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

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

Sameh Elnikety
MSR, USA
samehe@microsoft.com

Yuxiong He
MSR, USA
yuxhe@microsoft.com

Kathryn S. McKinley
Google, Inc., USA
ksmckinley@google.com

Björn B. Brandenburg
MPI-SWS, Germany
bbb@mpi-sws.org

## ABSTRACT

Developers use Machine Learning (ML) platforms to train ML models and then deploy these ML models as web services for inference (prediction). A key challenge for platform providers is to guarantee response-time Service Level Agreements (SLAs) for inference workloads while maximizing resource efficiency. *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 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 decreases resource utilization by up to 27%, while meeting the service-specific SLAs over 96% of the time during a three hour window. Microsoft Azure's Swayam-based framework was deployed in 2016 and has hosted over 100,000 services.

## CCS CONCEPTS

• **Computer systems organization → Cloud computing**; *Availability*; • **Software and its engineering → Cloud computing**; *Message oriented middleware*; • **General and reference →** *Evaluation*; *Experimentation*;

## KEYWORDS

Distributed autoscaling, machine learning, SLAs

**ACM Reference Format:**
Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *Proceedings of Middleware '17*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3135974.3135993


*Middleware '17, December 11–15, 2017, Las Vegas, NV, USA*

ACM ISBN 978-1-4503-4720-4/17/12.
https://doi.org/10.1145/3135974.3135993

## 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]. As a typical example, consider a mobile fitness application that collects sensor data and sends a request to an inference service for predicting whether a person is running, walking, sitting, or resting. Such inference requests 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 minimized.

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 changing demand. Provisioning resources, however, represents a major challenge for MLaaS providers because the processing time of an inference 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.

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 policy for predictive autoscaling 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 Microsoft Azure Machine Learning [5], a 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 (see Fig. 1 for an illustration).

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 production logs, such as request arrival patterns, execution time distributions, and non-trivial provisioning times for each service. It combines these factors analytically with the service-specific SLA, accounts for delays due to the underlying distributed load balancing mechanism, and accurately estimates the minimum number of service-specific backends required for SLA compliance.

While this protocol is deployed in the Microsoft Azure MLaaS platform, to comply with public disclosure rules, this paper reports on an independent re-implementation deployed in a server farm consisting of eight frontend and up to a hundred backend servers. For comparison, we simulate a clairvoyant system that knows the processing time of each request beforehand and uses it to make intelligent SLA-aware scheduling decisions. The clairvoyant system can travel back in time to setup an ML instance in the past, so as to "magically" provision new ML instances without latency. It thus represents a near-optimal, but unattainable baseline. We evaluate Swayam using production traces of 15 of the most popular services hosted on the production MLaaS platform, while imposing the SLA

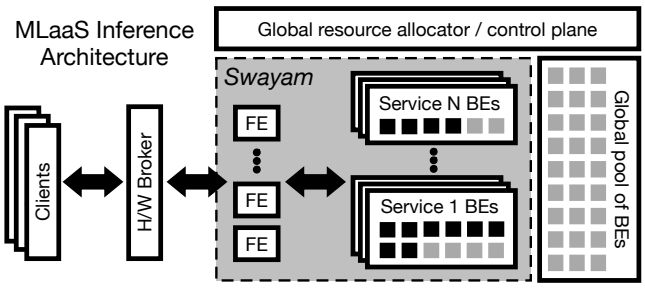



**Figure 1: System architecture of an ML as a service platform for handling inference requests corresponding to $N$ distinct services. The global resource allocator periodically assigns idle backends (BEs) from the global pool to each of the services. The frontends (FEs) setup service-specific ML models on these idle backends before using them for inference. Swayam minimizes the number of provisioned backends "in use" (black squares) while remaining SLA compliant, and ensures that redundant "non-in-use" backends (grey squares) self-decommission themselves to the global pool.**

that, for each service, at least 99% of requests must complete within a service-specific response-time threshold.

The results show that Swayam meets SLAs over 96% of the time during a three hour window, while consuming about 27% less resources than the clairvoyant autoscaler (which by definition satisfies all SLAs 100% of the time). In other words, Swayam substantially improves resource efficiency by trading slight, occasional SLA-compliance violations in a way that does not violate client expectations, i.e., only when the request rate is extremely bursty.

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:

(1) 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.
(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) 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.
(4) A simple gossip protocol for fault-tolerance and that scales well with load and infrastructure expansion.
(5) Workload characterization of 15 popular services hosted on the Microsoft Azure MLaaS platform, and evaluation of Swayam against a clairvoyant baseline for this production workload.

**Paper organization.** We discuss related work in §2. The MLaaS platform architecture, workload characterization, SLA definition, and system objectives are discussed in §3. The design of Swayam,

including request rate prediction, model-based autoscaling, distributed protocol, and comparison of different load balancing strategies is discussed in §4. Implementation details and evaluation results are presented in §5. Concluding remarks are given in §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 explicitly accounts for the non-zero waiting times due to non-ideal load balancing in its resource estimation model, considers the challenge of multiple frontends making distributed autoscaling decisions, and also handles crash failures. To meet SLAs, Urgaonkar et al. employ long-term predictive provisioning based on cyclic variations in the workload (over days or weeks) and reactive provisioning for short-term corrections. In contrast, Swayam is designed for short-term predictive provisioning to satisfy SLAs 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 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 limit useless work. Swayam solves an orthogonal efficiency problem: scaling backends to meet SLAs of multiple services in a resource-constrained 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 to implement 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 heterogenous machines in the Google cluster. Clipper reduces the latency of inference APIs through caching, use of intelligent model selection policies, and by reducing the accuracy of inference for straggler mitigation. Swayam is complementary to these frameworks, focusing on infrastructure scalability and efficiency for ML inference services. In addition, Swayam relies on a black box ML model—it does not leverage ML model internals, but uses externally-profiled ML model characteristics.

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., $99^{\text{th}}$ percentile response times). Swayam thus uses random dispatch for load balancing. It performs better in terms of the $99^{\text{th}}$ percentile waiting times, and it is also amenable to analysis (§4.5).

## 3 ARCHITECTURE, WORKLOAD, AND SLAS

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 for real-time requests based on the number of responses that complete under a given response time threshold stipulated by an SLA. Thus, to optimize profitability, 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 on the same infrastructure. However, achieving this goal is challenging due to varying service demands.

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

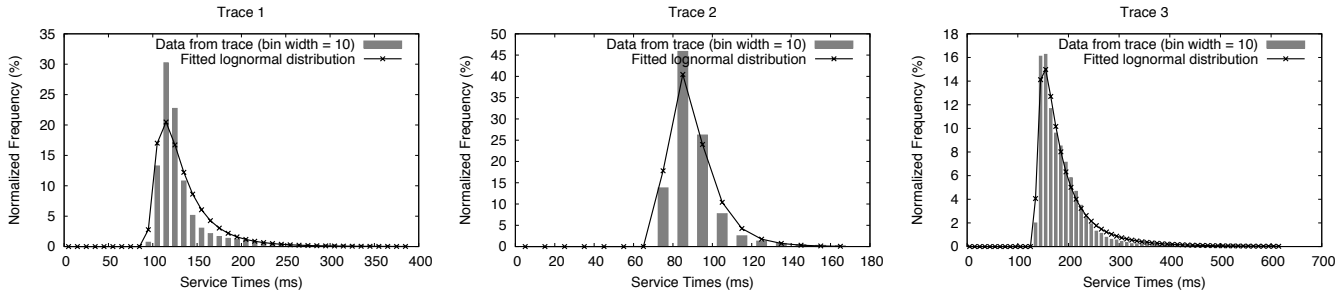



**Figure 2: Processing time histogram for three services over a three-hour period (truncated at the 99$^{th}$ percentile for clarity).**

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 respond 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 for serving the ML inference requests 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]. A similar architecture exists for batch requests and for the training APIs, which uses backends from the same general pool of servers, but which is driven by different objectives, such as ensuring a minimum training accuracy, minimizing the makespan of batch jobs, etc. In this paper, we focus on autoscaling the ML inference serving architecture.

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 requests tend to have *large memory footprints* and are 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 (or incremental) 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.

## 3.2 Workload Characterization

We describe the characteristics of MLaaS deployments by examining 15 popular services hosted on Microsoft Azure's 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, we study the ML model characteristics that can be profiled externally by the provider, i.e., we consider the ML models as a black box. In particular, we leverage the per-request computation times, the provisioning times, 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 2 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 (see §4.3 and 4.5).

Another key characteristic of ML workloads is the non-negligible *provisioning times* (also known as deployment 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 (§4.2). 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. Consecuently, 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.

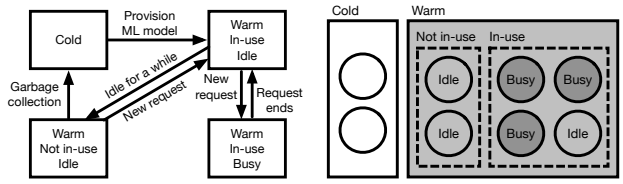



**Figure 3: State transition diagram for backends, and an example scenario for eight backends assigned to a service.**

Autoscaling is thus essential for MLaaS providers to meet SLAs of all services with efficient resource usage.

### 3.3 Service Level Agreements (SLAs)

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, denoted $RT_{max}$. 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 $SL_{min}$ percent of all requests are completed within the response-time threshold, where $SL_{min}$ 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 a factor of up to $U$, over a given short time interval, without violating objectives (1) and (2), where $U$ denotes the burst threshold.

The burst threshold determines the service's resilience to short-term load variations and the required degree of over-provisioning. It depends on the average time to provision a new backend for this service. Notice that when the load increases 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.

## 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 (§4.1), describe request rates prediction (§4.2) and backend resource estimation (§4.3), and then explain Swayam's fully distributed protocol to integrate them (§4.4). Last, we explore different load balancing policies for Swayam (§4.5).

### 4.1 Overview

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. See Fig. 3 for an illustration of the different state transitions. Swayam relies on these states for *passive scale-in* and *proactive scale-out* of backends.

Frontends and backends communicate with a simple messaging protocol. When a frontend receives a request from the broker, it forwards 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.

Each frontend periodically invokes the request rate predictor and demand estimator. Since the provisioning time for ML services (say $t_{setup}$) is much higher than the time to service a request, the demand estimator locally computes the demand at least $t_{setup}$ time in advance. If the predicted demand exceeds the current capacity of warm backends, the frontend increases the number of in-use backends by sending requests to the cold backends, which then start warming up in advance. Swayam thus proactively scales out backends without any coordination between the prediction and estimation procedures on each frontend.

In contrast to the proactive scale-out procedure, backends are scaled in passively. If a backend has been set up for a service $S$, but it has remained unused for longer than a given threshold, say $T$, it decommissions (or garbage-collects) itself and transitions into the cold state. The threshold $T$ is a small multiple of the request rate prediction period at the frontends.

The advantages of our design include simplicity, distribution of scale-in decisions, and tolerance to unpredicted 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; they do not need to communicate with each other. **(2)** If load increases unexpectedly or is imbalanced across frontends, frontends instantaneously transition warm-not-in-use backends to warm-in-use backends by sending them requests.

With this design, the objective of Swayam is to minimize resource usage across all services while meeting the service-specific SLAs. To achieve this goal, Swayam greedily minimizes the number of warm backends used by each service, while ensuring SLA compliance for that service. Therefore, in the remainder of this section, we describe Swayam's design in detail w.r.t. a single service.

### 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.

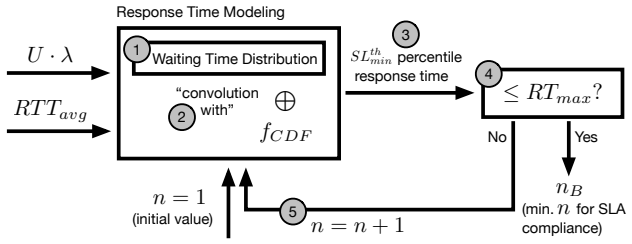



**Figure 4: Procedure to compute the minimum number of backends $n_B$ required for SLA compliance, based on $U$ and $\lambda$.**

We use a *gossip* protocol to communicate $n_F$ to frontends. When a backend returns a response to an inference request, it includes the number of frontends from which it had received requests during the last $t_{setup}$ time units. When new frontends come online or fail, other frontends are thus quickly notified by the backends without any special communication rounds. Alternatively, the broker could periodically communicate $n_F$ to all frontends, but we avoid using this approach since the broker is assumed to be a hardware load balancer with limited functionality.

Users define a burst threshold, denoted $U$, to cope with abrupt load increases ($U \geq 1$). 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 $U$, the predicted rate $\lambda$ is multiplied with threshold $U$ before using it for resource estimation.

## 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, a model-based approach quickly produces a high-fidelity prediction. Since prior model-based autoscalers using queuing-theoretic models assume ideal load balancing, we propose a new model that takes into account the non-zero waiting times due to load balancing, 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 $U \cdot \lambda$, the cumulative density function $f_{CDF}$ of the request computation times, the average round trip time $RTT_{avg}$ between the frontends and the backends, the minimum expected service level $SL_{min}$, and the response time threshold $RT_{max}$ specified in the SLA.

As per the model, the expected request response time assuming $n$ warm backends is estimated as follows. Based on the underlying load balancing policy, (1) the waiting time distribution is computed and (2) convoluted with the measured computation time distribution to calculate the response-time distribution (see §4.5 for details). We then compute the minimum number of backends $n_B$ required for SLA compliance. Starting with $n = 1$ potential backends in each iteration, (3) we compute the $SL^{th}_{min}$ percentile response time assuming $n$ warm backends, (4) compare this value with $RT_{max}$ to check for SLA compliance, and (5) repeat steps 3 and 4 for increasing values of $n$ to find the smallest value $n_B$ that satisfies the SLA. If $n$ can be potentially very large, we can use binary search and/or cache 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 SLA compliance—by performing globally consistent, short-term, prediction-based provisioning—and resource efficiency—by globally minimizing $n$.

Algorithm 1 presents a pseudocode of the frontend protocol for a single 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. Parameter $n$ denotes the number of warm backends in $B$ as estimated by the local instance of Swayam in the frontend, and is initialized to one (Line 2).

The ExecReq($r$) procedure is invoked when a new request $r$ arrives or if a busy or a failed backend rejects request $r$ that was earlier sent to it by 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 7-10).

After dispatching a request, the frontend updates the request history for the purpose of arrival-rate prediction (Line 11). It then invokes the autoscaling part of the protocol if needed, i.e., if the request exceeds its maximum response time (and violates the SLA), or if the periodic timer for autoscaling (not shown in Algorithm 1) expires (Line 12). The autoscaling logic consists of three steps: **(a)** request-rate prediction (Line 13), **(b)** model-based resource estimation (Line 14), and **(c)** autoscaling actions (Lines 16 and 19).

Since the set of available backends $B$ is ordered and since frontends generally agree on $n$, the load balancer almost always chooses $B_{idle}$ from the first $n$ backends in $B$. Scaling decisions are thus consistent, despite being fully distributed. For example, if an increase in the workload 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.

A practical problem may arise from short-term, low-magnitude oscillations due to small variations in the incoming workload and due to the discrete nature of the resource. That is, if $n$ frequently oscillates between, say, $x$ and $x + 1$, the extra backend $B_{x+1}$ is neither garbage collected nor utilized continuously for request execution. To avoid making such frequent small adjustments in $n$,

**Algorithm 1** Frontend protocol for receiving the requests.

```
1:  B ← {B_1, B_2, ... B_max}                          ▷ available backends
2:  n ← 1                                              ▷ single backend used initially
3:  procedure ExecReq(r)
4:      if RequestExceedsMaxRT(r) then
5:          AutoScalingRequired ← true
6:          TimeOut(r)                                 ▷ if request pruning is enabled
7:      if RequestHasNotTimedOut(r) then
8:          B_idle ← LoadBalancer(B, N)
9:          ▷ guarantees that B_idle ∈ {B_1, ..., B_n}
10:         DispatchReq(r, B_idle)
11:     UpdateReqHistory(r)                            ▷ used for prediction
12:     if AutoScalingRequired() then
13:         rate ← PredictRate()
14:         n_new ← AnalyticalModel(r, rate)
15:         if n_new > n then
16:             SendWarmupReqsTo(N_{n+1}, ... B_{n_new})
17:             n ← n_new
18:         else if n_new < n then
19:             n ← RegulateScaleDown(n_new)
```

RegulateScaleDown($n_{new}$) (Line 21) smoothes out oscillations in $n$ by enforcing a minimum time between two consecutive scale-in decisions. Although this affects the resource efficiency slightly (for genuine cases of scale-in), it prevents undue SLA violations.

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. Nonetheless, 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. We experimentally analyzed the best distributed LB algorithms in the literature, and based on the results, chose 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 model-based resource estimation to work correctly. Based on these criteria, we evaluated the following three distributed LB policies: *Partitioned scheduling (PART)*, *Join-idle-queue (JIQ)* [23], and *Random dispatch (RAND)*.

- 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.
- JIQ: 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 *approximate*s 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.
- 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, which then retries another random backend.
- Baseline: As a baseline, we used a *global scheduler*, where a single request queue is served by all the backends. Global scheduling thus 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 the request computation times follow a lognormal distribution. The distribution was derived from one of the production workload traces with a mean computation time of 117 $ms$. Fig. 5(a) illustrates the observed 99$^{\text{th}}$ 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, 99$^{\text{th}}$ 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, 99$^{\text{th}}$ 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.

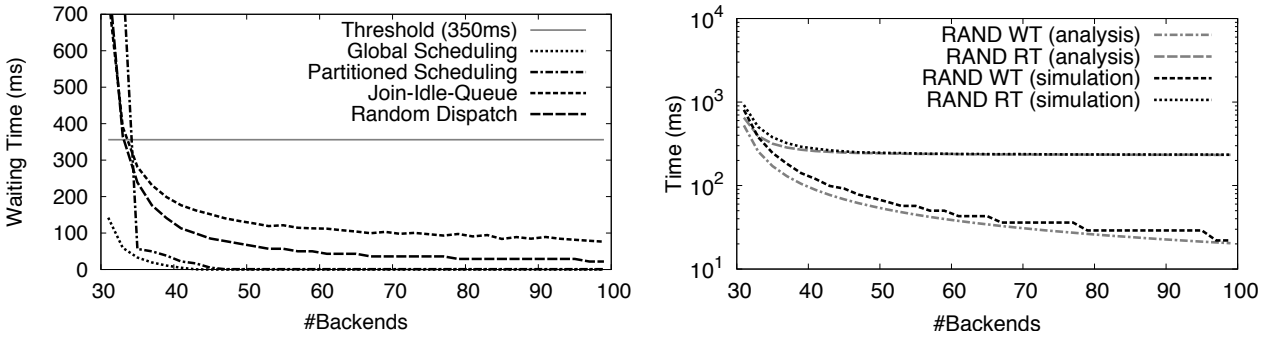
PART exhibited desirable tail waiting times, but it is not a good fit for Swayam, as it requires heavyweight, explicit measures to deal with frontend crashes, such as partition reconfiguration. That is, PART is not implicitly fault tolerant or tolerant of small variations in local predictions of $n$.

We thus chose 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 policy, the $p^{\text{th}}$ percentile waiting time of a request when there are $n$ active backends is approximated as:

$$\omega_p = d_1 + \left( \frac{\ln(1 - p/100)}{\ln(\lambda/n\mu)} - 1 \right) \cdot (d_1 + d_2 + \Delta), \tag{1}$$

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 §4.2. The $p^{\text{th}}$ percentile response time for the ML inference request is




(a) LB policies comparison

(b) Simulation vs. analysis for RAND LB policy

**Figure 5: (a) Comparing 99$^{th}$ percentile waiting times of different LB policies. The threshold of 350ms is a ballpark figure derived from production traces to provide a rough estimate of the maximum waiting time permissible for SLA compliance. (b) 99$^{th}$ percentile waiting (WT) and response times (RT) predicted by the analytical model versus simulation results. The model predicts the trends with sufficient accuracy.**

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 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 the Apache Thrift library, using its RPC protocols and C++ bindings. Each component, i.e., frontends, backends, and the 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 processors with 8 cores each. 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 costs, it spins for the setup time duration.

**Workload.** We obtained production traces for 15 services hosted on Microsoft Azure's MLaaS platform that received the most requests during a three-hour window (see §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 that are the most challenging services for autoscaling due to their noisiness and burstiness. Their mean computation times are $\bar{c}_1 = 135ms$, $\bar{c}_2 = 117ms$, and $\bar{c}_3 = 167ms$. The provisioning time was not known from the traces. We assumed it to be 10$s$ for each service based on discussions with the production team. For resource efficiency experiments, all 15 traces were used.

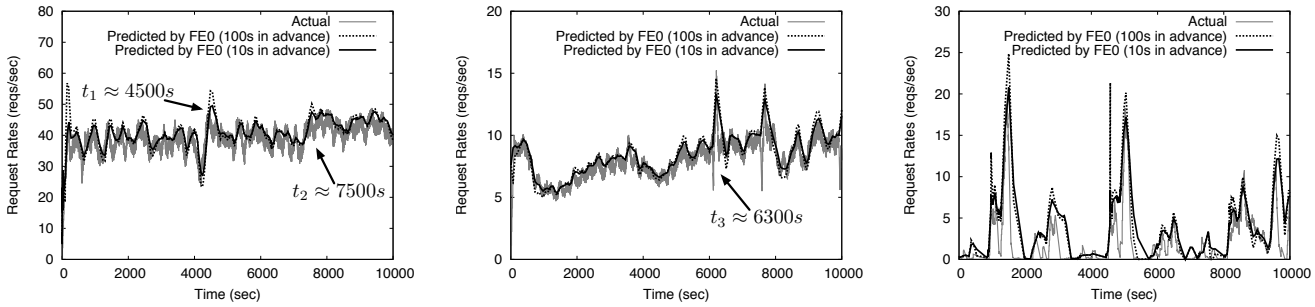
**Configuration parameters.** Unless otherwise mentioned, we configured Swayam to satisfy a default SLA with a desired service level of 99%, a response time threshold of $5\bar{c}$, where $\bar{c}$ denotes the service's mean computation time, a burst threshold of 2x, and *no* request pruning. We chose 2x as the default burst threshold to tolerate the noise observed in the arrival rates of the 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 §5.2). 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 to the clients.

Since the employed traces represent already active services, each service was pre-provisioned with five backends before replaying the trace, for bootstrapping. Traces 1 and 2 were configured with the default SLA. Trace 3 was configured with a higher burst factor of 8x 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.

**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 $t - 100s$.




(a) Request rates (trace 1) (b) Request rates (trace 2) (c) Request rates (trace 3)

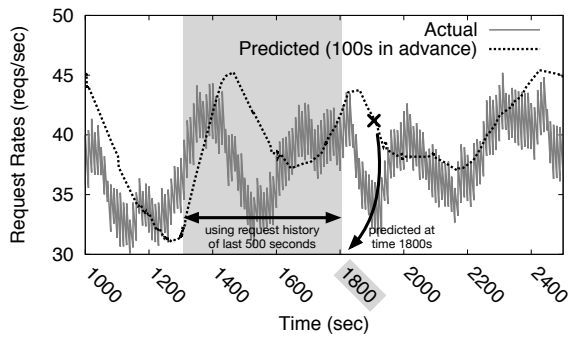
**Figure 6: Actual and predicted request rates at FE0 for production traces 1, 2, and 3. Results for other frontends are similar.**

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 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. 7(d)–7(f). Trushkowsky et al. [30] measured SLA compliance by computing the average 99$^{\text{th}}$ 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 *stricter* 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 [1000$s$, 2500$s$] is magnified and illustrated in the figure below.

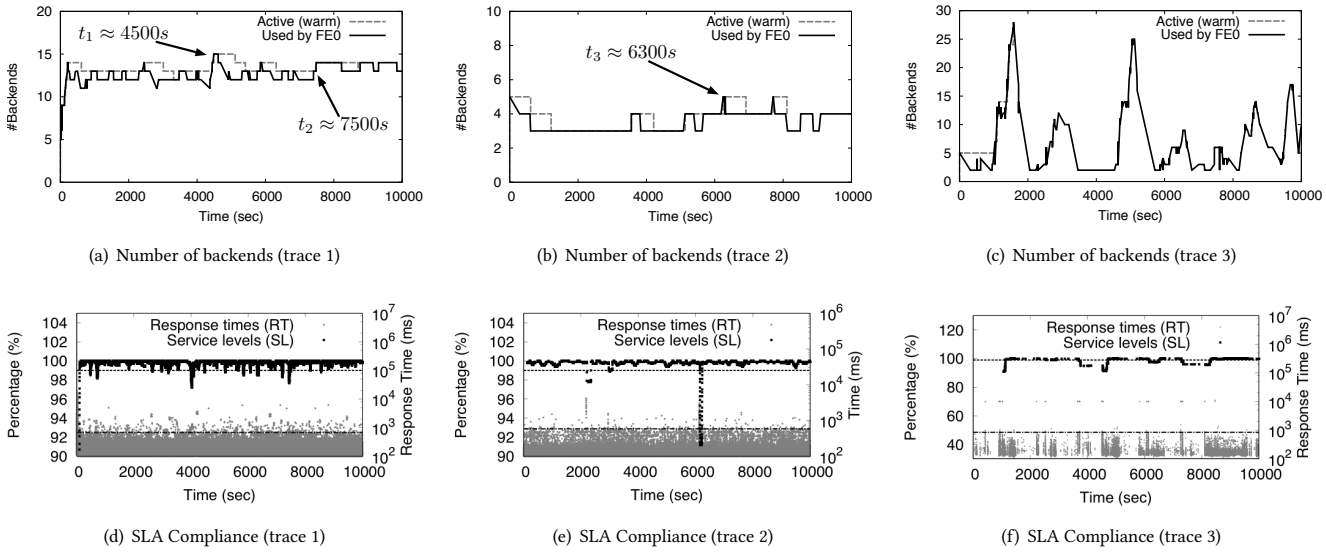



We observe that there are small frequent oscillations in the request rates, and that the peaks and troughs in the predicted request rate corresponding to those in the actual request rate are slightly shifted to the right. Swayam ensures that this behavior does not affect the end results since the system smoothes the effect of oscillations on the estimated resource demand (§4.4). In particular, the number of warm-in-use backends does not change frequently and it is reduced only if the drop in request rates is observed over a longer period of time (e.g., ten minutes). Furthermore, Swayam holds in reserve some servers (warm not-in use) for a while before they self-decommission.

Figs. 7(a)–7(c) validate the efficacy of the distributed scale-in and scale-out mechanisms used by Swayam. For example, at time $t_1 \approx 4500s$, due to a spike in the request rates as seen in Fig. 6(a), the frontend decides to scale-out the number of backends to fifteen (see Fig. 7(a)). 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. 7(d), 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 quickly scales-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 (see Fig. 7(e)) are similar, i.e., a few SLA violations occur during the steep spike at time $t_3 \approx 6300s$, and SLA compliance otherwise. Results for trace 3 (see Fig. 7(f)), however, are different owing to its unique arrival pattern. In Fig. 6(c), requests in trace 3 arrive almost periodically. In




(a) Number of backends (trace 1)

(b) Number of backends (trace 2)

(c) Number of backends (trace 3)

(d) SLA Compliance (trace 1)

(e) SLA Compliance (trace 2)

(f) SLA Compliance (trace 3)

**Figure 7: 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. Figs. 7(d)–7(f) illustrate the SLA compliance for production traces 1, 2, and 3, respectively. Response time thresholds for the three traces are $5\bar{c}_1 = 675$ ms, $5\bar{c}_2 = 583$ ms, and $5\bar{c}_2 = 833$ ms.**

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 traces 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 perfectly load balanced. 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. 8(a) shows that request pruning improves SLA compliance significantly during spikes without compromising on the SLA compliance during the steady states (w.r.t. Fig. 7(e)). This improvement is because **(1)** Swayam does not have strict admission control at its entry point, but it prunes a request only after its waiting time in the system has exceeded the response time threshold of its service; and **(2)** in a randomized load balancer like Swayam's, requests accumulated during spikes affect the requests that arrive later, which 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

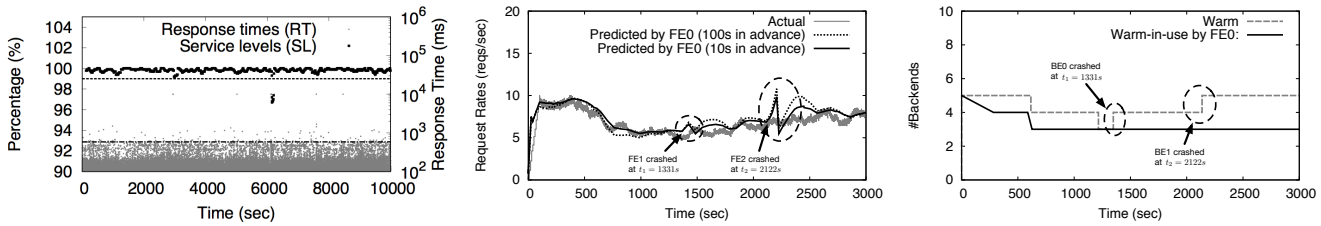
Swayam tolerates crash failures by incorporating a gossip protocol on top of the underlying request-response protocol, which ensures that all frontends have a consistent view of the total number of active frontends for each service. If a backend crashes, each frontend independently discovers this using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

Fig. 8(b) 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 from §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. 8(c) 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

Recall that Swayam's objective is to greedily minimizing the number of warm backends used by each service, while ensuring SLA compliance for that service. In the following, we thus measure




(a) Trace 2 with request pruning (b) Trace 2 with frontend failures (c) Trace 2 with backend failures

**Figure 8: (a) SLA compliance for production trace 2 with request pruning enabled. (b) and (c) Effect of frontend crash failures and backend crash failures on production trace 2, respectively.**

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.

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. 9, we illustrate the resource usage for all 15 production traces, when replayed with autoscalers ClairA1, ClairA2, and Swayam. The traces were each executed for about 3000s. We also denote in Fig. 9 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. Moreover, reducing startup times has large efficiency benefits. Comparing ClairA2 to Swayam, for certain services (traces 1, 2, and 12-15), we oberve that 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 because: **(1)** We measure SLA compliance with respect to a finite number of requests and not with respect to a finite interval of time. Hence, during instantaneous bursts, we record significantly many SLA compliance failures, as opposed to just one (i.e., our evaluation metric is biased *against* Swayam). **(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 by realizing significant resource savings at the cost of 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 unused 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. In addition, Swayam's resource-estimation model can be extended to account for vertical scaling, by modeling concurrent execution of requests in the

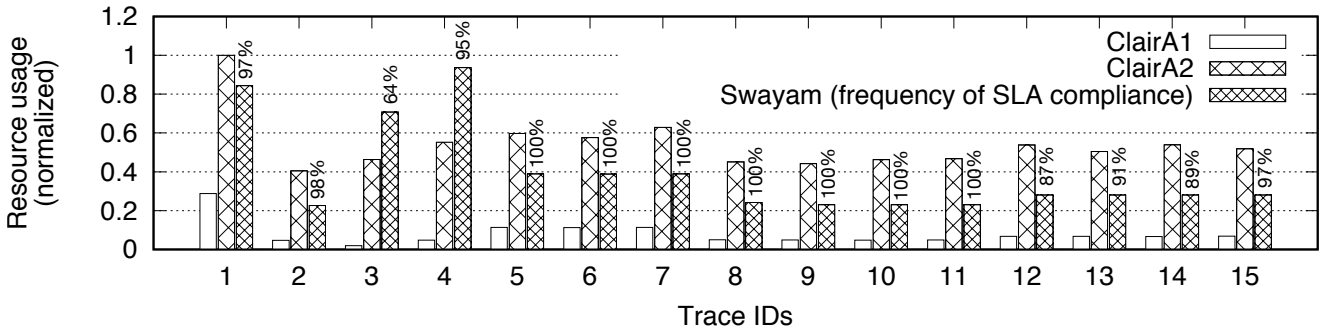



**Figure 9: Total resource usage (in terms of warm-backend-time, normalized with respect to the maximum resource usage) for all 15 traces when run with autoscalers ClairA1, ClairA2, and Swayam. Trace IDs 1, 2, and 3 correspond to Figs. 6(a)–6(c), respectively. For Swayam, we also denote for each trace the frequency of SLA compliance, i.e., how often does Swayam comply with the SLAs over the entire experiment duration. Note that ClairA1 and ClairA2 are designed to always comply with the SLAs (i.e., a 100% SLA compliance frequency). The garbage collection period was five minutes.**

backend. Swayam's approach is applicable to other stateless web services, if they have characteristics similar to that of ML inference requests. Note that supporting online or incremental updates of the ML model is orthogonal to the autoscaling problem, since it may require recompilation of the service containers.

The appendix and codebase are available at [17].

## REFERENCES


[1] 2017. Amazon Machine Learning - Predictive Analytics with AWS. (2017). https://aws.amazon.com/machine-learning/

[2] 2017. Google Cloud Prediction API Documentation. (2017). https://cloud.google.com/prediction/docs/

[3] 2017. HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. (2017). http://www.haproxy.org/

[4] 2017. Instance-based learning. (2017). https://en.wikipedia.org/wiki/Instance-based_learning

[5] 2017. Machine Learning - Predictive Analytics with Microsoft Azure. (2017). https://azure.microsoft.com/en-us/services/machine-learning/

[6] 2017. Smartphone-Based Recognition of Human Activities and Postural Transitions Data Set. (2017). http://archive.ics.uci.edu/ml/datasets/Smartphone-Based+Recognition+of+Human+Activities+and+Postural+Transitions

[7] 2017. Watson Machine Learning. (2017). http://datascience.ibm.com/features#machinelearning

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*.

[9] Constantin Adam and Rolf Stadler. 2007. Service middleware for self-managing large-scale systems. *IEEE Transactions on Network and Service Management* 4, 3 (2007), 50–64.

[10] Enda Barrett, Enda Howley, and Jim Duggan. 2013. Applying Reinforcement Learning Towards Automating Resource Allocation and Application Scalability in the Cloud. *Concurrency and Computation: Practice and Experience* 25, 12 (2013), 1656–1674.

[11] Lawrence Brown, Noah Gans, Avishai Mandelbaum, Anat Sakov, Haipeng Shen, Sergey Zeltyn, and Linda Zhao. 2005. Statistical analysis of a telephone call center: A queueing-science perspective. *Journal of the American statistical association* 100, 469 (2005), 36–50.

[12] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. 2008. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *NSDI*. 338–350.

[13] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*. 269–284.

[14] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2016. Clipper: A Low-Latency Online Prediction Serving System. *arXiv preprint arXiv:1612.03079* (2016).

[15] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80.

[16] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael Kozuch. 2011. Distributed, Robust Auto-Scaling Policies for Power Management in Compute Intensive Server Farms. In *OCS*.

[17] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. 2017. Swayam: online appendix. (2017). https://people.mpi-sws.org/~bbb/papers/details/middleware17

[18] Varun Gupta, Mor Harchol-Balter, JG Dai, and Bert Zwart. 2010. On the inapproximability of M/G/K: why two moments of job size distribution are not enough. *Queueing Systems* 64, 1 (2010), 5–48.

[19] Rui Han, Moustafa M Ghanem, Li Guo, Yike Guo, and Michelle Osmond. 2014. Enabling Cost-Aware and Adaptive Elasticity of Multi-Tier Cloud Applications. *Future Generation Computer Systems* 32 (2014), 82–98.

[20] Brendan Jennings and Rolf Stadler. 2014. Resource Management in Clouds: Survey and Research Challenges. *Journal of Network and Systems Management* (2014), 1–53.

[21] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[22] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing* 12, 4 (2014), 559–592.

[23] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R Larus, and Albert Greenberg. 2011. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation* 68, 11 (2011), 1056–1071.

[24] Michael Copeland. 2016. WhatâĂŹs the Difference Between Deep Learning Training and Inference? (2016). https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/

[25] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[26] Michael David Mitzenmacher. 1996. *The Power of Two Choices in Randomized Load Balancing*. Ph.D. Dissertation. UNIVERSITY of CALIFORNIA at BERKELEY.

[27] Ohad Shamir. 2014. Fundamental limits of online and distributed algorithms for statistical learning and estimation. In *Advances in Neural Information Processing Systems*. 163–171.

[28] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. 2013. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *ICAC*.

[29] Chandramohan A Thekkath, Timothy Mann, and Edward K Lee. 1997. Frangipani: A scalable distributed file system. In *ACM Symposium on Operating Systems Principles (SOSP)*. 224–237.

[30] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. 2011. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements.. In *FAST*. 163–176.

[31] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. 2008. Agile Dynamic Provisioning of Multi-Tier Internet Applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 3, 1 (2008), 1.

[32] Fetahi Wuhib, Rolf Stadler, and Mike Spreitzer. 2010. Gossip-based resource management for cloud environments. In *2010 International Conference on Network and Service Management*. IEEE, 1–8.