monitoring of request rates and resource usage to make accurate request rate predictions, and then proactively autoscale backend service instances based on the predicted rates. However, due to a number of mismatches in assumptions and target workloads, these approaches do not transfer well to the MLaaS setting. For one, they assume that setup times are negligible, whereas ML backends incur significant provisioning delays due to large I/O operations while setting up...
new ML models. Further, the lack of a closed-form solution to calculate response-time distributions even for simple models with setup times makes the problem more challenging. Prior solutions also assume a centralized frontend. In contrast, large ML services must be fault-tolerant and handle cloud-scale loads, which necessitates the use of multiple independent frontends.

In this paper, motivated by the needs and scale of a rapidly growing commercial MLaaS platform, we propose Swayam, a new decentralized, scalable autoscaling framework that ensures that frontends make consistent, proactive resource allocation decisions using only partial information about the workload without introducing large communication overheads.

Swayam tackles the dual challenge of resource efficiency and SLA compliance for ML inference services in a distributed setting. Given a pool of compute servers for hosting backend instances of each service, Swayam ensures an appropriate number of service instances by predicting load, provisioning new instances as needed, and reclaiming unnecessary instances, returning their hardware resources to the global server pool (Fig. 1).

Swayam is fully distributed, i.e., frontends execute the protocol in parallel without any mutual coordination. Each frontend obtains the number of active frontends \( F \) gossiped from backends and combines it with its local load to independently predict the global request rate and estimate the total number of backends required to meet SLAs. Despite small differences in frontend estimates, the protocol generally delivers a consistent global view of the number of backends required for the current load, leaving unnecessary backends idle. The request-assignment strategy used by the frontends then restricts requests to active “in-use” backends, so that “non-in-use” backends self-decommission passively in a distributed fashion.

The model-based and SLA-aware resource estimator in Swayam exploits ML inference workload properties as derived from the random-dispatch-based distributed load balancer, SLAs, and other specific characteristics of ML inference workloads. Novel mechanisms that proactively scale-out backends based on the resource estimation model, and passively scale-in backends by self-reclamation. The scale-out mechanism seamlessly integrates with an underlying request-response mechanism.

(1) A simple gossip protocol for fault-tolerance and that scales well with load and infrastructure expansion.

(2) An analytical model for global resource estimation and prediction that takes into account local frontend loads, delays due to the random-dispatch-based distributed load balancer, SLAs, and other specific characteristics of ML inference workloads.

(3) Comparison of distributed load balancing algorithms showing that a simple approach based on random dispatch, requiring no state on the backends, is sufficient to optimize for tail latencies.

To summarize, we present the design and implementation of Swayam, a fully distributed autoscaling framework for MLaaS infrastructure with multiple frontends that meets response time SLAs with resource efficiency. Our main contributions are as follows:

- A decentralized protocol for resource allocation decisions.
- A novel resource estimation model that accurately predicts the number of backends required.
- A gossip-based fault-tolerant mechanism.
- An evaluation of Swayam using production traces.

Paper organization. We discuss related work in Sec. 2. The MLaaS platform architecture, workload characterization, SLA definition, and system objectives are discussed in Sec. 3. The design of Swayam, including request rate prediction, model-based autoscaling, distributed protocol, and comparison of different load balancing strategies is discussed in Sec. 4. Implementation details and evaluation results are presented in Sec. 5. Concluding remarks are given in Sec. 6.

2 Related Work

Jennings et al. [20] and Lorido-Botran et al. [22] survey a wide range of resource management systems, focusing on horizontal and vertical scaling, stateless and stateful services, single- and multi-tier infrastructure, etc. Since Swayam specifically aims for SLA-aware autoscaling of ML backends for stateless inference services, in
a horizontally distributed, large-scale MLaaS infrastructure, we compare and contrast it against systems with similar objectives.

Urgaonkar et al. [31] target multi-tier e-commerce applications with a tier each for HTTP servers, Java application servers, and a database. They assume centralized and perfect load balancing at each tier, and model every server in each tier using a $G/G/1$ queuing theory model. Swayam considers the challenge of multiple frontends making distributed autoscaling decisions, and also handles crash failures. Swayam’s resource-estimation model accounts for the waiting times due to load balancing. To meet SLAs, Urgaonkar et al. employ long-term predictive provisioning based on cyclic variations in the workload (over a day, month, or a year) and reactive provisioning for short-term corrections. Swayam is designed for short-term predictive provisioning to satisfy SLAs despite workload fluctuations, while minimizing resource waste.

The imperial Smart Scaling engine (iSSE) [19] uses a multi-tier architecture similar to Urgaonkar et al. [31] and explicitly models the provisioning costs in each tier for autoscaling. It uses the HAProxy [3] network load balancer and a resource monitor in each tier; the profiling data obtained by the resource monitor is stored in a central database, and is queried by a resource estimator. This centralized design scales only to a handful of heavyweight servers, e.g., for Apache, Tomcat, and MySQL, but not the tens of thousands of lightweight containers required by large-scale MLaaS (or other PaaS) providers. SCADS [30], an autoscaler for storage systems, also uses a centralized controller with sequential actions and suffers from similar drawbacks.

Zoolander [28], a key-value store, reduces tail latency to meet SLAs using replication for predictability. Redundant requests sent to distinct servers tolerate random delays due to OS, garbage collection, etc. Google Search [15] uses a similar approach, but delays sending the redundant requests to avoid useless work. Swayam solves an orthogonal efficiency problem: scaling backends to meet SLAs of multiple services in a resource-crunch environment because there may be insufficient resources to run all published services. Replication for predictability can be incorporated into Swayam with minor changes to its analytical model.

Adam et al. [9] and Wuhib et al. [32] present distributed resource management frameworks organized as a single overlay network. Similar to Swayam, their frameworks are designed to strive for scalability, robustness, responsiveness, and simplicity. But an overlay network needs to be dynamically reconfigured upon scaling and the reconfiguration must propagate to all nodes. Swayam is a better fit for large-scale datacenter environments because scaling-in and scaling-out of backends is free of configuration overheads. Swayam uses a simple protocol based on a consistent backend order that allows it to use a passive distributed scale-out mechanism. Realizing such a design in an overlay network is fairly nontrivial as frontends do not have a complete view of all backends.

Recent frameworks such as TensorFlow [8] and Clipper [14] optimize the training and inference pipeline for ML workloads. For example, TensorFlow uses dataflow graphs to map large training computations across heterogeneous machines in the Google cluster. Clipper reduces the latency of inference APIs through caching, intelligent model selection policies, and stragglers mitigation by reducing the accuracy of inference. Swayam is complementary to these frameworks, focusing on infrastructure scalability and efficiency for ML inference services.

Autoscaling is closely related to load balancing. Many distributed load balancers dispatch incoming requests randomly to backends. Random dispatch is appealing because it is simple and stateless, but it incurs non-zero waiting times even in lightly-loaded systems. The power-of-$d$ approach reduces this waiting time by querying $d > 1$ servers independently and uniformly at random from the $n$ available servers, sending a request to the backend with the smallest number of queued requests [25, 26], which reduces the expected waiting time exponentially over $d = 1$. Join-Idle-Queue (JIQ) load-balancing further reduces waiting times by eliminating the time to find an idle backend from the critical path [23]. Both the power-of-$d$ and JIQ policies optimize for mean waiting times, whereas Swayam must optimize for tail latencies. (e.g., 99th percentile response times). Swayam thus uses random dispatch for load balancing. It performs better in terms of the 99th percentile waiting times, and it is also amenable to analysis (Sec. 4.5).

3 Architecture, Workload, and Objectives

Large-scale MLaaS providers generally support end-to-end training of ML models, such as for linear regression, cluster analysis, collaborative filtering, and Bayesian inference. Developers publish the trained models as a web service that users directly query. Users may submit real-time and batch requests. Batch APIs process a set of requests asynchronously, and may have response times in hours. We focus on the real-time APIs for which the expected response time is low, e.g., 100 ms. Providers charge clients based on the number of responses that completed within a given response time threshold according to an SLA. The goal for the provider is to minimize the resources dedicated to each service while still satisfying the SLA, so that more services can be deployed in the same infrastructure, thus optimizing profitability.

For example, an application for wearable devices may give real-time feedback on exercise routines, computing whether a person is running, walking, sitting, or resting using sensor data. This problem is computationally intensive and challenging [6], and therefore is typically performed server-side, rather than on the device. The application developer decides on a suitable learning algorithm, trains it using a data set (which includes the ground truth and a set of sensor values from the accelerometer, gyroscope, GPS, etc.), and then publishes the trained model as an ML web service. End users (or applications) query the model directly with requests containing sensor data from their wearable device. The request arrival rates for this service depend on the number of users, time of day, etc., since users may be more likely to exercise in the morning and early evening. The provider thus needs to provision enough backends to be responsive to requests during daily peaks. It must also simultaneously manage many such ML services.

This section describes the MLaaS provider infrastructure, workload characteristics of ML inference requests, SLA definitions, and the system objectives of Swayam.

3.1 System Architecture

The cloud provider’s overall system architecture consists of a general pool of servers, a control plane to assign a set of backends to each service for computation, a set of frontends for fault-tolerance and scalability, and a broker (see Fig. 1). Such a horizontally scalable design with multiple frontends and backends is common in distributed systems [29].
we study ML inference workload characteristics including the per-request computation times and the provisioning times for each service, and the number of active services hosted by the provider.

To characterize MLaaS computation (or service) times, we measured the CPU time used by each backend while serving a request, since it is the dominant bottleneck. Figure 3 depicts the computation time distribution for three representative services out of the chosen 15. Variation is low because requests consist of fixed-sized feature vectors, and since popular ML models exhibit input-independent control flow. Non-deterministic machine and OS events are thus the main source of variability. The computation times closely follow log-normal distributions, which has been previously observed in similar settings [11, 18]. We use this observation for designing the analytical model for resource estimation (Sec. 4.5).

Another key characteristic of ML workloads is the non-negligible provisioning times, which can be much larger than request processing times. They rule out purely reactive techniques for scale-out, and motivate the need for predictive autoscaling, where the prediction period is a function of the provisioning times (Sec. 4.1). Provisioning a backend requires a significant amount of time since it involves creating a dedicated container with an instance of the service-specific ML model and supporting libraries obtained from a shared storage subsystem.

MLaaS providers host numerous ML services concurrently, and each service requires a dedicated service instance due to its memory and CPU requirements. Consequently, a static partitioning of resources is infeasible. Even though only a fraction of the total number of registered services are active at any time, this set fluctuates. Autoscaling is thus very essential for MLaaS providers to meet SLAs of all services with efficient resource usage.

3.3 SLAs and System Objectives
SLA is defined w.r.t. a specific ML service, forming an agreement between its publisher and the provider. We define three common SLA components as our overall objective. (1) Response time threshold: Clients are only charged if the request response time is below this upper bound. Thus, the provider can prune a request that will not meet its deadline. Alternatively, clients can mark in the SLA if they do not want requests to be pruned. (2) Service level: Clients are charged only if at least P percent of all requests are completed within the response time threshold, where P denotes the desired service level, because otherwise the overall service is not considered responsive. (3) Burst threshold: Providers must tolerate an increase in request arrival rate by up to U times, over a given short time interval, without violating objectives (1) and (2), where U denotes the burst threshold.

Burst threshold determines the service’s resilience to short-term load variations and the required degree of over-provisioning. Accounting for it depends on the average time to provision a new backend for this service. Notice that when load bursts by, say, 50%, the system still needs sufficient time to provision additional backends to maintain the same burst threshold at the new higher load. The higher the burst threshold, the more likely it is for a service to satisfy objectives (1) and (2), albeit at the cost of requiring more resources to be held back in reserve.

### System Objectives
The objective of Swayam is to minimize resource usage across all services, while meeting the aforementioned

---

**Figure 2.** State transition diagram for backends, and an example scenario for a set of eight backends assigned to a service.

The broker is an ingress router and a hardware load balancer. Frontends receive requests from the broker, and act as a software-based distributed dynamic load balancer (and in case of Swayam, also as a distributed autoscaler). Each frontend inspects the request to determine the target ML service, and then selects one of the backends assigned to the service to process the request. Since all frontends perform the same work and their processing is lightweight, we employ a standard queuing theory algorithm to autoscale the total number of frontends based on load [29].

Each backend independently processes ML inference requests. ML inference tends to have large memory footprints and is CPU-bound [4, 13, 27]. This constraint limits the number of backends that each chip multiprocessor (CMP) machine can host. A backend is encapsulated in a container with an ML model and pre-loaded libraries with dedicated CPU and memory resources on the host machine. Each backend can execute only one request at a time, all requests are read-only, and any two requests to the same or different services are independent. As none of the existing ML platforms currently perform online learning, clients must redeploy their service to update the model. We assume that backends are assigned to CMP machines by the control plane using a packing algorithm that considers the service’s memory and CPU requirements, and do not consider this problem further.

Swayam assigns a cold or a warm state to each backend associated with any service. A cold backend is not active (i.e., either not allocated, starting up, or failed) and cannot service requests. A warm backend is active and can service requests. It is either busy, if it is currently servicing a request, or is idle otherwise. A warm in-use backend is frequently busy. A warm not-in-use backend has been idle for some time, and becomes a cold backend after a configurable idle-time threshold when it releases its resources. Fig. 2 illustrates the state transition diagram and an example scenario with eight backends.

Frontends and backends communicate with a simple messaging protocol. When a frontend receives a request from the broker, it tries to forward the request to a warm in-use backend corresponding to the requested service. If the backend is cold or busy, it declines the request and returns an unavailability message to the frontend. If the backend is idle, it executes the request and returns the inference result to the same frontend.

### 3.2 Workload Characterization

We describe the characteristics of MLaaS deployments by examining 15 popular services hosted on a major production MLaaS platform. The services were chosen based on the number of requests issued to each of them during a three hour window. To design Swayam,
SLAs for each service. To achieve this goal, Swayam greedily minimizes the number of warm backends used by each service, while ensuring SLA compliance for that service. We thus measure resource usage in terms of the gross warm-backend-time, which is defined as the cumulative duration for which backends remain warm across all services. This includes time that backends spend while provisioning new service instances.

4 Swayam

Autoscaling involves two main decisions: when to scale-out or scale-in, and how many backends to use. This section describes our distributed, scalable, and fault tolerant solution to these problems. We first give an overview of Swayam (Sec. 4.1), describe request rates prediction (Sec. 4.2) and backend resource estimation (Sec. 4.3), and then focus on how Swayam integrates these using a fully distributed protocol (Sec. 4.4). Last, we explore different load balancing policies for Swayam (Sec. 4.5).

4.1 Overview

Recall that provisioning time for ML services (say $t_{setup}$) is much higher than the time to service a request. Swayam must therefore estimate demand at least $t_{setup}$ time in advance, and proactively scale-out backends if the predicted demand exceeds the current capacity of warm backends. Swayam (on each frontend) invokes the request rate predictor and demand estimator periodically and each time it detects an SLA violation, and warms up additional backends, if required. The prediction and estimation procedures require no coordination among frontends.

Swayam implements passive scale-in at backends. Recall the backend states from Sec. 3.1. If a backend has been setup for any service $S$, but it has been in warm-not-in-use state for more than a threshold amount of time, say $T$, it decommissions (or garbage collects) itself and transitions into a cold backend. Threshold $T$ is few times the provisioning prediction period at the frontends.

The advantages of our design include simplicity, distribution of scale-in decisions, and tolerance to unredicted load increases. (1) When frontends determine they need fewer backends, they simply stop sending requests to backend(s), transitioning them from warm-in-use to warm-not-in-use. Frontends do not need to communicate with each other. (2) If load increases unexpectedly or is imbalanced across frontends, frontends can instantaneously transition warm-not-in-use backends to warm-in-use backends by sending them requests.

4.2 Request Rate Prediction

Swayam predicts loads over short time periods to tolerate the rapid changes that we observe in MLaaS workloads. Let $n_F$ denote the total number of frontends. Let $\lambda$ denote the global request arrival rate for a service observed at the provider, and let $\lambda_i$ denote the local request arrival rate for that service observed at frontend $F_i$. Swayam uses linear regression to predict the request arrival rate $t_{setup}$ time units from now, where $t_{setup}$ is an upper bound on the backend service provisioning time. Because requests are uniformly distributed among $n_F$ frontends, each frontend can independently compute the global request rate as $\lambda = n_F \cdot \lambda_i$. If some frontend fails, load at the other frontends will increase. If it takes longer to detect the new correct value of $n_F$ the resulting estimates for $\lambda$ will conservatively over provision the warm number of backends.

We use a gossip protocol to communicate $n_F$ to frontends. When a backend returns a response, it includes the frontends that sent it requests in the previous $t_{setup}$ time period. When new frontends come online or fail, the backends will quickly notify frontends without any special communication rounds. Alternatively, the broker could periodically communicate $n_F$ to all frontends. We avoid using this approach since the broker is assumed to be a hardware load balancer with limited functionality.

We allow users to define a burst threshold ($U \geq 1$) to cope with abrupt load increases. If one expects the request rate to double within $t_{setup}$ time, $U = 2$ would ensure SLA-compliance during the transient period of sharp load increase, at the cost of over-provisioned resources. To account for $U$, we multiply it with the predicted request rate $\hat{\lambda}$ before using it for estimating the number of backends.

4.3 Backend Resource Estimation Model

We use a model-based approach to estimate the minimum number of backends $n_B$ required for SLA compliance. Compared to using control theory or learning, our model quickly produces a high-fidelity prediction. It takes into account both the SLA parameters and the ML workload characteristics. The model is illustrated in Fig. 4. Each frontend uses the model to independently arrive at a consistent view of backend resource demands, using the following input parameters: the expected request arrival rate $U \cdot \hat{\lambda}$, the cumulative density function $f_{CDF}$ of the request computation times, the average round trip time $RTT_{avg}$ between the frontends and the backends, and the minimum expected service level $SL_{min}$ and the response time threshold $RT_{max}$ specified in the SLA.
compute the waiting time distribution is computed and convoluted with the measured time distribution to calculate the response-time distribution (see Sec. 4.5 for details). We then compute the number of backends \( n_B \) required for SLA compliance. In particular, starting with \( n = 1 \) potential backends, in each iteration, we compute the \( 51^{\text{st}} \) percentile response time assuming that \( n \) warm backends are used, compare this value with \( RT_{\text{max}} \) to check for SLA compliance, and repeat steps 3 and 4 for increasing values of \( n \) to find the smallest value \( n_B \) that satisfies the SLA. For large ranges of potential \( n \) values, we can use binary search and/or caching values.

4.4 Distributed Autoscaling Protocol

This section presents the Swayam distributed frontend protocol, in which (1) all frontends operate independently, without communicating with any other frontends, and (2) the system globally complies with the SLA. If the global load requires \( n \) backends, then each frontend locally arrives at this same estimate and sends requests only to the same set of \( n \) warm backends, such that no backends remain warm unnecessarily. The protocol offers global SLA compliance—by performing globally consistent short-term, prediction-based provisioning—and resource efficiency—by globally minimizing \( n \).

Algorithm 1 presents pseudocode for the frontend protocol for one service. It relies on two key assumptions. (1) All frontends have the same ordered map \( B \) (Line 1) of the set of backends assigned to the service, for example, through a configuration file in a fault-tolerant service registry. (2) The map uniquely orders the backends, say, based on their unique host IDs. \( n \) denotes the number of warm backends in \( B \) as estimated by the local instance of Swayam in the frontend. Initially, \( n = 1 \).

The ExecReq() procedure is invoked when a new request arrives and if a busy or failed backend rejects a request from the frontend. The later policy limits the effects of backend failures. If the request has already exceeded its maximum response time (Line 4), the algorithm triggers autoscaling (Line 5) and optionally prunes (drops) the request (Line 6). Otherwise, the frontend dispatches the request to an idle backend \( B_{idle} \) from the prefix \( \{ B_1, B_2, \ldots, B_n \} \) of the ordered set \( B \) (Lines 9-11). After dispatching a request, Line 13 updates the request history for the purpose of arrival-rate prediction, and the autoscaling part of the protocol is invoked if needed (Line 14). The autoscaling logic consists of three steps: (a) request-rate prediction (Line 15), (b) model-based resource estimation (Line 16), and (c) autoscaling actions (Lines 18 and 21).

Since the set of available backends \( B \) is ordered, frontends generally agree on \( n \), and the load balancer chooses \( B_{idle} \) from the first \( n \) backends in \( B \) (Lines 9-11). Scaling decisions are thus consistent, despite being fully distributed. For example, if a workload increase requires \( x \) additional backends, and assuming each frontend independently arrives at the same value of \( x \), then all frontends start sending warmup requests to the same set of \( x \) backends \( \{ B_{n+1}, B_{n+2}, \ldots, B_{n+x} \} \). By updating their local copies of variable \( n \), frontends seamlessly migrate to the new consistent system configuration without any explicit communication. Similarly, if a decrease in workload reduces the number of required backends by \( y \), then, by virtue of backend ordering, all frontends stop dispatching requests to backends \( \{ B_{n-y+1}, \ldots, B_{n-1}, B_n \} \). These backends thus gradually transition into warm-not-in-use state, and eventually garbage-collect themselves, without any explicit signal from the frontends or from other backends.

One practical problem may arise from short-term, low-magnitude oscillations due to small variations in the incoming workload and the discrete nature of the resource. To avoid making frequent small adjustments in \( n \), RegulateScaleDown() (Line 21) smoothly out oscillations in \( n \) by enforcing a minimum time between two consecutive scale-in decisions. The same effect can cause frontends to produce off by one or two estimates of \( n \), but because all frontends use a consistent set of backends, the effects are small.

The order of operations in Algorithm 1 ensures that autoscaling never interferes with the critical path of requests, and the idempotent nature of model-based autoscaling ensures that strict concurrency control is not required. But we can prevent multiple threads of a frontend from sending concurrent warmup requests to the same set of backends by simple mechanisms, such as by using a try_lock() to guard the autoscaling module.
4.5 Load Balancing

Load balancing (LB) is an integral part of Swayam’s resource estimation model and its distributed autoscaling protocol. The LB policy is thus crucial to Swayam’s design. We experimentally analyze the best distributed LB algorithms in the literature, and based on the results, choose to use random dispatch in Swayam.

An ideal LB policy for Swayam (1) must effectively curb tail waiting times (rather than mean waiting times) since the target SLAs are defined in terms of high response-time percentiles. (2) may not use global information that requires communication or highly accurate workload information to ensure unhindered scalability w.r.t. the number of frontends and backends, and (3) must be amenable to analytical modeling for the resource estimation to work correctly. Based on these criteria, we evaluated three distributed LB policies:

- **Partitioned scheduling (PART):** Backends are partitioned evenly among all frontends. Pending requests are queued at the frontends. Each partition resembles an ideal global scheduling system with a single queue.
- **Join-idle-queue (JIQ) [23]:** Requests are queued at the backends. To help frontends find idle or lightly loaded backends, a queue of idle backends is maintained at every frontend that approximates the set of idle backends. A frontend dispatches a request to a backend in the idle queue, or to a random backend if the queue is empty.
- **Random dispatch (RAND):** Frontends forward any incoming request to a random backend. If the backend is busy, it rejects the request and sends it back to the frontend. The frontend then retries another random backend. We also analyze a global scheduler with a single request queue, where all backends serve the single shared queue. Global scheduling represents a lower bound on actually achievable delays.

We simulated these policies in steady state assuming that requests arrive following a Poisson distribution and that request computation times follow a lognormal distribution derived from production workload traces. Fig. 5(a) illustrates the observed 99th percentile waiting times as a function of the number of backends \( n \). Additionally, a threshold of 350ms (the dashed line) illustrates a typical upper bound on acceptable waiting times.

We observed three trends. First, PART performs closest to the global baseline. Second, 99th percentile waiting times under both JIQ and RAND initially drop quickly as \( n \) increases, but there exists a point of diminishing returns close to \( n = 40 \), which results in long tails (albeit well below the threshold). Third, 99th percentile waiting times under JIQ are almost twice as high as those under RAND.

The last observation was unexpected. JIQ is designed to ensure that requests do not bounce back and forth between frontends and backends, and that the time to find an idle backend does not contribute to the critical path of a request. Thus, mean waiting times under JIQ are close to mean waiting times under the global scheduling baseline [23]. However, we found that this property does not extend to JIQ’s tail waiting times. It is further nontrivial to analytically compute percentile waiting times in JIQ.

PART exhibited desirable tail waiting times, but it is not a good fit for Swayam, as it requires heavyweight, explicit measures to deal with frontend crashes, such as partition reconfiguration, i.e., it is not implicitly fault tolerant or tolerant of small variations in local predictions of \( n \).

We thus choose RAND (1) due to its simplicity and obliviousness to host failures, (2) because the resulting tail waiting times are comfortably below typically acceptable thresholds (they are good enough for ML inference workloads), and (3) the percentile waiting times can be analytically derived (see below). We modify RAND to maintain a configurable minimum delay between two consecutive retries of the same request to prevent message storms during times of transient overload. RAND can also be enhanced with the power-of-d approach to reduce variances in the percentile waiting times without affecting other components of Swayam [26, 30].

Under the modified RAND LB policy, the \( p \)th percentile waiting time of an ML inference request when there are \( n \) active backends is approximated as:

\[
\omega_p = d_1 + \left( \frac{\ln \left( 1 - \frac{p}{100} \right)}{\ln \left( \frac{1}{n^p} \right)} - 1 \right) \cdot (d_2 + d_3 + \Delta),
\]

where \( d_1 \) is the mean time required to send a request from the frontend to the backend, \( d_2 \) is the mean time required to send a request from the backend to the frontend, \( \Delta \) is the enforced delay between consecutive retries, \( \lambda \) is the mean arrival rate, and \( \mu \) is the mean service rate. Parameters \( d_1, d_2, \) and \( \mu \) are derived from system measurements, and \( \lambda \) is determined as explained in Sec. 4.2.

The \( p \)th percentile response time for the ML inference request is approximated by convoluting the derived waiting time distribution with the measured computation time distribution of the respective ML service. Derivation of Eq. 1 and the convolution procedure are provided in the online appendix [17]. The analysis results are very close to the measurements obtained from the simulator, demonstrating the high accuracy of our model (see Fig. 5(b)).

5 Evaluation

As we are unable to report results from the production system, we prototyped Swayam in C++ on top of Apache Thrift Library, using its RPC protocols and C++ bindings. Each component, i.e., frontends, backends, broker, is a multi-threaded Apache Thrift server, and communicates with other components through pre-registered RPC calls.

We used a cluster of machines networked over a 2x10GiB Broadcom 57810-k Dual Port 10Gbit/s KR Blade. Each machine consists of two Intel Xeon E5-2667 v2 and 8 core processors. We used a pool of one hundred backend containers, eight frontend servers, one server for simulating clients, and one broker server.

The client machine replays production traces by issuing requests to the broker based on recorded arrival times. The broker, which simulates the ingress router, forwards the requests to the frontends in a round-robin fashion. The frontends then dispatch requests as per Swayam’s autoscaling policy or a comparison policy. Since we do not have access to the actual production queries, but only their computation times, to emulate query computation, the backend spins for the duration equivalent to the request’s computation time. Similarly, to emulate setup, it spins for the setup time duration.

**Workload.** We obtained production traces for 15 services hosted on a major production MLaas platform that received the most requests during a three-hour window (see Sec. 3.2). Each trace contained the request arrival times and the computation times. For a detailed evaluation of Swayam, we chose three traces with distinct request arrival patterns and that are most challenging services for autoscaling due to their noisiness and burstiness. Their mean computation
times are \( t_1 = 135\text{ms} \), \( t_2 = 117\text{ms} \), and \( t_3 = 167\text{ms} \). The provisioning time was not known from the traces. We assumed it to be 10s for each service based on discussions with the production team. For resource efficiency experiments, all 15 traces were used.

**Configuration parameters.** Unless otherwise mentioned, we configured Swayam to satisfy a default SLA with a desired service level of 99%, a response time threshold of 5\( c \), where \( c \) denotes the service’s mean computation time, a burst threshold of 2\( c \), and no request pruning. We chose 2\( c \) as the default burst threshold to tolerate the noise in the arrival rates we observed in production traces, e.g., see the oscillatory nature of trace 1 in Fig. 6(a), and to tolerate frequent bursts in the arrival pattern, e.g., see the spikes at time 6000s and 7500s in Fig. 6(b).

Request pruning was disabled by default even though it actually helps improve the overall SLA compliance during spikes or bursts in the request arrivals (as shown below in Sec. 5.2). This is because, based on conversations with the production team, most client applications seem to time out on requests that take a long time to complete, but requests that complete within a reasonable amount of time, despite violating the stated SLA, are useful.

For each service, we provisioned five backends before replaying the trace for bootstrapping, since the employed traces represent already active services. Traces 1 and 2 were configured with the default SLA. Trace 3 was configured with a higher burst factor of 8\( c \) to tolerate very high bursts in its request rates. The default SLA configuration is just a suggestion. In practice, the SLA is determined by the client who has better information about the workload pattern and desired performance.

**Interpreting the figures.** We illustrate results using three types of figures: request rate figures, resource usage figures, and SLA compliance figures. Each of these is explained in detail below.

Figs. 6(a)–6(c) illustrate the actual request rates and the predicted request rates at one frontend to evaluate the prediction accuracy. We computed actual request rates offline using trailing averages over a 100s window in steps of 10s. Swayam predicted request rates online over a window of size 100s, using linear regression and request history from the past 500s. The system forecasted request rates 10s and 100s into the future, i.e., the request rates predicted for time \( t \) are computed at time \( t−10s \) and \( t−100s \).

Figs. 7(a)–7(c) illustrate the total numbers of backends that are warm and the total number of backends that are in-use by a frontend. We keep track of state changes on the backend to report warm backends. We report warm-in-use backends for any frontend \( F_i \) based on its local state that its load balancer uses for dispatching. For both request request rate and resource usage figures, results for remaining frontends were similar (and not shown to avoid clutter).

Response times and SLA compliance over time are illustrated in Figs. 8(a)–8(c). Trushkowsky et al. [30] measured SLA compliance by computing the average 99th percentile latency over a 15 minutes smoothing interval. In contrast, we measure SLA compliance over a fixed number of requests, i.e., we use a set of 1000 requests as the smoothing window. We slide the window in steps of 10 requests, and report the SLA achieved for each step. Our metric is actually harsher than prior work since during load bursts, prior metrics might report only a single failure.

### 5.1 Mechanisms for Prediction and Scaling

Although both actual and predicted request rates are measured over an interval of 100s, Figs. 6(a)–6(c) show that predicted request rate curves are smoother in comparison. This is because the predictor relies on data from the past 500s, and forecasts based on the trend observed in that data (which acts as an averaging factor). Second, we observe that despite the noise in the production traces 1 and 2, the predictor is able to track the actual request rates closely. The predictor is also able to track intermittent spikes in these request rates. To understand the predictor’s behavior in detail, a portion of Fig. 6(a) corresponding to time interval [1000s, 2500s] is magnified and illustrated in Fig. 9(a) below. It can be observed that the peaks and troughs in the predicted request rate corresponding to those in the actual request rate are slightly shifted to the right. But recall from Sec. 4.4 that the resource estimator in Swayam is designed to mask the effect of such oscillations on the estimated resource demand, i.e., the number of backends used by the frontends does not oscillate very frequently, and that this number is reduced only if the drop in the request rate is observed over a slightly longer period of time (e.g., ten minutes). Therefore, these slight shifts in the predicted rate do not affect the end results.
Figure 6. The actual and the predicted request rates at FE0 for production traces 1, 2, and 3, respectively. Results for other frontends are similar and not illustrated to avoid clutter.

Figs. 7(a)–7(c) illustrate the number of backends active (warm) and the number of backends used by one of the frontends (i.e., FE0) for production traces 1, 2, and 3, respectively. Results for other frontends are similar and not illustrated to avoid clutter.

$t_1 \approx 4500s$ in Fig. 7(a), due to a spike in the request rates, the frontend decides to scale-out the number of backends to fifteen. As a result, the number of warm backends increases from thirteen to fifteen, i.e., two additional backends are warmed up. But the spike lasts only for few minutes, and so the frontends scale-in locally, i.e., they start using a reduced number of backends. Once all frontends scale-in locally to thirteen or fewer backends, the two additional backends idle, and are eventually collected. If at least one frontend had continued using more than thirteen backends for a longer period of time, or if at least one frontend used a different
set of thirteen backends, then the two additional backends would not have been garbage collected.

5.2 SLA Compliance with Autoscaling
As illustrated in Fig. 8(a), Swayam complies with the SLA for almost the entire duration of trace 1. There are some SLA violations in the beginning because the five backends initially setup for bootstrapping fall short of the expected resource demand of twelve backends, but Swayam is able to quickly scale-out to account for this difference. The SLA is also briefly violated at times \( t_1 \approx 4500s \) and \( t_2 \approx 7500s \) when there are sudden increases in the request rate. Even though these increases were within the over-provisioning factor of 2x, the instantaneous rate at the time of increase was much higher, resulting in SLA violations. Results for trace 2 are similar, i.e., a few SLA violations occur during the steep spike at time \( t_3 \approx 6300s \), and SLA compliance otherwise (see Fig. 8(b)). Results for trace 3, however, are different owing to its unique arrival pattern (see Fig. 8(c)). From Fig. 6(c), it can be seen that requests in trace 3 arrive almost periodically. In fact, upon a closer look at the logs, we found that the request trace consists of periods of absolutely no request arrivals alternating with periods where many requests arrive instantaneously, resulting in more SLA violations than for trace 1 and 2. Note that it is impossible to schedule for an instantaneous arrival of so many requests unless the necessary resources are already provisioned and unless there is perfect load balancing. Overall, while SLAs are violated less than 3% of the time for traces 1 and 2, they are violated about 20% of the time for trace 3, which pushes Swayam to its limits due to its extremely bursty nature.

To understand the benefits of request pruning on SLA compliance, we replayed trace 2 with the pruning option turned on. Fig. 9(b) shows that request pruning improves SLA compliance significantly during spikes without compromising on the SLA compliance during the steady states (w.r.t. Fig. 8(b)). This is because (1) we do not have strict admission control at the entry point of Swayam, but only pruning of requests after their waiting times in the system have exceeded the response time threshold; and because (2) in a randomized load balancer like Swayam’s, requests accumulated during spikes affect the requests that arrive later, but this is prevented in this case by pruning tardy requests. To the client, a pruned request is similar to a timed-out request (i.e., the client is not charged for the request) and can be handled by reissuing the request or by ignoring the request, based on the application semantics.

5.3 Fault Tolerance
Swayam tolerates crash failures by incorporating a gossip protocol on top of the underlying request-response protocol, which ensures that all frontends have a consistent view of the total number of active frontends for each service. If a backend crashes, each frontend independently discovers this using the RAND LB policy, when an RPC to that backend fails.

Fig. 10(a) shows the effect of two frontend crashes on the predicted request rates when trace 2 is replayed for 3000s. We used only three frontends to maximize the impact of failures. A frontend failure causes the broker to redirect load to the other live frontends. After the first failure, the live frontends over-estimate the global request rate by 1.5x (and after the second failure, by 2x) until they find out about the failures through backend gossip (recall Sec. 4.2). Since the period of over-estimation is small, the impact on resource usage is minimal. In addition, since the frontends do not explicitly queue requests locally, frontend failures directly impact only the requests waiting for a retry. We did not see any impact of frontend failures on the SLA compliance of trace 2.

We repeated the experiment with backend failures instead of frontend failures. Fig. 10(b) shows that the frontends quickly recover from backend failures by reconfiguring their respective load balancers to use additional warm backends. The number of warm in-use backends (as suggested by the analytical model) remains at three, the total number of warm backends (including the failed backends) increases to four after the first crash, and then to five after the second crash. Since backends do not explicitly queue pending requests, a failure affects the request being processed and may
increase the waiting time of other requests by a single RTT each. But we did not see any noticeable effect on SLA compliance.

5.4 Autoscaling and Resource Usage

We compare the resource efficiency of Swayam with that of a clairvoyant autoscaler, ClairA. The clairvoyant autoscaler ClairA has two special powers: (1) it knows the processing time of each request beforehand, which it can use to make intelligent scheduling decisions; and (2) it can travel back in time to provision a backend, eliminating the need for proactive autoscaling (and for rate-prediction and resource estimation). Using these powers, ClairA follows a “deadline-driven” approach to minimize resource waste (similar to [21]). For example, if a request arrives at time $t$, since ClairA knows the processing time of the request in advance (from 1), say $c$, and since the response time threshold of the corresponding service is also known, say $RT_{max}$, it postpones the execution of the request as late as possible, i.e., until time $t + RT_{max} - c$. It guarantees that there is an idle backend ready to execute the request at time $t + RT_{max} - c$ by provisioning it in the past, if required (using 2).

Moreover, backends can be scaled-in either instantly after processing a request or lazily using periodic garbage collection (like in Swayam). We evaluate two versions: ClairA1, which assumes zero setup times and immediate scale-ins, i.e., it reflects the size of the workload, and ClairA2, which is Swayam-like and assumes non-zero setup times with lazy collection. Both ClairA1 and ClairA2 always satisfy the SLAs, by design.

In Fig. 11, we illustrate the resource usage for all 15 production traces, when replaced with autoscalers ClairA1, ClairA2, and Swayam. The traces were each executed for about 3000s. We measure resource usage in terms of warm-backend-time, i.e., the cumulative duration for which backends consumed resources, including warmup time. We also denote in Fig. 11 the frequency with which Swayam complies with the SLA for each service. For example, out of all the windows consisting of 1000 consecutive requests (recall our SLA compliance metric), if Swayam complied with the SLA in only 97% of these windows, then its SLA compliance frequency is 97%.

Both ClairA1 and ClairA2 significantly differ in terms of their resource usage, which shows that setup costs are substantial and should not be ignored. Comparing ClairA2 to Swayam, for certain services (traces 1, 2, and 12-15), Swayam is more resource efficient but at the cost of SLA compliance. This shows that guaranteeing a perfect SLA comes with a substantial resource cost.

For trace 3, despite provisioning significantly more resources than ClairA2, SLA compliance is very poor. This is because (1) we measure SLA compliance w.r.t. a finite number of requests and not w.r.t. a finite interval of time; hence, during instantaneous bursts,
we record multiple SLA compliance failures, as opposed to just one (i.e., our evaluation metric is biased against Swayam); and (2) ClairA2’s deadline-driven approach, which takes into account the request computation times and their response-time thresholds before dispatching, is unattainable in practice. Trace 4 is similar to trace 3, but relatively less bursty.

For all other traces, Swayam always guarantees SLAs while performing much better than ClairA2 in terms of resource usage. This is again because of ClairA2’s deadline-driven approach, due to which it occasionally ends up using more backends than Swayam, but then these extra backends stay active until they are collected. Overall, Swayam uses about 27% less resources than ClairA2 while complying with the SLA over 96% of the time.

Providing perfect SLA irrespective of the input workload is too expensive in terms of resource usage, as modeled by ClairA. Practical systems thus need to trade off some SLA compliance, while managing client expectations, to ensure resource efficiency. Our results show that Swayam strikes a good balance in this regard by realizing significant resource savings at the cost of only occasional SLA violations.

6 Conclusion

This paper introduces a new distributed approach for autoscaling services that exploits ML inference workload characteristics. It derives a global state estimate from local state and employs a globally consistent protocol to proactively scale-out service instances for SLA compliance, and passively scale-in unnecessary backends for resource efficiency.

Since guaranteeing all SLAs at all times is economically not viable in practice, a practical solution must find a good tradeoff between SLA compliance and resource efficiency. Our evaluation shows that Swayam achieves the desired balance — it still meets most SLAs with substantially improved resource utilization.

In future work, it will be interesting to extend Swayam for heterogeneous frontends by gossiping request rates observed at the backends to the frontends for prediction, and to incorporate long-term predictive provisioning by determining burst threshold as a function of diurnal workload patterns. Note that supporting online updates of the ML model is orthogonal to the autoscaling problem, since it may require recompilation of the service containers. We are planning to make Swayam publicly available and are currently in the process of preparing the codebase for an open-source release.

References


[18] Varun Gupta, Mor Harchol-Balter, JG Dai, and Bert Zwart. 2010. On the approximability of M/G/K: why two moments of job size distribution are not...
Swayam: Distributed Autoscaling to Meet SLAs of ML Inference Services


