Model Selection for Latency-Critical Inference Serving

Daniel Mendoza
dmendo@stanford.edu
Stanford University

Francisco Romero
faromero@stanford.edu
Stanford University

Caroline Trippel
trippel@stanford.edu
Stanford University

Abstract

In an inference service system, *model selection and scheduling (MS&S)* schemes map inference queries to trained machine learning (ML) models, hosted on a finite set of workers, to solicit accurate predictions within strict latency targets. MS&S is challenged by both varying query load and stochastic query inter-arrival patterns; however, state-of-the-art MS&S approaches conservatively account for load exclusively.

In this paper, we first show that explicitly considering inter-arrival patterns creates opportunities to map queries to higher-accuracy (higher-latency) models during intermittent arrival *lulls*. We then propose RAMSIS, a framework for generating MS&S policies that exploits this finding. RAMSIS leverages a statistical problem model of query load and inter-arrival pattern to produce policies that maximize accuracy given some latency target. We evaluate RAMSIS-generated MS&S policies alongside state-of-the-art approaches. Notably, RAMSIS requires as low as 50.00% (on average 18.77%) fewer resources to achieve the same accuracy for an ImageNet image classification task given 26 trained models.

CCS Concepts: • **Computing methodologies** → **Machine learning**; **Planning under uncertainty**; **Markov decision processes**.

Keywords: inference serving systems, model selection, query scheduling, systems for machine learning, machine learning for systems, Markov decision processes

ACM Reference Format:
Daniel Mendoza, Francisco Romero, and Caroline Trippel. 2024. Model Selection for Latency-Critical Inference Serving. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3627703.3629565

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629565

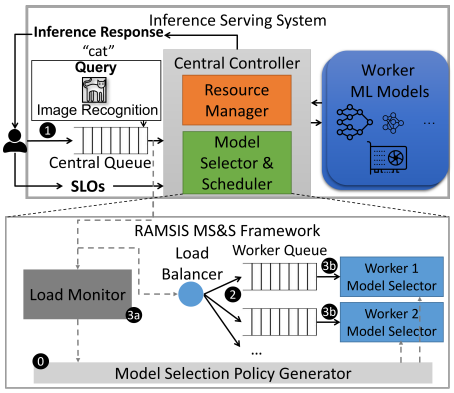

Figure 1. Inference applications specify SLOs (e.g., latency and/or accuracy) and submit queries to the ISS, which implements a resource manager and a model selector & scheduler (§1) to derive predictions given SLO constraints.

1 Introduction

Machine learning (ML) inference has proliferated to many application domains, e.g., recommendation [13], finance [12], analytics [55], computer vision [21], healthcare [24], and natural language processing [10, 34]. Thus, modern hyperscalars devote significant systems infrastructure to *inference serving*—the task of soliciting accurate predictions for inference queries (i.e., inference requests) from a corpus of trained ML models. For example, at Amazon Web Services, ML inference consumes more than 90% of infrastructure costs [2]. At Meta, more than 200 trillion predictions are made each day [27].

Inference serving systems (ISSs) [6, 7, 15, 16, 32, 38, 43, 54, 57] have been designed to produce accurate predictions for application inference queries while satisfying agreed-upon latency constraints and maintaining low infrastructure costs. They commonly feature five main components, depicted in Fig. 1 (top): (1) a central queue, (2) trained ML models, (3) workers, (4) a resource manager [6, 16, 38, 54], and (5) a model selector and scheduler [16, 32, 38, 57].

Applications submit inference queries (e.g., images or text segments) that are buffered at a *central queue*. Queries are then dispatched from the central queue and assigned to *workers*, i.e., compute resources which host the ML *models* and serve batches of inference requests.

As inference queries are often user-facing, ISSs must abide by strict *service level objectives (SLOs)*—generally constraints on response latency [6, 7, 15, 32, 43, 54, 57], but sometimes

constraints on inference accuracy [16, 38]. For example, inference applications (e.g., ads, new feed, facial tag recommendation) typically require response latencies within tens to hundreds of milliseconds [17, 20].

To co-optimize for latency, accuracy, and cost, ISSs typically implement a *central controller*, which consists of a *resource manager* and a *model selector and scheduler*. Resource managers handle resource allocation, which includes provisioning workers to run ML inference and maintaining a pool of loaded models at each worker. Given a finite set of workers and loaded models, model selectors and schedulers employ *model selection and scheduling (MS&S)* schemes to assign queries to models on workers that can satisfy their SLOs. This process requires navigating accuracy-latency tradeoffs, since more accurate models typically exhibit longer inference latencies [4, 16, 32, 38, 57], and even hardware performance characteristics [16, 32, 38, 57].

Clearly, an ISS's resource manager directly impacts its infrastructure costs. However, the MS&S scheme (our focus) determines achievable inference accuracy within a latency SLO when resources are fixed [16, 32, 57]. Notably, MS&S is challenged by varying *query load* (i.e., query arrival rate) and stochastic *query inter-arrival patterns* [1, 6, 17, 20, 26, 37, 38, 54, 57], which are characteristic of real-world inference applications. Shifts in query load impact the set of models capable of satisfying an application's latency SLO. Plus, stochastic inter-arrival patterns (e.g., Poisson [17]) lead to unexpected arrival bursts, enabling MS&S decisions for one batch of queries to adversely affect future queries.

To cope with the above challenges, ISSs (e.g., INFaaS [38], Jellyfish [32], ModelSwitching [57], Cocktail [16]) implement *load-granular* MS&S schemes, which select models and workers whose throughput can sustain a given query load regardless of the inter-arrival pattern. However, we observe that ignoring the inter-arrival pattern results in *overly-conservative* MS&S decisions and missed opportunities to achieve higher inference accuracy or the same accuracy at lower cost.

This Paper. Given our observation, we propose *Random Arrival Model Selection for Inference Serving (RAMSIS)*, an MS&S framework that explicitly accounts for variable query load *and* stochastic inter-arrival patterns in order to achieve *maximally accurate* predictions for queries within a user-defined latency SLO. **Our key insight** in designing RAMSIS is that stochastic inter-arrival patterns exhibit arrival *lulls* (in addition to bursts), during which slower, more accurate models can be *safely* selected for queries.

During its *offline phase*, RAMSIS formulates the MS&S problem as a *Markov Decision Process (MDP)* [36, 39, 42]. Naïvely formulated, states consist of central queue states (i.e., finite queues of pending query deadlines) and worker statuses (e.g., busy or available), and actions are MS&S decisions. Central to our approach, transition probabilities between states are derived from a *query arrival distribution*: in our experiments, a Poisson distribution [17, 37, 38, 54, 57] which is parameterized by query load.

The naïve MDP formulation above is computationally challenging to solve as it requires modeling all possible central queue states. Thus, **Our second insight** is the state space can be greatly simplified in three ways. First, since inference latency is predictable [15, 31, 43], the finite set of models loaded on workers and supported batch sizes quantize the space of satisfiable deadlines, i.e., RAMSIS policies do not require a continuous time space for deadlines. Second, RAMSIS policies need only account for the earliest deadline when serving a batch of queued queries; an MS&S decision that meets the earliest deadline in a batch meets the rest. Third, RAMSIS decomposes the MS&S problem into two sub-problems: *query load balancing* (assigning queries to workers) and *worker-level model selection* (assigning queries to models) [57]. Doing so reduces the MDP state space by enabling RAMSIS to derive model selection (MS) policies per-worker, independent of other workers. Given simplified worker-level MDPs (which account for a query load balancing strategy) and a reward function, RAMSIS uses an exact solution method (value iteration [36, 39, 42]) to derive optimal MS policies that maximize overall accuracy and minimize latency SLO violations.

During its *online phase*, RAMSIS performs query load balancing following a round-robin strategy and conducts worker-level model selection dynamically according to the pre-computed MS policies. Overall, RAMSIS achieves higher accuracy per inference query than state-of-the-art systems with the same resources and latency SLO violations, or the same accuracy and latency SLO violations with fewer resources. We summarize our contributions as follows:

- We demonstrate that existing MS&S approaches are overly-conservative in the presence of stochastic inter-arrival patterns which give rise to periods of arrival lulls.
- We show that an MS&S policy which accounts for query inter-arrival patterns can be efficiently pre-computed offline by leveraging predictable ML model latencies and decomposing the MS&S problem into simpler sub-problems.
- We design RAMSIS, a framework for generating MS&S policies, which provide probabilistic guarantees on inference accuracy and latency.
- Using a real-world inference query trace from Twitter [1], we evaluate RAMSIS-generated MS&S policies alongside state-of-the-art approaches. RAMSIS achieves the same accuracy as state-of-the-art approaches with as low as 50.00% (on average 18.77%) fewer resources on an image classification task. For a text classification task, RAMSIS requires as low as 75.00% (on average 28.28%) fewer resources.

2 Background and Motivation

In this work, we consider an ISS architecture like the one in Fig. 1 (top), consistent with prior work [7, 15, 32, 38, 43, 57].

Queries from inference applications arrive at a central queue and are subsequently assigned to models on workers by a model selector and scheduler. A resource manager strives to minimize infrastructure cost through model and worker scaling [6, 38, 54]. Each worker has one or more pre-loaded models. For inference latency predictability, each worker's hardware resources are assumed to be isolated and workers execute one model at a time [15, 32, 43]. However, approaches which do not assume isolation are still applicable to the assumed ISS architecture [7, 38, 57].

2.1 Challenges in Model Selection & Scheduling

Our focus is on improving the model selector and scheduler component of ISSs. Given fixed resources (workers and loaded models), the model selector and scheduler ideally produces the highest possible inference accuracy per query while meeting response latency constraints. Such a goal is challenged by variable *query load* and stochastic *inter-arrival patterns* [1, 6, 17, 20, 26, 37, 38, 54, 57].

Production inference workloads exhibit variable query load [1, 6, 17, 20, 26, 37, 38, 54, 57], typically calculated as average query arrivals over a time interval. Query load impacts the set of models which are able to satisfy a query's latency SLO, since different models offer different throughput.

Inference query traces also exhibit stochastic inter-arrival patterns, i.e., variance in the time elapsed between query arrivals at a constant query load. Prior work observes that the inter-arrival patterns of inference workloads conform to a Poisson process [17, 37, 38, 54, 57], exhibiting intermittent arrival bursts and lulls. Ideally, the model selector and scheduler would select lower latency models during bursts and higher latency models during lulls. However, the stochastic nature of query inter-arrivals makes it difficult to ensure that an optimistic MS&S decision for one query will not adversely impact future queries. Unexpected bursts in query arrivals can quickly overload workers, resulting in latency SLO violations [20, 38, 54].

2.2 Model Selection and Scheduling: Limitations

MS&S schemes in state-of-the-art ISSs (e.g., INFaaS [38], Cocktail [16], Jellyfish [32], ModelSwitching [57]) conservatively account for query load exclusively, which is sufficient to mitigate SLO violations. Such a *load-granular* approach employs an offline profiling step to characterize the accuracy of each model (using an application-provided test set) as well as the response latency and throughput of each model running on each worker [16, 32, 38, 57]. During inference serving (online), MS&S decisions ensure that the throughput and response latency of the target (worker, model) pair can meet the current query load and latency SLO, respectively. That is, the query load uniquely defines what model to run for each worker. Moving averages or neural networks are used to anticipate query load for short time windows (one second to one minute in the future) [16, 32, 38, 57].

ISS	MS	Latency	Accuracy	Constraints
Clipper [7]	-	SLO	-	-
Nexus [43]	-	SLO	-	D
Clockwork [15]	-	SLO	-	D
MArk [54]	-	SLO	-	-
Inferline [6]	-	SLO	-	-
INFaaS [38]	X	min	SLO	-
Cocktail [16]	X	min	max	P, E
Jellyfish [32]	X	SLO	max	D
ModelSwitching [57]	X	SLO	max	-
RAMSIS (this paper)	X	SLO	max	D

Table 1. Key features of ISSs. D: assumes deterministic, predictable inference response latency, E: model ensembling, P: preemptible workers. ISSs without a model selection (MS) component rely on users to select models. All ISSs here perform query scheduling, or mapping queries to workers.

Moreover, load granular MS&S decisions aim to maximize accuracy while accounting for the above throughput and latency restrictions [16, 32, 57] or accept accuracy SLOs as input [38].

In this paper, we propose an MS&S approach which aims to maximize an application's inference accuracy while meeting its latency SLO. Thus, ModelSwitching [57] and Jellyfish [32] are the most relevant points of comparison, as highlighted in Table 1 and explained in further detail in §7. Other MS&S schemes from the literature, namely those implemented in Cocktail [16] and INFaaS [38], are less relevant for the following reasons. Cocktail specifically targets MS&S for model ensembling with preemptible workers, which is outside the scope of our assumed ISS architecture (Fig. 1). INFaaS requires an accuracy SLO and chooses the lowest cost model (typically lowest latency/accuracy model) that satisfies it. Thus, INFaaS differs in its objective and constraints (see §H for more discussion).

We observe that by ignoring query inter-arrival patterns, load-granular MS&S approaches satisfy latency SLOs at the cost of overly-conservative MS&S decisions. We illustrate this point in Fig. 2, which depicts the same timeline of query inter-arrivals subject to two distinct MS&S schemes: the load-granular approach (left) and our approach, RAMSIS (right). In both scenarios, two models (A and B) are pre-loaded on both Worker 1 and Worker 2. Assume that both models' response latencies (on both workers) are less than the latency SLO of incoming queries, but only model B has sufficient throughput to meet the query load. Fig. 2 (left) shows a load-granular MS&S approach would select model B for all queries.

We show that an MS&S policy which explicitly accounts for query inter-arrival patterns can leverage intermittent arrival lulls to proactively select higher accuracy models *while avoiding SLO violations*. Fig. 2 (right) highlights the benefits of this approach where model A is occasionally selected during arrival lulls, thereby achieving higher overall

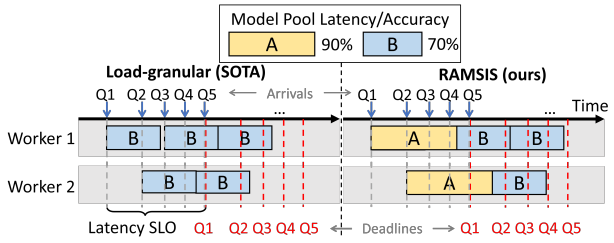

Figure 2. State-of-the-art MS&S versus RAMSIS, assuming the same query load and inter-arrival pattern. Blue arrows denote query arrivals over time. Red dotted lines denote query deadlines. Each query has the same latency SLO. RAMSIS achieves higher accuracy with the same latency SLO violations (none) by accounting for the inter-arrival pattern when selecting between models A and B.

average inference accuracy than the load-granular approach with the same resources.

3 RAMSIS Overview

We present *Random Arrival Model Selection for Inference Serving (RAMSIS)*, a framework for automatically generating (offline) and deploying (online) load- and inter-arrival pattern-aware MS&S policies. We summarize RAMSIS in this section and give the details of its design in §4.

3.1 Offline Phase: MS&S Policy Generation

Offline (pre-deployment), RAMSIS formulates the MS&S problem as an MDP [36, 39, 42] that captures the central queue state, query load, and query inter-arrival pattern. With respect to our assumed ISS architecture in Fig. 1 (top), the MS&S problem refers to the task of mapping a batch of inference queries in the central queue to a loaded model on a particular worker. The goal of this mapping is to ensure that the latency SLO violation rate is held below an acceptable threshold, while the average inference accuracy is maximized. MDPs are a natural choice for modeling the MS&S problem where outcomes are partially stochastic due to the query arrival pattern and partially determined by a decision making policy [36, 39, 42]. RAMSIS generates MS&S policies that are each specialized for a particular query inter-arrival pattern and response latency SLO, given the available resources (workers) and pre-loaded models for an application as allocated by the resource manager.

3.1.1 Offline Inputs.

To formulate the MDP problem model, RAMSIS requires the following inputs:

- **Response latency SLO:** We define an application's response latency SLO as the maximum time that an ISS may take to produce an inference response after receiving an inference request at its central queue.
- **Query arrival distribution:** The query arrival distribution $PF(k, T)$ defines the probability of k queries arriving during a time interval of length T; it is a function of query load and captures the stochastic nature of the inter-arrival pattern. For web-services, including inference-serving, the query arrival distribution is typically the Poisson distribution [17, 37, 38, 54, 57], as assumed in our experiments (§7). However, RAMSIS is parameterized by the arrival distribution (e.g., the Gamma distribution could be used [28]). Upon encountering an unexpected inter-arrival pattern with no corresponding pre-computed policy, RAMSIS generates a new one (§3.2.2).
- **Latency profiles:** As in prior work [15, 32, 38, 43], to anticipate the latency of MS&S decisions, RAMSIS requires profiling the inference latency for all models on all workers with all supported batch sizes. Note that only (worker, model, batch size) triples which exhibit a latency less than the application's latency SLO are relevant.[1] We define inference latency $l_w(m, b)$ as the time elapsed between sending a batch of b queries from the central queue to model m on worker w and receiving an inference response at the central controller.
- **Inference accuracy profile:** To assess the accuracy of a MS decision, RAMSIS collects an inference accuracy profile $Accuracy(m)$ for each trained ML model m. As in prior work [16, 32, 38, 57], accuracy profiles are collected with respect to a test set provided by the application (§7).

3.1.2 Simplifying Policy Generation.

A naïve formulation of the MS&S problem as an MDP is challenged by state space explosion. States are most naturally comprised of central queue states (i.e., finite queues of pending query deadlines) and worker statuses (i.e., busy or available). Fortunately, due to the predictability of ML model inference latencies [15, 31, 43], discrete time is sufficient to represent all possible central queue states consisting of *relevant* deadlines. Nevertheless, a direct formulation of the MS&S problem as a discrete time MDP [42] still requires an exponential state space. Assuming a discrete time space of size D, a maximum queue size of N, and K workers, the number of possible MDP states is $O(N^D + K^D)$. This state space complexity limits the efficiency and scalability of MS&S policy generation. For example, with $N = 32$, $D = 100$ (as required for our experiments, §4.2.2), and $K = 1$, we find that policy generation via value iteration [36, 39, 42] does not complete after a 24 hour timeout. Since accounting for the entire arrival distribution is computationally hard, existing MS approaches [16, 32, 38, 57] are *conservatively* load-granular—the load uniquely determines the model to select.

To avoid an exponential state space, we simplify RAMSIS's MDP in two ways. First, as shown in Fig. 1 (bottom), we decompose the MS&S problem into *query load balancing* (§3.2.1) and *model selection* (§3.2.2) components. Load balancing maps queries in the central queue to *worker queues*,

[1] If the same worker types exhibit near identical profiles, (*worker type*, model, batch size) triples are sufficient. This is the case in our experiments (§7).

so that model selection, including batch selection, can be performed per-worker, independent of other workers. With this problem split, we can construct per-worker MDPs that capture the worker-level arrival process (which accounts for the load balancing policy) and worker-level state space only. Second, RAMSIS models each worker queue state as a tuple (n, T_j), where n is the number of queued queries, and T_j is the time remaining to service the query with the earliest (i.e., strictest) deadline (§4.2). This approach is a simplification over explicitly tracking n deadlines.

Together, the design choices above reduce the state space size from exponential to polynomial, enabling MS&S policies to be efficiently derived and verified to meet SLOs.

3.1.3 Offline Outputs. RAMSIS's policy generator ⓪ (in Fig. 1) constructs per-worker MS policies. Since MS policy generation with RAMSIS is specialized to a query arrival distribution (§3.1.1), which is a function of the arrival load and inter-arrival pattern, the policy generator pre-computes a *set* of policies. In our experiments (§7), we assume a Poisson arrival distribution and direct RAMSIS to compute MS policies for a range of expected query loads, as in prior work [28, 57].

3.2 Online Phase: Inference Serving

Online, RAMSIS performs load balancing across workers and worker-level MS. The latter is performed according to the offline-generated policies which assume the former.

3.2.1 Load Balancing: Assigning Queries to Workers. Per Fig. 1, all incoming queries arrive at the central queue ❶. Upon arrival of query q at time t_q, it is assigned a *deadline* $t_q + SLO$ (the arrival time plus the response latency SLO).

We design RAMSIS's load balancer ❷ to distribute the queries in the central queue to worker queues in a *round-robin* manner, as it encourages high resource utilization [18]. However, this choice is not fundamental; RAMSIS can be extended in a straightforward manner to use other load-balancing strategies, e.g., shortest-queue-first [18]. Doing so requires changes to only the MDP transition probabilities, as defined in §4.4.2. All other components of RAMSIS are unchanged.

3.2.2 Model Selection: Assigning Queries to Models. During inference serving, query load is tracked by a load monitor ❸ₐ. Given a set of offline-generated MS policies specialized for differing inter-arrival patterns, worker-level model selectors use the *lowest-load MS policy* that meets the anticipated query load. If that anticipated load is higher than any pre-computed MS policy can support, a new one is generated.

Each worker is associated with a local worker queue to buffer queries distributed by the load balancer. Per-worker model selectors ❸ᵦ service queries from their worker queue in deadline order (earliest first) according to their offline-generated MS policies. Each MS decision considers the worker queue state (§3.1.2) to select a pre-loaded model m and batch size b, directing b queries with the earliest deadlines to run on m.

4 MS&S Problem Formulation

Per §3.1.2, RAMSIS formulates the per-worker model selection problem as a discrete time MDP. MDPs provide a mathematical framework for modeling decision making problems where outcomes are partially stochastic and partially controlled by a decision making policy [36, 39, 42]. As such, they are commonly used in scheduling under uncertainty for applications such as queueing systems, robotics, and manufacturing [3, 23, 36, 39, 42, 44, 51].

An MDP is a stochastic control process that defines a finite state transition system, denoted by (S, A, P_a, R_a) [36, 39, 42], where:

- S is a set of *states* (worker queue states, §4.2).
- A is a set of *actions* (model and batch size decisions, i.e., MS decisions, §4.3).
- $P_a(s, s') = \mathrm{P}[s_{d+1} = s' | s_d = s, A_d = a]$ is the probability that action $a \in A$ taken in state $s \in S$ at decision epoch d will result in a transition to state s' for decision epoch $d + 1$ (§4.4).
- $R_a(s, s')$ defines the reward received for transitioning to state s' by taking action $a \in A$ in state $s \in S$ (§4.1).

At each *decision epoch* d, a decision making policy can take any action $a \in A$ that is valid in the current state $s \in S$. Taking action a in s results in a random transition to next state $s' \in S$ with transition probability $P_a(s, s')$, giving the decision maker a reward $R_a(s, s')$. Notably, MDPs exhibit the *Markov property*: the probability of transitioning from state s to state s' given action a, $P_a(s, s')$, solely dependents upon a and s [36, 39, 42].

4.1 Policy Generation

A worker-level MS policy is designed to maximize the joint expectation of *accuracy per query* and *latency SLO satisfaction* over the state space of the worker's MDP. Thus, $R_a(s, s') = Accuracy(a) * SLOSatisfied(s, a)$. $Accuracy(a)$ returns the inference accuracy of the model selected by action a; $SLOSatisfied(s, a)$ returns a Boolean value indicating whether action a, an MS decision, satisfies the strictest deadline in the batch of queries selected for inference from state s.

We use an exact solution method to generate an optimal MS policy given a worker's MDP, namely value iteration [36, 39, 42]; other exact solution methods, like policy iteration [36], may be used. Value iteration [36, 39, 42] is a dynamic programming algorithm that repeatedly updates estimates of the expected reward of each state (i.e., the *value* of each state) by considering the maximum expected reward of all possible next states. It iteratively refines state values

until convergence at which point the MDP is solved and an optimal (MS) policy is produced.

Per-worker MS policies generated this way are optimal with respect to the arrival distribution to the worker as captured by the transition probabilities of the MDP. Transition probabilities are defined by the query arrival distribution to the central queue and the load balancing strategy (i.e., round-robin) (§4.4).

4.2 States

A state $s \in S$ in a worker w's MDP formulation encodes its worker queue state as a tuple: $s = (n, T_j)$. Recall that n is the number of queries in w's worker queue, and T_j is the amount of time remaining to service the query with the earliest deadline (§3.1.2); we call T_j the *slack time* of the queued query in s with the earliest deadline.

Slack times are continuous in general and induce an infinite state space for a worker's MDP. However, a continuous representation of slack times is unnecessary (§3.1.2).

We present two discretization strategies used by RAMSIS in §4.2.1 and §4.2.2. The result is a finite sequence of unique *time lengths*, $\mathbf{T}_w = (T_0, T_1, ...)$, where each T_j represents a possible slack time and $T_j < T_{j+1}$. Note that we use j to index into $\mathbf{T}_w$; $\mathbf{T}_w[j] = T_j$. With our time discretization, every continuous slack time Δ is represented with a time length $T_j \in \mathbf{T}_w$, where $T_j \leq \Delta < T_{j+1}$.

4.2.1 Model-Based Discretization. RAMSIS's slack time discretization need only be able to identify which actions (m, b), or (model, batch size) pairs, are *valid* in a given state. For worker w's MDP, action $a = (m, b)$ is valid in state $s = (n, T_j)$ if the inference latency of running model m with batch size b on w satisfies the slack time T_j (i.e., $l_w(m, b) \leq T_j$).

Suppose worker w has M_w models, and B_w is the largest batch size that meets the inference application's latency SLO among all $m \in M_w$. Then, the upper bound on unique inference latencies exhibited by w, which meet the latency SLO, is $|M_w| * B_w$. So, $O(|M_w| * B_w)$ distinct time lengths are sufficient to represent all *relevant* slack times.

From the above observation, we propose *Model-based Discretization (MD)* which defines $\mathbf{T}_w$ as the sequence of all unique inference latencies $l_w(m, b)$ for worker w, for all $m \in M_w$ where $b \leq B_w$.

4.2.2 Fixed Length Discretization. With MD (§4.2.1), a worker w's MDP state space grows linearly with the number of loaded models $|M_w|$ and maximum batch size B_w. However, this reduced state space still poses scalability challenges if $|M_w|$ and B_w are large. Given the experimental setting detailed in §7.3.2 with $|M_w| = 60$ models per worker and $B_w = 29$, policy generation (§4.1) with MD does not complete within 24 hours, as shown in Table 2.

For cases where MD is too costly (e.g., tens to hundreds of models per worker), we propose *Fixed Length Discretization (FLD)*. FLD defines $\mathbf{T}_w = 0, \frac{SLO}{D}, 2 * \frac{SLO}{D}, ..., SLO$, where SLO is the application latency SLO and D is a hyperparameter.

TD	Batch	$\lvert M_w \rvert = 9$ Runtime (s)	$\lvert M_w \rvert = 60$ Runtime (s)
MD	variable	3693.53	timeout
FLD $D = 100$	variable	3014.89	timeout
MD	max	115.50	timeout
FLD $D = 100$	max	132.20	1355.09
FLD $D = 10$	max	14.62	148.87

Table 2. Policy generation runtimes for different time discretization (TD) and batching strategies (§4.3.2) for an image classification task (§7) when $B_w = 29$. Timeout is 24 hours.

D is a knob for reducing the policy generation runtime that may come at the cost of generating overly-conservative MS policies that miss opportunities to select higher accuracy models. That is, FLD may *underestimate* the real (continuous) slack time associated with a worker queue state (i.e., $T_j << \Delta$). Empirically, we find that using FLD with large enough D (e.g, $D = 100$ in our evaluation) produces equally performant policies as MD (§C).

4.2.3 Full Queue State. We assume worker queues are of finite size N_w for worker w. We reserve a special state $(\phi, \varnothing)$ to represent the situation where the number of queries in a worker queue accumulate beyond the maximum queue size N_w (i.e., $s = (n, T_j)$ for $n > N_w$). For this state, we assume the worker queue size is truncated to N_w (i.e., $\phi = N_w$), indicating $n - N_w$ queries are dropped. We conservatively assume the slack time is zero in this state (i.e., $\varnothing = 0$) which indicates the earliest deadline is not satisfiable by any action. Generated policies are thus encouraged to avoid this state by the reward function (§4.1).

The maximum queue size N_w need only be large enough to accommodate the maximum batch size B_w. This is because accumulating a number of queries in a worker queue beyond B_w is an indication that the ISS cannot meet the latency SLO under the query load, i.e., there are not enough workers and/or the loaded models are too slow. In our evaluation, we use $N_w = 32$ for $B_w = 29$ (§7). In practice, we observe that special state $(\phi, \varnothing)$ is only reached in situations where the ISS's resources (workers and models) cannot provide sufficient throughput to meet the query load. In such a scenario, the resource manager should provision more workers.

In summary, the state space of the worker MDP is:

$$S = \{(n, T_j) | 0 \leq n \leq N_w, T_j \in \mathbf{T}_w\} \cup \{(\phi, \varnothing)\}$$

4.3 Actions

Let $a = (m, b)$ denote an action taken in state $s = (n, T_j)$ at a worker w. Recall that each worker w is modeled as a distinct MDP and decision epochs are per-worker. For efficient policy generation, we only consider an action a as valid in a state s

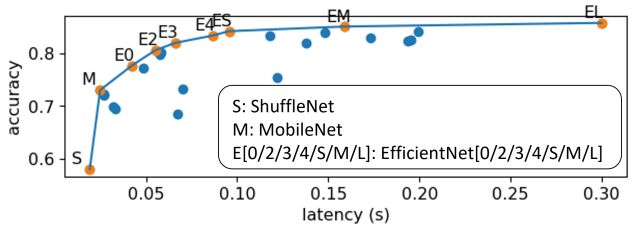

Figure 3. 95^{th} percentile inference latency vs. accuracy profile for image classification model set of 26 TorchVision ImageNet models. The inference latency includes the time to transfer input data to the worker, the data pre-processing time, and the inference time.

if it conforms to particular latency (§4.3.1), batch size (§4.3.2), and model (§4.3.3) constraints.

4.3.1 Latency Constraints. Per §4.2.1, we constrain the action space (i.e., the set of valid actions) at each state $s = (n, T_j)$ to actions $a = (m, b)$ which exhibit a inference latency $l_w(m, b)$ that meets the slack time, i.e., $l_w(m, b) \leq T_j$. If there is no action that can satisfy the slack time in s, then a latency SLO violation is unavoidable. Similar to existing work [7, 38, 57], RAMSIS assumes queries are better served late than never. Thus, in such a state, the only valid action is $(m_{w_{min}}, n)$, where $m_{w_{min}}$ is the lowest latency model available on w. RAMSIS can be re-formulated in a straightforward manner to drop queries whose deadlines cannot be satisfied [15, 43] via changes to the transition probabilities in §4.4.2.

4.3.2 Batch Size Constraints. We consider two batching strategies for RAMSIS: variable batching and maximal batching. By default, RAMSIS uses the latter.

With *variable batching*, the action space for state $s = (n, T_j)$, where $n > 0$, includes all actions $a = (m, b)$, such that $1 \leq b \leq n$. That is, b can be any non-zero integer batch size up to n. Empirically, we observe that variable batching MS policies select the maximum batch size $b = n$ in 80% of decisions. Thus, RAMSIS by default uses *maximal batching*, where the action space for state $s = (n, T_j)$, where $n > 0$, is $a = (m, n)$. That is, all queued queries are always collected into a single batch when taking an action in a state. We observe that policies which use maximal batching offer equivalent performance in practice to those which use variable batching (§D) while requiring significantly less time for policy generation, as shown in Table 2.

4.3.3 Model Constraints. To further reduce the action space for RAMSIS MDPs, we prune from consideration models that are not on the Pareto Front of accuracy and latency. Fig. 3 plots accuracy versus latency of 26 ImageNet models (§7). Of the 26 models, 17 are not on the Pareto Front and would be pruned, leaving 9 to be involved in valid actions.

4.3.4 Action on Empty Queue. The state $s = (0, T_j)$ represents an empty worker queue. Since the worker queue is empty, $T_j \in \mathbf{T}_w$ is left unconstrained. The action space in an empty queue state consists of a single special *arrival action* $\hat{a}$, which indicates that the worker will idle until the next query arrives at its queue.

4.4 Transition Probabilities

Given the (1) states (§4.2) and (2) actions (§4.3) for a worker w's MDP, together with an (3) arrival distribution for the central queue and (4) a query load balancing strategy (§3.1.1), we can precisely define worker MDP transition probabilities.

Recall that transition probability $P_a(s, s')$ defines the probability that taking action $a \in A$ in state $s \in S$ results in a transition to state $s' \in S$ in the next decision epoch (§4). Given $s = (n, T_j)$ and $s' = (n', T_{j'})$, transition $s \xrightarrow{a} s'$, or $(n, T_j) \xrightarrow{(m,b)} (n', T_{j'})$, implies:

(I) $n' - (n - b)$ queries arrived at the worker between s and s', by the definition of states and actions, and
(II) $T_{j'}$ is the slack time of the query with the earliest deadline in s', by the definition of states.

Thus, $P_a(s, s')$ is the joint probability of (I) and (II). For brevity, we focus our discussion on the maximal batching strategy (§4.3) where $b = n$, although the derivation of transition probabilities with variable batching (i.e., $b < n$) follows similar reasoning. We organize our discussion of transition probabilities into distinct cases conditioned on the number of queued queries in current state n and future state n'.

4.4.1 Case 1: $n = 0$ and $n' = 1$. The current state $s = (0, T_j)$ represents an *empty queue state*. Recall that the only applicable action is the arrival action $\hat{a}$, and $(0, T_j) \xrightarrow{\hat{a}} (1, SLO)$ (§4.3.4). That is, taking the arrival action $\hat{a}$ in an empty queue state yields a next state, which features one queued query whose slack time is SLO. Hence,

$$\mathrm{P}[s' = (1, SLO) | s = (0, T_j), a = \hat{a}] = 1 \tag{1}$$

4.4.2 Case 2: $0 < n \leq N_w$ and $0 \leq n' \leq N_w$. In this case, $s = (\phi, \varnothing)$, since $\phi = N_w$ (§4.2.3), or $s = (n, T_j)$, for $T_j \in \mathbf{T}_w$. The only valid actions are $a = (m, n)$, due to maximal batching, (§4.3.2). Note that special state $s = (\phi, \varnothing)$ exhibits equivalent transition probabilities to $s = (N_w, 0)$ given our assumptions outlined in §4.2.3.

Action $a = (m, n)$ taken in current state s defines the time elapsed $l_w(m, n)$ before next state s', which impacts the number of queued queries n' and slack $T_{j'}$ in s'. Further, considering K workers, the number of arrivals to worker w between s and s' depends on the number of arrivals to the central queue and the round-robin load balancer. Thus, we derive transition probabilities of the worker MDP in this case in terms of the arrival distribution at the central queue.

Recall that the arrival distribution $PF(k, T)$ supplied to RAMSIS as input expresses the probability of k arrivals at

the central queue during a time interval of length T (§3.1.1). RAMSIS assumes the arrival process at the central queue exhibits the *independent and stationary increments* property [41]. Thus, the number of arrivals for any pair of non-overlapping time intervals are independent. For example, given non-overlapping time intervals A and B, the joint probability of k_A arrivals during interval A of time length T_A and k_B arrivals during interval B of time length T_B is $PF(k_A, T_A) * PF(k_B, T_B)$. Note that independent and stationary increments is a property of commonly observed arrival processes (e.g., Poisson, Gamma). It holds for a general class of processes called Levy processes [41].

Recall that $P_a(s, s')$ is the joint probability of conditions (I) and (II) stated at the start of §4.4. With round-robin load balancing among K workers, a worker receives every K^{th} arrival to the central queue. Thus, the probability of $n' - (n - n) = n'$ arrivals at the worker queue between s and s' (condition (I) with maximal batching) is a function of the number of arrivals to the central queue prior to s, the time elapsed between s and s', and the round-robin load balancer. Further, given $n' > 0$, and $b = n$ (due to maximal batching), the first query arrival at worker w between s and s' exhibits the earliest deadline in next state $s' = (n', T_{j'})$, and defines slack $T_{j'}$. We define time points $t_{\alpha'}$ and $t_{\beta'}$ to encode time interval $(t_{\alpha'}, t_{\beta'})$ during which the query corresponding to $T_{j'}$ arrives (condition (II) corresponds to the probability of this "first" query arriving during $(t_{\alpha'}, t_{\beta'})$). Thus, we compute the joint probability of conditions (I) and (II) by counting query arrivals during four non-overlapping time intervals: A, B, C, and D, illustrated in Fig. 4. Through applying the stationary and independent increment property, we express the transition probability of a worker w's MDP between state $s = (n, T_j)$ and state $s' = (n', T_{j'})$ given action $a = (m, n)$ as:

$$\mathrm{P}[s' = (n', T_{j'}) | s = (n, T_j), a = (m, n)] = \frac{\sum_{k_A,k_B,k_C,k_D} PF(k_A, T_A) * PF(k_B, T_B) * PF(k_C, T_C) * PF(k_D, T_D)}{\sum_{k_A} PF(k_A, T_A)} \quad (2)$$

Above, k_A, k_B, k_C, k_D denote the number of arrivals to the central queue and T_A, T_B, T_C, T_D are time lengths, each corresponding intervals A, B, C, and D. We detail each interval in the following paragraphs.

Interval A. Interval A represents the time elapsed between t_α, the arrival time of the query with the earliest deadline in state $s = (n, T_j)$, and t_s, the time at which action a is taken in state s. T_A is the amount of time that the query with the earliest deadline in s spends in the worker queue. Thus, $T_A + T_j = SLO$, and consequently,

$$T_A = SLO - T_j$$

Since $s = (n, T_j)$, and one query arrives at t_α, $n - 1$ queries arrive at worker w during interval A. Since RAMSIS uses a round-robin load balancer with K workers, worker w gets every K^{th} query that arrives at the central queue. For $n - 1$ queries to arrive at worker w, the number of arrivals to the central queue k_A must be at least $(n-1)K$ and at most $nK-1$. That is,

$$k_A \in [(n-1)K, nK - 1]$$

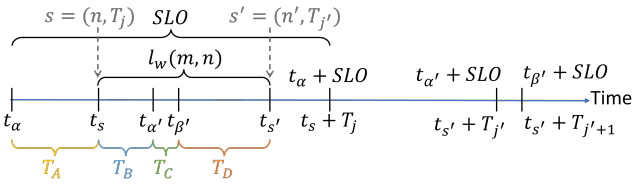

Figure 4. Four non-overlapping time intervals for transition probability analysis when $b = n$.

Interval B. Interval B represents the time elapsed between t_s, when action $a = (m, n)$ is taken in state $s = (n, T_j)$, and $t_{\alpha'}$, when the next query arrives. Note that $t_{s'} - t_{\alpha'}$ defines how long the query corresponding to $T_{j'}$, in next state $s' = (n', T_{j'})$, is queued at worker w before being serviced. Thus, $t_{s'} - t_{\alpha'} + T_{j'} = SLO$. Moreover, $T_B + t_{s'} - t_{\alpha'} = l_w(m, n)$, where $l_w(m, n)$ is the time elapsed between t_s and $t_{s'}$ from taking action $a = (m, n)$ defined by inference latency of model m with batch size n on worker w. Consequently,

$$T_B = l_w(m, n) + T_{j'} - SLO$$

Since the first arrival at the worker following t_s must take place at or after $t_{\alpha'}$, zero queries arrive at worker w during interval B. Since worker w is assigned a query for every K^{th} central queue arrival with round-robin, $k_A\%K$ denotes the number of central queue arrivals during interval A that worker w does *not* receive after worker w's last arrival. Then, after t_s (interval A), worker w receives its first next query from $K - (k_A\%K)$ central queue arrivals. Thus, for zero queries to arrive at worker w during interval B, the number of queries that arrive at the central queue during interval B is at least zero and at most $K - (k_A\%K) - 1$. That is,

$$k_B \in [0, K - (k_A\%K) - 1]$$

When solving the MDP, which considers all current-next state pairs, it is possible that $T_B = l_w(m, n) + T_{j'} - SLO < 0$. This is true for the case where the transition from s to s' with action $a = (m, n)$ where the sum of the inference latency $l_w(m, n)$ and the slack time $T_{j'}$ in state s' is less than SLO. Here, we set $T_B = 0$. Intuitively, this is sound since exactly zero arrivals can occur during a time length less than zero.

Interval C. Interval C, of length T_C, represents the time interval during which the query corresponding to slack time $T_{j'}$ arrived at worker w's queue. From Fig. 4, observe that $T_D + T_{j'+1} = SLO$ (note $T_{j'+1} \in \mathbf{T}_w$ as described in §4.2) and $T_B + T_C + T_D = l_w(m, n)$. Consequently,

$$T_C = l_w(m, n) + T_{j'+1} - SLO - T_B$$

If $T_C = l_w(m, n) + T_{j'+1} - SLO - T_B < 0$, we set $T_C = 0$ following the same reasoning as for interval B.

We split the derivation of the number of arrivals to the central queue during interval C into two cases: $n' = 0$ and $0 < n' \leq N_w$. In both cases, the number of arrivals to worker w depends on the number of central queue arrivals during previous intervals A and B due to round-robin scheduling.

For $n' = 0$, the number of queries to the worker during interval C must be zero. After $t_{\alpha'}$ (interval B), worker w receives its first next query from $K - (k_A \% K) - k_B$ central queue arrivals (recall zero queries arrive at worker w during interval B). Thus, the number of queries k_C that arrive at the central queue during interval C is at least zero and at most $K - (k_A \% K) - k_B - 1$. That is, when $n' = 0$,

$$k_C \in [0, K - (k_A \% K) - k_B - 1]$$

For $0 < n' \leq N_w$, the number of arrivals to worker w during interval C must be at least one and at most n'. For at least one query to arrive at worker w during interval C, the number of central queue arrivals during interval C must be at least $K - (k_A \% K) - k_B$. For at most n' arrivals to worker w during interval C, k_C must be at most $(n'+1)K - (k_A \% K) - k_B - 1$. Thus, when $0 < n' \leq N_w$,

$$k_C \in [K - (k_A \% K) - k_B, (n'+1)K - (k_A \% K) - k_B - 1]$$

Interval D. Interval D represents the time elapsed between the end of the interval during which the query corresponding to $T_{j'}$ arrives and the time at which the MDP transitions to future state $t_{s'}$. From Fig. 4, observe that

$$T_D = l_w(m, n) - T_C - T_B$$

Recall that n' queries must arrive at the worker between t_s and $t_{s'}$. Then, the total number of central queue arrivals during intervals C and D must be at least $n'K - (k_A \% K) - k_B$ and at most $(n'+1)K - (k_A \% K) - k_B - 1$ with respect to round-robin scheduling with K workers. Further, if i queries arrive at the worker during interval C, then $n' - i$ queries must arrive at the worker during interval D. Thus, the number of central queue arrivals along interval D:

$$k_D \in [max(0, n'K - (k_A \% K) - k_B - k_C), \\ (n'+1)K - (k_A \% K) - k_B - 1 - k_C]$$

4.4.3 Case 3: $n \neq 0$ and $n' > N_w$.

This case represents the transition to the state where the number of arrivals accumulate beyond the maximum worker queue size (i.e., $n' > N_w$). Recall that $n' > N_w$ is represented as a special discrete state $s' = (\phi, \varnothing)$ (§4.2.3). Given $s \in S$, we calculate the probability of transitioning to state $s' = (\phi, \varnothing)$ in terms of *not* transitioning to the state using equations 1 and 2 as follows:

$$\mathrm{P}[s' = (\phi, \varnothing) | s = (n, T_j), a = (m, b)] = \\ 1 - \sum_{s' \in S \setminus \{(\phi, \varnothing)\}} \mathrm{P}[s' | s = (n, T_j), a = (m, b)] \tag{3}$$

Together, equations 1, 2, and 3 fully specify the transition probabilities $P_a(s, s')$.

5 RAMSIS Scalability and Guarantees

In this section, we detail RAMSIS's accuracy and latency guarantees (§5.1) and policy generation complexity (§5.2).

5.1 Latency and Accuracy Guarantees

Using a worker w's MDP (§4), RAMSIS constructs an MS policy π_w, which offers probabilistic guarantees on accuracy and latency (akin to Cocktail [16]). That is, RAMSIS can compute latency violation rate and accuracy summary statistics (e.g., expectation, median, 99th percentile) over the entire state space of w's MDP; latency violation rate refers to the fraction of all served queries which do not meet their deadlines.

In particular, ISSs can benefit from RAMSIS's support for computing the *expectation of inference accuracy* and the *expectation of latency SLO violation rate* of a policy π_w. That is, either users or the ISS resource manager can use the expectation of inference accuracy and latency violation rate provided by RAMSIS to direct resource scaling decisions, e.g., via an offline search for resource configurations that achieve sufficient accuracy and latency SLO violation rate.

The expected latency SLO violation rate for π_w can be expressed as:

$$E_{\pi_w}[SLOViolation] \leq \sum_s P_{\pi_w}(s) * (1 - SLOSatisfied(s, \pi_w[s]))$$

where $P_{\pi_w}(s)$ is probability of being in state s for each state in worker w's MDP which is calculated via power iteration [40] from the transition probabilities (§4.4).

The expected accuracy for π_w can be expressed as:

$$E_{\pi_w}[Accuracy] \geq \sum_{s \in S^*} P_{\pi_w}(s) * Accuracy(\pi_w[s])$$

where $S^* = \{s | s \in S, SLOSatisfied(s, \pi_w[s]) = 1\}$.

Empirically (§7.3.1), we find that observed (online) average accuracy and latency violation rate closely follow their expectations as computed above. Notably, the expected accuracy provides a lower bound while the expected latency violation rate serves as an upper bound. We provide the following intuition for this finding:

(1) The slack time of a state T_j underestimates the real slack Δ (§4.2) and thus a *SLOSatisfied* may report false negatives but does not return false positives.
(2) Given state s and action $\pi_w[s]$, $SLOSatisfied(s, \pi_w[s])$ assumes all the queries served in the action are missed if the earliest deadline is not met.

5.2 Scalability

Recall RAMSIS uses value iteration (§4.1) for policy generation, which reasons about all possible transitions $P_a(s, s')$, i.e., for all a, s, and s'. Policy generation thus exhibits a runtime complexity of $O(|S|^2|A|)$, where $|S|$ and $|A|$ denote the state and action space size (§4), respectively [8].

Since the maximum worker queue size N_w need only be roughly large enough to handle the largest possible batch

size B_w (§4.2.3), $N_w = O(B_w)$. Since all relevant slack times $T_j \in \mathbf{T}_w$ can be represented by a discrete space of size $|\mathbf{T}_w| = O(|M_w| * B_w)$ (§4.2), the state space required to represent $s = (n, T_j)$, including special states §4.2.3, is $|S| = O(N_w * |\mathbf{T}_w|) = O(|M_w| * B_w^2)$. The action space size with maximal batching (§4.3.2) is $O(|M_w|)$ which counts all possible model selections. Thus, the time complexity to generate an MS policy is $O(|M_w|^3 * B_w^4)$.

RAMSIS's offline policy generation is polynomial in the number of models and maximum batch size. The total number of pre-loaded models per worker $|M_w|$ is often small (i.e., much less than 100 [15, 28, 32, 38, 57]) since there are few models on the Pareto Front (e.g., Fig. 3) [4, 37, 47, 56], and available memory capacity limits the number of models which can be simultaneously loaded on a worker [15, 28, 32]. Further, maximum batch sizes B_w are often small (1-64) due to the prevalence of low latency SLOs [5, 17, 25, 29] (e.g., we observed $B_w = 29$ in our experiments). If the policy generation runtime with MD is too slow due to large $|M_w|$ or B_w, FLD (§4.2.2) can be used to reduce runtime (Table 2).

6 RAMSIS Implementation

We evaluate RAMSIS in simulation and as a prototype implementation.[2] We design RAMSIS as a client-server architecture, where the *central controller* is a single virtual machine (VM) that receives all client queries and dispatches them to worker VMs.

Simulation Framework. The RAMSIS simulator is implemented in 1K lines of Python. Given a trace of arrival times [1], it records MS&S decisions and tracks the central queue state (queued queries/deadlines) and worker statuses (busy or available). We use the model inference latency profiles collected on the target hardware platform to determine how long a worker is busy as a result of an MS decision.

Prototype Implementation. Our RAMSIS prototype consists of about 3K lines of Python code using TorchServe [35] for executing inference. The central controller VM runs a *workload generator* process in addition to a *load balancer* process and per-worker *model selector* processes. The workload generator produces a stream of query arrivals according to a query load trace (§7) under a stochastic inter-arrival pattern (e.g., Poisson). The central queue is implemented as a first-in-first-out data structure in shared memory that stores the wall clock arrival time of each query and a reference to its input data (e.g., an image or text). The load balancer distributes queries in the central queue to worker queues; one worker queue is associated with each worker VM. Worker model selectors dispatch queries from their respective queues to their associated worker VMs. Worker VMs run a TorchServe server which exposes an HTTP Request API for their worker model selectors to send queries.

[2] https://github.com/dmmendo/RAMSIS

Load Monitor. Both RAMSIS and our baselines (§7) use the same load monitor implementation (§3), which tracks query load via a moving average over a window of 500 milliseconds [38, 57]. Other approaches to track query load (e.g., neural network load prediction [16, 54]) can be used.

Policy Generation. Unless otherwise stated, RAMSIS policy generation uses maximal batching (§4.3) and fixed-interval time discretization (§4.2) with $D = 100$ throughout our evaluation. We set max queue size $N_w = 32$ in policy generation since we observed the maximum batch size for the largest evaluated SLO was 29 (§4.2.3).

Query Load Adaptation. RAMSIS pre-computes model selection policies for a range of arrival distributions; all feature Poisson inter-arrivals but vary according to query load (§3.1.3). We generate (pre-compute) policies for differing query load such that the largest difference between the expected accuracies (§5.1) among all pairs of adjacent policies (when ordered by increasing load) is below a threshold—1% in our experiments.

7 Evaluation

Our main evaluation answers the following questions, and §C, §D, §F contain additional results. How does MS&S with RAMSIS compare to state-of-art approaches [32, 57] on a production trace [1] (§7.1)? To what degree does accounting for the query inter-arrival pattern improve achieved query accuracy (§7.2)? Does the observed accuracy and latency violation rate on our RAMSIS implementation align with simulation observations and theoretical expectation (§7.3.1)? How does the available set of ML models impact RAMSIS's accuracy (§7.3.2)?

Hardware Setup. We conduct our experiments on Google Cloud Platform (GCP) VMs [14]. The central controller VM is a GCP n2 instance, equipped with 128 Intel Haswell CPUs and 512GB of DRAM. Workers VMs are GCP n1 instances, equipped with 4 Intel Haswell CPUs and 16GB of DRAM. By design, compute, memory, and network bandwidth are not a bottleneck to meeting latency SLOs.

Inference Tasks. We evaluate RAMSIS on two tasks: *image classification* and *text classification*. For image classification, RAMSIS has access to 26 models from Torch Vision [30] (Fig. 3): 11 EfficientNets [47], 5 ResNets [21], 2 ResNexts [53], GoogleNet [45], 2 MobileNets [22], Inception [46], and 4 ShuffleNets [58]. For sequence classification, RAMSIS is supplied with 5 Bert models [11] from huggingface [52] (Fig. 9): tiny, mini, small, medium, and base.

For each task, all models are pre-loaded on each worker VM. Worker homogeneity is not a fundamental requirement for RAMSIS since policies are generated per worker (§4).

We evaluate each task under three representative latency SLOs: 150 ms, 300 ms, and 500 ms for image classification, and 100 ms, 200 ms, and 300 ms for text classification. The middle SLO is set as the latency of the highest-latency model

rounded up to the nearest hundred milliseconds. The lowest SLO is half the middle SLO. The highest SLO is 1.5× the latency of the highest-latency model rounded up to the nearest hundred milliseconds. Latency SLOs for ML applications are typically in the range of a few hundred milliseconds [17, 38].

Workloads. We evaluate RAMSIS and existing approaches using a 24-hour production inference query trace from Twitter [1], which exhibits both diurnal patterns and unexpected spikes in query load [38, 54]. We scale the Twitter trace down to five minutes (from one day) for our experiments, as is done in prior work [38]. The resulting trace defines query loads over time ranging from 1,617 to 3,905 queries per second (QPS). Since the Twitter trace logs query load over fixed time intervals rather than explicit query arrival times, we sample arrival times of each query via a Poisson process under the aforementioned range of query loads, resulting in 554,395 total queries.

In addition to the production trace, we evaluate RAMSIS and baselines on constant query load (specified per experiment) for 30 seconds under Poisson arrivals to measure the impact of the inter-arrival pattern on MS&S.

Baseline MS&S Policies. We compare RAMSIS to two MS&S approaches, *Jellyfish+* and *ModelSwitching* [57]. Jellyfish+ extends Jellyfish [32] with support for balancing query load from an inference application across multiple workers. RAMSIS, Jellyfish+, and ModelSwitching all collect model profiles and generate MS policies offline. Policies are generated once per query load (and worker type for RAMSIS), resulting in negligible overhead online for MS&S decisions. Like RAMSIS, Jellyfish and ModelSwitching aim to maximize inference accuracy within a comparable ISS architecture (Table 1). Unlike RAMSIS, they conservatively do not explicitly account for the query inter-arrival pattern to support this goal. We describe both baselines in more detail below. Note that since both baselines are load-granular MS&S approaches (§2), model selections switch only on query load changes. For fair comparison, we evaluate RAMSIS and baselines in the same implementation framework (§6) and they do not drop queries when facing latency SLO violations.

Jellyfish+. JellyFish [32] assumes a single worker per SLO, and JellyFish+ extends JellyFish with multiple workers per SLO. Given some query load, Jellyfish+ selects the most accurate model such that the model's average throughput is greater than the anticipated query load, and the model's *inference latency* is less than half the latency SLO. The inference latency is constrained to half the latency SLO to avoid latency SLO violations in anticipation of the worst-case (maximum) wait times in the central queue [32, 43]. Jellyfish+ estimates a model's throughput as the sum of the average profiled throughput among each worker. Workers eagerly grab and service queries from the central queue in batches up to a maximum batch size set according to *adaptive batching* [7].

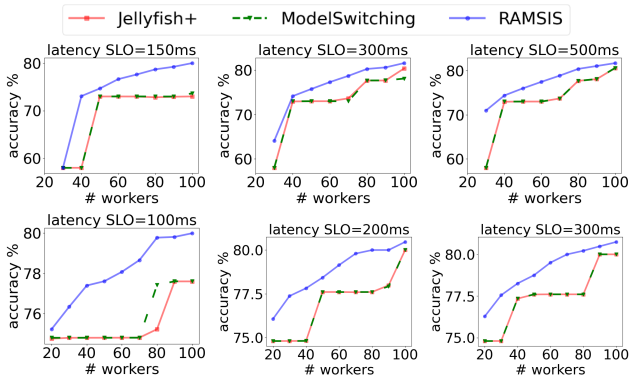

Figure 5. Existing MS&S approaches vs. RAMSIS prototype implementation, given production Twitter trace on the image classification (top) and text classification (bottom) tasks. RAMSIS achieves higher accuracy with same resources or the same accuracy with fewer resources.

ModelSwitching. ModelSwitching [57] measures the *response latency* of each model under anticipated query loads offline. Given some query load, it selects the most accurate model such that the model's 99^{th} percentile *response latency* is less than the latency SLO under the anticipated query load. ModelSwitching and Jellyfish+ employ the same implicit load balancing strategy with adaptive batching. The response latency of each model is collected in an offline profiling step over the relevant a range of query load (i.e., 400 to 4000 QPS in increments of 100 QPS) on all evaluated resource configurations (i.e., 20 to 100 workers).

Performance Metrics. We compare RAMSIS, Jellyfish+, and ModelSwitching for each classification task and workload (i.e., query trace) with respect to their observed *Latency SLO Violation Rate* and *Accuracy Per Satisfied Query*, as is done in prior work [16, 32, 57]. Latency SLO Violation Rate is the fraction of all serviced queries whose latency deadline is missed. Accuracy Per Satisfied Query is the average profiled accuracy over all satisfied queries given the model selection decision of each query.

7.1 Evaluating RAMSIS on a Production Trace

We now compare MS&S with our prototype RAMSIS implementation to our baselines given the Twitter trace. We consider each application separately (image and text classification) and vary the number of workers from 20 to 100 in increments of 10.

Fig. 5 plots accuracy versus number of workers for this experiment. Only data points which correspond to a latency SLO violation rate of less than 5% are included. RAMSIS and the baselines are susceptible to inference latency variance where the inference latency is occasionally unexpectedly long (e.g., beyond profiled 95th percentile). Thus, SLO violation rates vary slightly between runs for both RAMSIS and

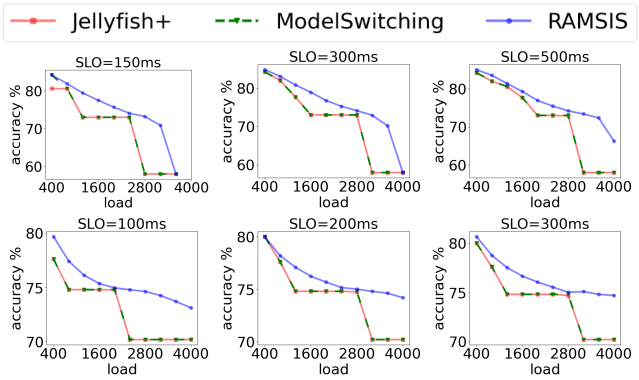

Figure 6. Existing MS&S schemes vs. RAMSIS prototype implementation, queried with constant load on the image classification (top) and text classification (bottom) task. RAMSIS achieves equal or higher accuracy given same constant query load and resources.

the baselines. However, when query load is satisfiable, RAMSIS and the baselines exhibit low SLO violation rates. The average latency violation rates for RAMSIS, ModelSwitching, and Jellyfish+ are 0.14%, 0.24%, and 0.21%, respectively (see §F for more details). Given the same resources and latency SLO, compared to ModelSwitching and Jellyfish+, RAMSIS exhibits a 0.04% and 0.07% lower SLO violation rate for image classification while achieving 0.15% and 0.06% lower SLO violation rate for text classification, respectively, and *always* achieves higher accuracy. For image classification, RAMSIS achieves up to 15.09% and 15.08% (on average 4.43% and 4.35%) higher accuracy than ModelSwitching and Jellyfish+, respectively. For text classification, RAMSIS achieves up to 3.85% and 4.55% (on average 1.93% and 2.01%) higher accuracy than ModelSwitching and Jellyfish+, respectively. RAMSIS's accuracy benefits are a direct result of its fine-grained, inter-arrival pattern-aware decisions, where models are selected *per batch* of queries. In contrast, Jellyfish+ and ModelSwitching adapt MS decisions upon changes in query load only.

The accuracy improvements of RAMSIS enable reduction in costs through achieving the same accuracy with fewer resources than the baselines. Notably, to meet the same accuracy as ModelSwitching and Jellyfish+ for image classification, RAMSIS requires as low as 50.00% and 42.86% (on average 20.01% and 17.53%) fewer resources, respectively. To meet the same accuracy as ModelSwitching and Jellyfish+ for text classification, RAMSIS uses as low as 44.44% and 75.00% (on average 25.31% and 31.25%) fewer resources, respectively.

Insight: RAMSIS significantly improves ISS accuracy compared to our baselines for a real-world trace and enables considerable cost reduction while providing the same accuracy.

7.2 Evaluating RAMSIS with Constant Query Load

Next, we demonstrate how RAMSIS can leverage inter-arrival patterns to maximize accuracy even at constant query load and constant resources (i.e., workers). We conduct the same experiment with our prototype RAMSIS implementation at several representative 30-second constant load traces, ranging from 400 QPS to 4000 QPS in increments of 400. We select the number of workers so that at high load (3600-4000 QPS) only the lowest latency model exhibits sufficient throughput to meet the load with both baselines. The result is 60 and 20 workers for image and text classification, respectively. For these experiments, we assume the load monitor perfectly predicts the query load to focus our evaluation on comparing the best possible performance of all evaluated MS&S approaches.

Fig. 6 plots accuracy versus query load for this experiment. Again, only data points with a latency SLO violation rate of less than 5% are included. Given same resources, query load, and latency SLO, RAMSIS consistently achieves equal or higher accuracy with a comparable latency violation rate compared to the baselines. The average latency violation rates for RAMSIS, ModelSwitching, and Jellyfish+ are 0.30%, 0.23%, and 0.39%, respectively (see §F for more details). For image classification, RAMSIS compared to ModelSwitching and Jellyfish+ achieves up to 15.42% (on average 4.82% and 4.95%) higher accuracy with a 0.13% higher and 0.15% lower average violation rate, respectively. For text classification, RAMSIS achieves up to 4.88% (on average 2.25% and 2.26%) higher accuracy with a 0.02% and 0.01% lower average latency violation rate compared to ModelSwitching and Jellyfish+, respectively.

At the extreme ends of the query load range (low and high), RAMSIS performs similarly to the baselines. This is because at high load, close to the largest satisfiable load, the only MS decision which can meet the query load and stochastic inter-arrival pattern is the lowest latency model. At low load, queries arrivals are too infrequent for the inter-arrival pattern to significantly impact performance.

Insight: Accounting for the inter-arrival pattern enables RAMSIS to achieve consistent and significant accuracy benefits over load-granular approaches across the range of satisfiable query loads with constant resources.

7.3 Sensitivity Experiments

7.3.1 RAMSIS's Fidelity. We now evaluate the ability of our theoretical expectation calculations (§5.1) and simulation infrastructure to emulate the RAMSIS implementation (§6). We consider these three RAMSIS variants—expectation, simulation, and implementation—on the image classification task for 30 second constant query load traces with 40, 60, and 80 workers. Fig. 7 plots accuracy (top) and latency SLO violation rates (bottom) for each variant.

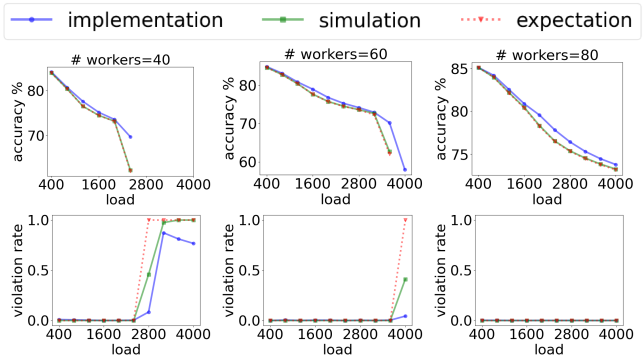

Figure 7. Comparison of RAMSIS's achieved accuracy and latency SLO violation rate in expectation, in the simulation framework, and in the prototype implementation. RAMSIS's performance in simulation and in the prototype implementation closely follow the expectation.

The simulation framework uses the same scheduling code as the implementation. The only discrepancy between simulation and implementation is that the simulation assumes inference latency is deterministically the 95^{th} percentile of the model profile (i.e., Fig. 3 and Fig. 9). In general, RAMSIS achieves higher accuracy and a lower SLO violation rate in implementation than in simulation. This is because our RAMSIS simulation does not capture inference latency variance. In our real implementation, inference latency may be shorter than its profile, enabling RAMSIS MS&S policies to select higher accuracy models for subsequent batches. We observe a standard deviation in inference latency of around 10 milliseconds for all models during latency profiling.

Fig. 7 also shows that RAMSIS simulation closely follows expected accuracy. The same is true for SLO violation rate, except at high query loads which are close to the largest satisfiable query load given the resources (i.e., the peak capacity). This is because at satisfiable query load (below peak capacity), the number of latency violations is low both in expectation and simulation. However, under high query load (beyond peak capacity), latency violations are unavoidable. In the latter case, RAMSIS expectation overestimates latency violation rate. This is because with a load beyond the peak capacity, the probability of reaching the special full queue state (§4.2.3) becomes high. RAMSIS assumes latency violations in the state are unavoidable, thereby causing the expectation to overestimate the violation rate at high query load.

7.3.2 Scaling to Many Models. To evaluate sensitivity to the number of available models for RAMSIS and our baselines, we compare performance (accuracy and latency violation rate) for image classification assuming *low* and *high* model counts. For the low model count scenario, all MS&S approaches use $M = 9$ models taken from the accuracy-latency Pareto Front spanned by the original image classification

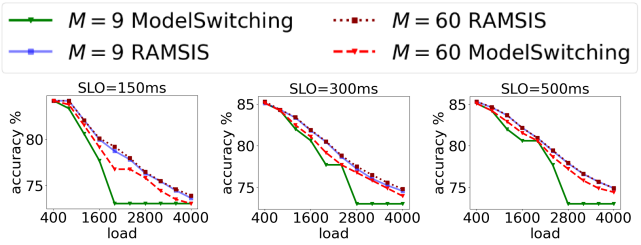

Figure 8. ModelSwitching and RAMSIS with differing model set sizes where the larger model set ($M = 60$) is constructed based on the Pareto Front of the image classification task (Fig. 3) consisting of $M = 9$ models.

model set (Fig. 3). For the high model count scenario, we construct a synthetic set of $M = 60$ models. We use linear interpolation on the Pareto front of the original 9 models to instantiate synthetic models in 0.5% accuracy increments. The set of models in the high model count scenario is a strict superset of those in the low model count scenario. Our experiments in this section consider 30 second constant query load traces with 100 workers. For brevity, we include results for RAMSIS and ModelSwitching only, as ModelSwitching is the best performing baseline in this experiment.

Fig. 8 plots accuracy versus query load for this experiment. Only data points which correspond to a latency SLO violation rate of less than 5% are included. Notably RAMSIS sees negligible performance improvement when equipped with 60 models versus 9.

Insight: RAMSIS emulates a large model set through fine-grained MS&S decisions that maximize accuracy.

Alternately, ModelSwitching exhibits significantly improved accuracy in the high model count scenario—still lower than RAMSIS, however. This is because ModelSwitching must select the same model for a constant query load. To avoid latency SLO violations, ModelSwitching is forced to consistently select a model which exhibits an inference latency that is (sometimes significantly) lower than required.

Insight: RAMSIS MS&S policies, which exploit the query inter-arrival pattern, offer higher accuracy improvements for existing ISSs compared to increasing the model count for load-granular approaches.

8 Related Work

MS&S for Inference Latency Variance. Some MS&S approaches, like MDInference [33] and ALERT [48], target hardware resources which exhibit high inference latency variance, e.g., due to network latency, co-location interference, or dynamic frequency scaling [27, 33, 48]. These systems greedily select the most accurate model given the current arrived queries and their deadlines, which is not sufficient to avoid latency SLO violations under varying query load and stochastic inter-arrival patterns. In contrast, we focus

on model selection for hardware resources which exhibit predictable latency [15, 28, 43].

ML Model Ensembling. MS&S approaches specialized for ML model ensembling have been proposed [7, 16, 50]. Here, multiple models are invoked per query and the output of each model is aggregated via some weight scheme (e.g., most popular class). Ensembling has only been shown to be cost-effective with preemptible instances [16]. Further, these model selection techniques do not account for stochastic inter-arrival patterns—our focus.

Resource Management. Alpaserve [28] partitions models across hardware resources to optimize operator parallel strategies. MArk [54] and Inferline [6] autoscale workers in order to save cost while meeting latency SLOs. INFaaS [38] proposes model autoscaling with heterogeneous resources to minimize cost. REEF [19] demonstrates a GPU kernel preemption strategy to share GPUs across both latency-critical and best-effort tasks. RAMSIS is an MS&S framework that maximizes accuracy constrained to latency SLOs given a resource allocation from the resource manager. RAMSIS can be combined with existing resource management strategies.

9 Conclusion

Existing MS&S approaches are overly-conservative in the presence of stochastic inter-arrival patterns, missing opportunities to improve inference accuracy by sending queries to higher accuracy/latency models. Our solution is RAMSIS, an MS&S framework which efficiently pre-computes MS&S policies offline that explicitly account for the query inter-arrival pattern to maximize query accuracy given a latency SLO. We evaluate RAMSIS against state-of-the-art MS&S approaches and show that it achieves the same accuracy with as low as 50.00% and 75.00% (on average 18.77% and 28.28%) fewer resources on an image classification task and a text classification task, respectively.

Acknowledgments

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 16KIS1138). We thank the anonymous reviewers and our shepherd Shivaram Venkataraman for their valuable feedback.

References

[1] ArchiveTeam 2018. Twitter Streaming Traces. https://archive.org/details/archiveteam-twitter-stream-2018-04.

[2] AWS. 2019. Deliver high performance ML inference with AWS Inferentia. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deliver_high_performance_ML_inference_with_AWS_Inferentia_CMP324-R1.pdf.

[3] Pamela Badian-Pessot, Mark Lewis, and Douglas Down. 2019. OPTIMAL CONTROL POLICIES FOR AN M/M/1 QUEUE WITH A REMOVABLE SERVER AND DYNAMIC SERVICE RATES. *Probability in the Engineering and Informational Sciences* 35 (07 2019), 1–21. https://doi.org/10.1017/S0269964819000299

[4] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. 2018. Benchmark Analysis of Representative Deep Neural Network Architectures. *IEEE Access* 6 (2018), 64270–64277. https://doi.org/10.1109/ACCESS.2018.2877890

[5] Marshall Choy. 2021. Accelerating the Modern Machine Learning Workhorse: Recommendation Inference. https://sambanova.ai/blog/accelerating-the-modern-ml-workhorse-recommendation-inference/

[6] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285

[7] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 613–627.

[8] Tapas K. Das, Abhijit Gosavi, Sridhar Mahadevan, and Nicholas Marchalleck. 1999. Solving Semi-Markov Decision Problems Using Average Reward Reinforcement Learning. *Management Science* 45, 4 (1999), 560–574. http://www.jstor.org/stable/2634824

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[12] Matthew F. Dixon, Igor Halperin, and Paul Bilokon. 2020. *Machine Learning in Finance*. Springer International Publishing. https://doi.org/10.1007/978-3-030-41068-1

[13] Facebook Research. 2021. An implementation of a deep learning recommendation model (DLRM). https://github.com/facebookresearch/dlrm.

[14] Google. 2023. *Google Cloud Platform*. https://cloud.google.com/

[15] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 25, 20 pages.

[16] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2022. Cocktail: A Multidimensional Optimization for Model Serving in Cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1041–1057. https://www.usenix.org/conference/nsdi22/presentation/gunasekaran

[17] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. 2020. DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 982–995. https://doi.org/10.1109/ISCA45697.2020.00084

[18] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. 2007. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation* 64, 9 (2007), 1062–1081. https://doi.org/10.1016/j.peva.2007.06.012 Performance 2007.

[19] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[20] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 620–629. https://doi.org/10.1109/HPCA.2018.00059

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[22] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]

[23] Qiying Hu and Wuyi Yue. 2003. Optimal replacement of a system according toa semi-Markov decision process in a semi-Markov environment. *Optim. Methods Softw.* 18, 2 (2003), 181–196. https://doi.org/10.1080/1055678031000111803

[24] Fei Jiang, Yong Jiang, Hui Zhi, Yi Dong, Hao Li, Sufeng Ma, Yilong Wang, Qiang Dong, Haipeng Shen, and Yongjun Wang. 2017. Artificial intelligence in healthcare: past, present and future. *Stroke and Vascular Neurology* 2, 4 (2017), 230–243. https://doi.org/10.1136/svn-2017-000101 arXiv:https://svn.bmj.com/content/2/4/230.full.pdf

[25] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA52012.2021.00010

[26] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1586–1597. https://doi.org/10.14778/3137628.3137664

[27] Kevin Lee, Vijay Rao, and William Arnold. 2019. Accelerating Facebook's infrastructure with application-specific hardware. https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/.

[28] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. arXiv:2302.11665 [cs.LG]

[29] machynist and kippinitreal. 2020. How We Scaled Bert To Serve 1+ Billion Daily Requests on CPUs. https://blog.roblox.com/2020/05/scaled-bert-serve-1-billion-daily-requests-cpus/

[30] TorchVision maintainers and contributors. 2016. *TorchVision: PyTorch's Computer Vision library*.

[31] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Interference-Aware Scheduling for Inference Serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 80–88. https://doi.org/10.1145/3437984.3458837

[32] Vinod Nigade, Pablo Bauszat, Henri Bal, and Lin Wang. 2022. Jellyfish: Timely Inference Serving for Dynamic Edge Networks. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. 277–290. https://doi.org/10.1109/RTSS55097.2022.00032

[33] Samuel S. Ogden and Tian Guo. 2020. MDINFERENCE: Balancing Inference Accuracy and Latency for Mobile Applications. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*. 28–39. https://doi.org/10.1109/IC2E48712.2020.00010

[34] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[36] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., USA.

[37] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*.

[38] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.

[39] Sheldon M. Ross. 1983. *Introduction to Stochastic Dynamic Programming: Probability and Mathematical*. Academic Press, Inc., USA.

[40] Christopher De Sa, Bryan He, Ioannis Mitliagkas, Christopher Ré, and Peng Xu. 2017. Accelerated Stochastic Power Iteration. arXiv:1707.02670 [math.OC]

[41] Ken-iti Sato. 2001. *Basic Results on Lévy Processes*. Birkhäuser Boston, Boston, MA, 3–37. https://doi.org/10.1007/978-1-4612-0197-7_1

[42] Linn I. Sennott. 1998. *Stochastic Dynamic Programming and the Control of Queueing Systems*. Wiley-Interscience, USA.

[43] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[44] Shaler Stidham and Richard R. Weber. 1993. A survey of Markov decision models for control of networks of queues. *Queueing Systems* 13 (1993), 291–314.

[45] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. arXiv:1409.4842 [cs.CV]

[46] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. arXiv:1512.00567 [cs.CV]

[47] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6105–6114. https://proceedings.mlr.press/v97/tan19a.html

[48] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. 2020. ALERT: Accurate Learning for Energy and Timeliness. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 353–369. https://www.usenix.org/conference/atc20/presentation/wan

[49] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. arXiv:1804.07461 [cs.CL]

[50] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: Machine Learning as an Analytics Service System. *Proc. VLDB Endow.* 12, 2 (oct 2018), 128–140. https://doi.org/10.14778/3282495.3282499

[51] D. J. White. 1993. A Survey of Applications of Markov Decision Processes. *The Journal of the Operational Research Society* 44, 11 (1993), 1073–1096. http://www.jstor.org/stable/2583870

[52] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771 [cs.CL]

[53] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. arXiv:1611.05431 [cs.CV]

[54] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1049–1062.

[55] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[56] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 377–392. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang

[57] Jeff (Jun) Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'20)*. USENIX Association, USA, Article 5, 1 pages.

[58] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2017. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. arXiv:1707.01083 [cs.CV]

A Artifact Appendix

A.1 Abstract

This artifact evaluates the performance of RAMSIS, Jellyfish+ [32], and ModelSwitching [57] in terms of achieved inference accuracy and latency SLO violation rate in a simulation of stochastic query arrivals. Compared to the baselines, RAMSIS achieves higher accuracy while incurring similar or fewer latency SLO violations. In this evaluation, we first generate MS policies offline for each approach. Then, each approach is simulated across various settings of query load and number of workers. Both policy generation and simulation are implemented in Python.

A.2 Description & Requirements

We provide a docker image with all required software dependencies pre-installed. This section details the contents of the container and how to run it.

A.2.1 How to access. The artifact features a Docker image which contains all software dependencies and needed datasets. It can be accessed at this link:
https://doi.org/10.6084/m9.figshare.24223438

A.2.2 Hardware dependencies. Minimally, a four-core CPU with 16 GB RAM.

A.2.3 Software dependencies. Docker is required to run the artifact.

Required Dependencies (pre-installed in Docker image): Python3 and Python modules including NumPy, Pytorch, SciPy, tqdm, Numba, Scikit-learn, and tabulate.

A.2.4 Benchmarks.

- **Twitter trace.** We use a five minute Twitter trace [1] to simulate inference queries as described in §7. The trace is located in `twitter_trace/twitter_04_25_norm.txt`, a text file that lists the average queries per second (QPS) for ten-second intervals, ranging from 1,617 to 3,905 QPS.
- **Inference model profiles.** We evaluate RAMSIS and baselines on a text classification and image classification task. Each task consists of a unique set of inference models, as described in §7. We collect a profile for each model, which consists of its inference accuracy (ImageNet [9] for image task and GLUE-MNLI [49] for text task) and inference latency running on n1 GCP CPU instances. The latency profiles are located in `profiles/MODELNAME/BATCHSIZE.json` where each latency profile is a list of latencies for the model invoked 100 times. The accuracy profiles are located in `profiles/image_models.py` and `profiles/seq_models.py` which are dictionaries that map model name to its accuracy for the image and text models, respectively.

A.3 Set-up

This section provides step-by-step instructions to prepare the system environment for the artifact evaluation.

1. **Extract and Run the Docker container**

   ```
   docker load -i ramsis_ae.tar
   docker run -it ramsis_ae
   ```

2. **Verify set up**

   ```
   python3 RAMSIS_gen.py --worker 1 --load 1
   ```

   The message `script complete!` should be output to terminal within two minutes.

A.4 Evaluation workflow

You are now ready to run the evaluation.

A.4.1 Major Claims. In this artifact evaluation, we verify the main claim of the paper: RAMSIS achieves significantly higher accuracy than state-of-the-art approaches (i.e., ModelSwitching and Jellyfish+) given the same number of resources under the same query load while incurring similar latency SLO violations.

Note there is a discrepancy in measurements on achieved accuracy and latency SLO violation rate between this artifact evaluation and results in §7. This is because this artifact evaluation is conducted in simulation while the results reported in §7 are collected from a real implementation as explained in §7.3.1.

A.4.2 Experiments. We first generate all MS policies before deploying them on a production query trace and under constant query load.

So that this artifact evaluation can be completed within a few hours, the following steps are designed to reproduce a subset of the main results in §7.1 and §7.2 for the image classification task. However, the scripts we provide can be used to reproduce all main results in §7.1 and §7.2 via simulation and we detail how to do so in §A.5.

Policy Generation [5 human-minutes + 1 compute-hour]. To run the experiments, we first generate the policies for each technique. The following generates all required RAMSIS policies for the evaluation:

```
python3 RAMSIS_gen.py
```

As each RAMSIS policy is specialized to a query load and resource configuration, we generate a set of policies for query load ranging from 200 to 4,000 QPS in intervals of 200 and number of workers ranging from 60 to 80 in intervals of 10. For brevity of compute time, the policies for ModelSwitching are provided in the container and thus generation is not needed. We detail how the ModelSwitching policies can be generated in §A.5. Jellyfish+ only requires the model profiles to enforce its MS policies. Thus, there is not an explicit policy generation step. Each policy can be viewed in `policy_gen/METHOD_NUMWORKERS_SLO/LOAD.json`. Each file

contains a policy, which is a dictionary mapping states of the MDP to actions.

Experiment: Production Trace [5 human-minutes + 1 compute-hour]. Once RAMSIS policies have been generated, we can evaluate RAMSIS and baselines with a Twitter trace [1] across varying resource configurations on the image classification task.

1. **Simulation.** Simulate RAMSIS, ModelSwitching (MS), and Jellyfish+ (JF) on image classification task with 150 millisecond latency SLO while sweeping number of workers from 60 to 80 in intervals of 10.

   ```
   python3 run_sim.py --m RAMSIS --trace real
   python3 run_sim.py --m MS --trace real
   python3 run_sim.py --m JF --trace real
   ```

   This step measures the achieved accuracy and latency SLO violation rate of each technique and stores the results in `results/TASK_METHOD_TRACE_SLO_*.json`.
2. **Plot Results.** Plots the accuracy and latency SLO violation rate across varying number of workers:

   ```
   python3 plot.py --trace real
   ```

   The plots are printed to terminal and saved as `image_real_accuracy.png` and `image_real_violation.png`. The following output should be printed to terminal:

   ```
   average accuracy % increase for RAMSIS vs. Jellyfish: 5.70
   highest accuracy % increase for RAMSIS vs. Jellyfish: 8.07
   average accuracy % increase for RAMSIS vs. ModelSwitching: 3.48
   highest accuracy % increase for RAMSIS vs. ModelSwitching: 4.63
   ```

   The results correspond to Fig. 5 in §7.1 demonstrating that RAMSIS achieves significantly higher accuracy than state-of-the-art approaches on a real production trace given same number of workers. In other words, RAMSIS can achieve same accuracy as the baselines with less resources.

Experiment: constant load [5 human-minutes + 1 compute-hour]. Evaluate RAMSIS and baselines with 60 workers under Poisson arrivals with constant load on the image classification task.

1. **Simulation.** Simulate RAMSIS, ModelSwitching (MS), and Jellyfish+ (JF) on image classification task with 150 millisecond latency SLO, while sweeping load from 400 to 4,000 QPS in intervals of 400.

   ```
   python3 run_sim.py --m RAMSIS --trace constant
   python3 run_sim.py --m MS --trace constant
   python3 run_sim.py --m JF --trace constant
   ```

   This step measures the achieved accuracy and latency SLO violation rate of each technique and stores the results in `results/TASK_METHOD_TRACE_SLO_LOAD_*.json`.
2. **Plot Results.** Plots the accuracy and latency SLO violation rate across varying query load:

   ```
   python3 plot.py --trace constant
   ```

   The plots are printed to terminal and saved as `image_constant_accuracy.png` and `image_constant_violation.png`. The following output should be printed to terminal:

   ```
   average accuracy % increase for RAMSIS vs. Jellyfish: 3.70
   highest accuracy % increase for RAMSIS vs. Jellyfish: 15.04
   average accuracy % increase for RAMSIS vs. ModelSwitching: 1.84
   highest accuracy % increase for RAMSIS vs. ModelSwitching: 6.71
   ```

   The results correspond to Fig. 6 in §7.2 demonstrating that RAMSIS achieves significantly higher accuracy than state-of-the-art approaches under the range of satisfiable query load.

A.5 Notes on Reusability

The scripts we provide can be used to evaluate RAMSIS and baselines across configurations beyond those evaluated in §A.4.2.

The inference task, latency SLO, query load, and number of workers can be specified via commandline arguments for `RAMSIS_gen.py` for policy generation:

- `--task ["image" or "text"]` Specify image or text classification task.
- `--SLO [float]` Specify latency SLO in milliseconds.
- `--worker [int]` Specify number of workers.
- `--load [float]` Specify query load in queries per second.

For example, the following generates a RAMSIS policy for the text classification task with 200 millisecond latency SLO, query load of 10 queries per second, and 20 workers:

```
python3 RAMSIS_gen.py --task text --SLO 200 --worker 20 --load 10
```

The ModelSwitching policies used in the evaluation can be generated with `python3 MS_gen.py`. This script supports the same commandline arguments to generate policies as `RAMSIS_gen.py`.

The inference task, latency SLO, query load, and number of workers can be specified via commandline arguments for `run_sim.py` for simulation and `plot.py` as well:

- `--m ["RAMSIS", "JF", or "MS"]` Specify model selection method.
- `--trace ["real" or "constant"]` Specify query load trace as the production trace or constant load.
- `--task ["image" or "text"]` Specify image or text classification task.
- `--SLO [float]` Specify latency SLO in milliseconds.
- `--worker [int]` Specify number of workers.
- `--load [float]` Specify query load in queries per second (for "constant" load trace).

To reproduce the results of Figure 5 with SLO = 150ms and 100 workers for the image classification task:

```
python3 RAMSIS_gen.py --task image --SLO 150 --worker 100
python3 MS_gen.py --task image --SLO 150 --worker 100
python3 run_sim.py --m RAMSIS --task image --trace real --SLO 150 --worker 100
python3 run_sim.py --m MS --task image --trace real --SLO 150 --worker 100
python3 run_sim.py --m JF --task image --trace real --SLO 150 --worker 100
```

Then, to view the results:

```
python3 plot.py --task image --trace real --SLO 150 --worker 100
```

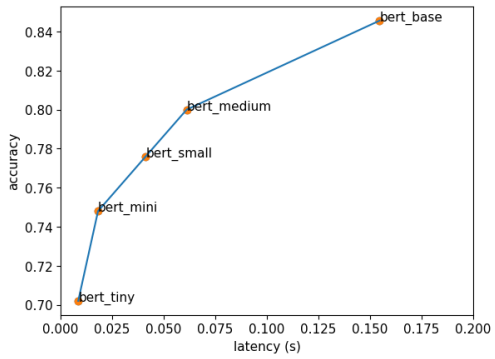

Figure 9. 95^{th} percentile inference latency vs. accuracy profile for text classification model set of 5 Bert models from Huggingface [11, 52]. Accuracy of each model is measured on the GLUE-MNLI test set [49].

B Model Profiles for Text Classification

Fig. 9 shows model profile for text classification task in our evaluation (§7).

C Impact of Time Discretization

Fig. 10 shows the accuracy of MS&S with RAMSIS using different D for FLD (§4.2.2). When $D = 100$, FLD performs similarly to MD (§4.2.1).

Overly-conservative actions can occur with FLD in the situation where the slack represented in the state space T_j underestimates the actual slack Δ. To avoid latency SLO violations, only actions which meets the slack T_j are considered at each decision (§4.3.1). However, in situations where T_j severely underestimates the actual slack Δ, the policy may be taking actions with significantly lower inference latency than necessary and may miss out on opportunities to select slower, but higher accuracy models. Note that with MD, this situation is not possible since the state space represents all possible relevant slack times defined by all possible inference latency are represented in the state space.

For FLD, as D decreases, in addition to the state space size decreasing, the distance between time lengths T_j, T_{j+1} increases since the distance between adjacent T_j, T_{j+1} is $\frac{SLO}{D}$. Thus decreasing D for policy generation makes it more likely to be in the situation where the actual slack is severely underestimated (i.e., $T_j << \Delta < T_{j+1}$) and may result in generating a more conservative policy. As D increases, accuracy improves. However, there are diminishing returns: the performance gap between $D = 100$ and $D = 10$ is smaller than between $D = 2$ and $D = 10$. This is because the adjacent slack times $T_j, T_{j+1} \in \mathbf{T}_w$ are increasingly similar as D increases and the generated policy often makes the same decisions for similar slack times.

D Impact of Batching

Fig. 11 compares the performance of MS&S with RAMSIS under the maximal and variable batching policies (see 4.3). The performance difference between these approaches in negligible, since variable batching selects maximum batch size (i.e., all pending queries in the worker queue) in 80% of its decisions.

E Evaluation with Fewer Models

Fig. 12 shows the comparison of Jellyfish+ and RAMSIS with model removed from the original model set of the image classification task. 3 model Jellyfish and 3 model RAMSIS demonstrate the results of when the approaches only have 3 models to select from. Overall, RAMSIS does not rely on many models to achieve high accuracy, and always achieves higher accuracy than Jellyfish+. The 3 models are chosen from Fig. 3 where the minimum latency model (shufflenet_v2_x0_5), a medium latency model (efficientnet_b2), and a long latency model (efficientnet_v2_s) are kept in the model set.

F Latency SLO Violation Rates

In this section we show the latency SLO violation rates for RAMSIS and baselines corresponding to the experiments conducted in §7.

Production Trace. Table 3 shows the SLO violation rates corresponding to Fig. 5 with the prototype RAMSIS implementation. Except for image classification with 20 workers, RAMSIS's violation rate is less than 1%. For image classification with 20 workers, both RAMSIS and the baselines cannot meet the load, i.e., the lowest latency model does not offer sufficient throughput.

Constant Query Load. Table 4 shows the SLO violation rates corresponding to Fig. 6 with the prototype RAMSIS implementation. Except for image classification at 3600-4000 QPS, RAMSIS violation rate is less than 1%. 3600-4000 QPS is near the peak throughput of the lowest latency model.

G Multiple Latency SLOs

RAMSIS handles multiple latency SLOs similar to existing systems [32]: each worker is assigned a latency SLO, per-SLO central queues are instantiated, and workers are associated with a central queue whose SLO matches.

H Notes on INFaaS

INFaaS [38] requires accuracy and latency SLOs from the application and its model selector and scheduler chooses the lowest cost model (i.e., typically lowest latency) that meets both. RAMSIS requires only a latency SLO and maximizes accuracy given the latency target. A fair comparison between RAMSIS and INFaaS would require re-designing RAMSIS to target the same objective as INFaaS. To demonstrate, we adapted INFaaS to our experimental evaluation

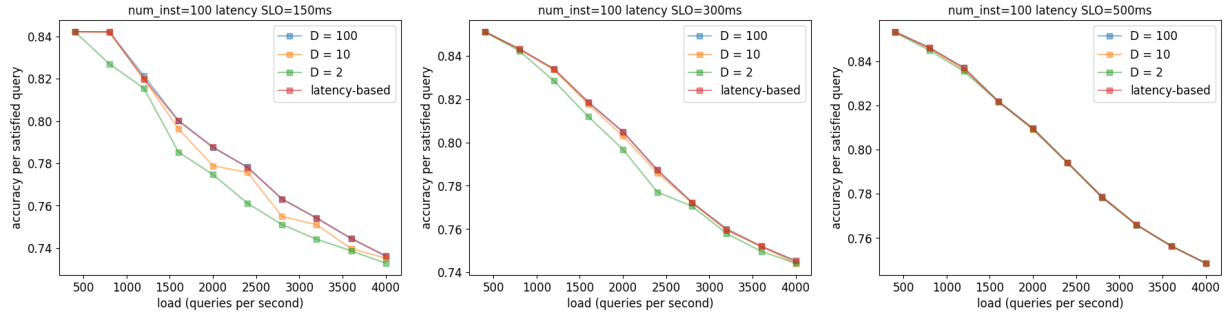

Figure 10. Comparison of time discretization strategies (see §4.2). With large enough D, FLD can match performance of MD.

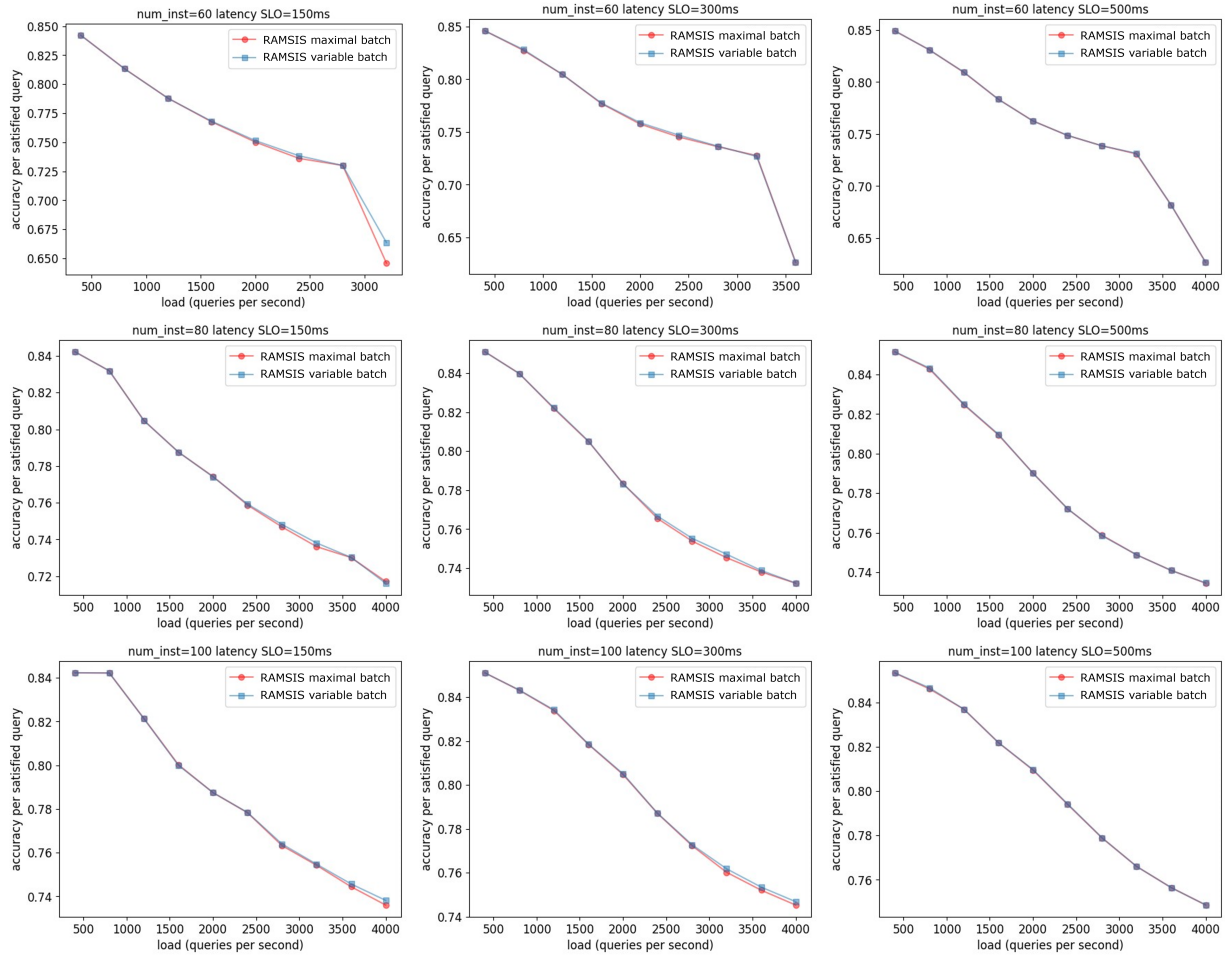

Figure 11. RAMSIS maximal batching vs. variable batching (see §4.3). Variable batching performs similarly to maximal batching.

(§7) by sweeping a range of accuracy targets equal to the set of accuracies achievable by each inference model. However, its objective to minimize latency effectively minimizes accuracy, and INFaaS always selects the minimally accurate model which achieves the accuracy target. As a result, we found that INFaaS performs no better than RAMSIS or the baselines (§7), reinforcing the notion that RAMSIS and INFaaS are not directly comparable due to differing objectives and constraints.

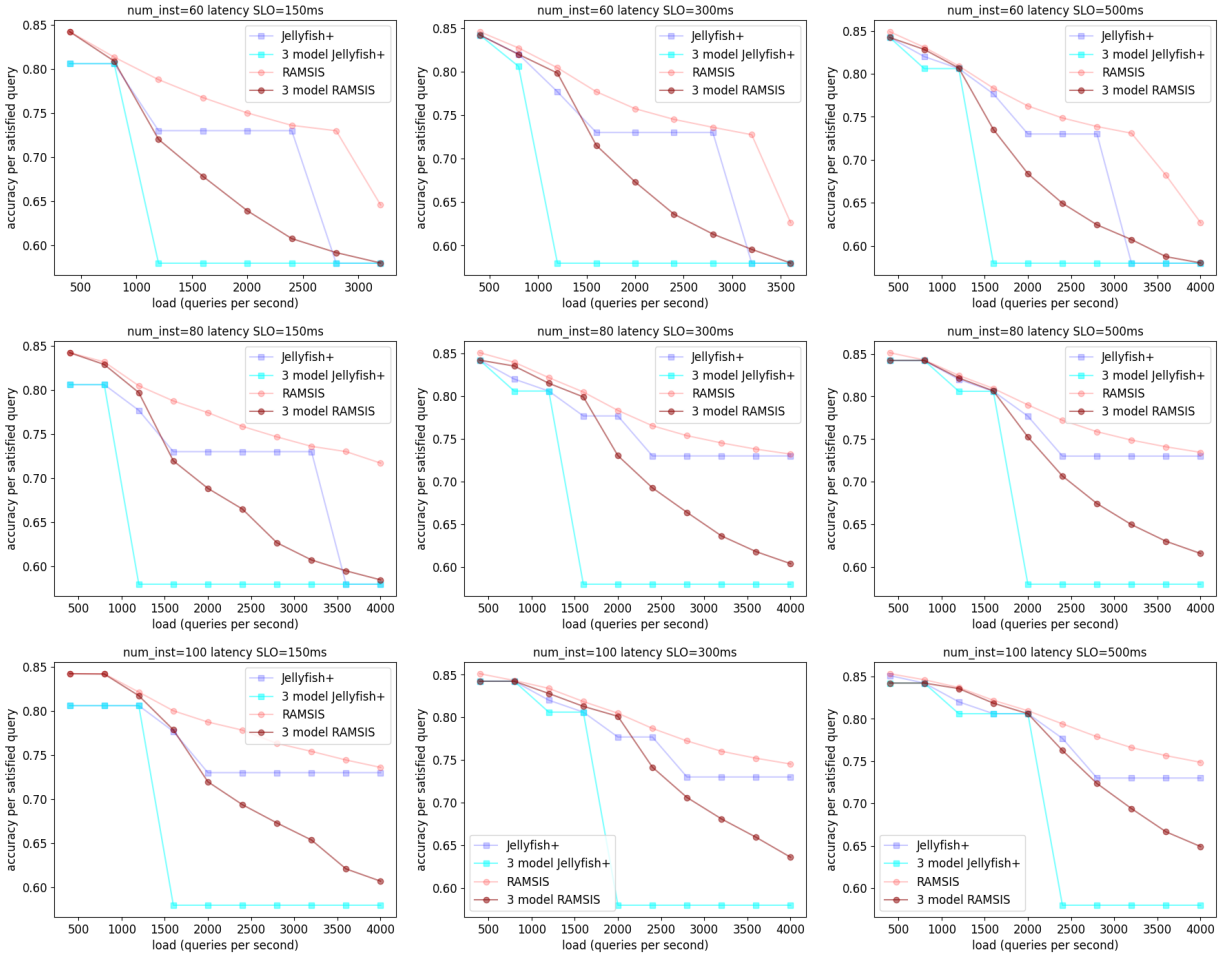

Figure 12. Ablation study to show impact of RAMSIS against existing approach from ablating model from the model set.

I Other Load Balancing Strategies

Our RAMSIS evaluation and implementation uses a round-robin load balancing strategy (§3). In this section, we explain how RAMSIS can be extended to other load balancing strategies.

The MDP transition probabilities are the only aspect of RAMSIS which directly depend on the load balancing strategy. In §4.4, we derive transition probabilities for worker w assuming round-robin load balancing applied to the arrival distribution to the central queue. Moreover, only case 2 of the transition probabilities (§4.4.2) depends on the load balancing strategy.

I.1 Shortest-Queue-First

To demonstrate how RAMSIS can be extended to another load balancing strategy, we show how to formulate the case 2 transition probabilities (§4.4.2) given a shortest-queue-first strategy. *Shortest-queue-first* load balancing (i.e., *join-the-shortest-queue* [18]) is a dynamic load balancing approach which on a query arrival to the central queue, assigns the query to the shortest worker queue (i.e., the worker queue with lowest number of queued queries) [18]. As in §4.4.2, we express the transition probabilities for the MDP of worker w in terms of the arrival distribution to the central queue PF assuming independent and stationary increments.

For case 2 (i.e., when $0 < n \leq N_w$ and $0 \leq n' \leq N_w$) given the same intervals B, C, and D from §4.4.2 (Fig. 4), we express the transition probability of a worker w's MDP between state $s = (n, T_j)$ and state $s' = (n', T_{j'})$ given action $a = (m, n)$ as:

$$\mathrm{P}[s' = (n', T_{j'})|s = (n, T_j), a = (m, n)] = \sum_{k_C^w} PF_w(k_B^w, T_B|s) * PF_w(k_C^w, T_C|s) * PF_w(k_D^w, T_D|s) \quad (4)$$

where $PF_w(k, T|s)$ denotes the probability of k arrivals at worker w during a time interval of length T given the current state s; k_B^w, k_C^w, and k_C^w correspond to the number of arrivals to worker w during intervals B, C and D, respectively (Fig 4). Note the distinction between PF_w and PF where PF denotes the arrival distribution at the central queue.

Recall T_B, T_C, T_D denote the time lengths of intervals B, C, and D, which for shortest-queue-first, are derived identically

Latency SLO (ms)	150			300			500		
#Workers	**RAMSIS**	**JF+**	**MS**	**RAMSIS**	**JF+**	**MS**	**RAMSIS**	**JF+**	**MS**
20	93.38%	100.00%	100.00%	81.77%	99.90%	99.90%	58.61%	99.76%	99.78%
30	0.6430%	0.8607%	0.9707%	0.4408%	0.0047%	0.0121%	0.0592%	0.2152%	0.1474%
40	0.2296%	0.6137%	1.3070%	0.0369%	0.3482%	1.0709%	0.1023%	0.3209%	0.2147%
50	0.0500%	0.1041%	0.2030%	0.0297%	0.0588%	0.1464%	0.0129%	0.0498%	0.1043%
60	0.1277%	0.4141%	0.4136%	0.1471%	0.2260%	0.6517%	0.1607%	0.0863%	0.2296%
70	0.2231%	0.7234%	0.2368%	0.5036%	0.0913%	0.0937%	0.2586%	0.0556%	0.1523%
80	0.4329%	0.8292%	0.6080%	0.2226%	1.0460%	0.7470%	0.1540%	0.0824%	0.3155%
90	0.4914%	0.1652%	0.2141%	0.1132%	0.2618%	0.2861%	0.1117%	0.1972%	0.2042%
100	0.4428%	0.5811%	0.4767%	0.3213%	0.4693%	0.5000%	0.4201%	0.2911%	0.1949%
Latency SLO (ms)	100			200			300		
#Workers	**RAMSIS**	**JF+**	**MS**	**RAMSIS**	**JF+**	**MS**	**RAMSIS**	**JF+**	**MS**
20	0.0009%	0.0032%	0.2244%	0.0001%	0.0144%	0.0727%	0.0009%	0.0000%	0.0043%
30	0.0013%	0.6427%	0.0005%	0.0051%	0.0005%	0.0023%	0.0101%	0.0004%	0.0000%
40	0.0157%	0.0016%	0.0013%	0.0174%	0.0386%	0.0188%	0.0024%	0.0072%	0.0083%
50	0.0354%	0.0009%	0.0406%	0.0492%	0.0081%	0.0863%	0.0119%	0.0293%	0.0964%
60	0.0557%	0.0063%	0.0615%	0.4642%	0.0892%	0.7564%	0.0591%	0.0416%	0.0580%
70	0.1366%	0.0011%	0.1308%	0.0235%	0.0437%	0.0980%	0.0281%	0.0029%	0.0080%
80	0.0680%	0.1382%	0.0775%	0.1376%	0.2673%	0.3012%	0.0049%	0.0002%	0.0165%
90	0.0122%	0.0063%	0.0549%	0.0187%	0.0163%	0.0540%	0.0044%	0.0002%	0.0029%
100	0.0201%	0.6011%	0.2081%	0.0647%	0.6527%	0.0678%	0.0162%	0.0193%	0.0557%

Table 3. Latency SLO violation rate for RAMSIS and baselines queried under the Twitter trace on the image classification (top) and text classification (bottom) task.

Latency SLO (ms)	150			300			500		
load (QPS)	**RAMSIS**	**JF+**	**MS**	**RAMSIS**	**JF+**	**MS**	**RAMSIS**	**JF+**	**MS**
400	0.0669%	0.0000%	1.2377%	0.0252%	0.0419%	0.0000%	0.5038%	0.0000%	0.0000%
800	0.1089%	0.2302%	1.3180%	0.4402%	0.2306%	2.6245%	0.7513%	0.0587%	0.5998%
1200	0.4990%	0.0279%	0.0306%	0.0725%	0.8504%	1.4995%	0.1312%	0.2090%	0.6742%
1600	0.3637%	0.1087%	0.0543%	0.3390%	0.0000%	0.0084%	0.2324%	0.1366%	1.7317%
2000	0.3076%	0.0150%	0.0000%	0.2876%	0.0067%	0.0284%	0.1739%	0.0938%	0.0033%
2400	0.0516%	0.0140%	0.0084%	0.0168%	0.1340%	0.1173%	0.0112%	0.3522%	0.0028%
2800	0.2288%	0.0695%	0.0084%	0.2278%	1.0823%	0.0970%	0.0239%	0.4759%	0.1678%
3200	0.4786%	3.6645%	0.0587%	0.0745%	0.1419%	0.0126%	0.0935%	0.6393%	0.0032%
3600	2.4540%	2.7240%	0.0019%	0.3375%	0.6982%	0.0000%	0.5560%	0.3343%	0.1402%
4000	19.460%	24.119%	97.874%	4.3920%	4.2465%	0.0210%	1.7970%	2.8462%	0.7437%
Latency SLO (ms)	100			200			300		
load (QPS)	**RAMSIS**	**JF+**	**MS**	**RAMSIS**	**JF+**	**MS**	**RAMSIS**	**JF+**	**MS**
400	0.0000%	0.0000%	0.0499%	0.0083%	0.0416%	0.1831%	0.0583%	0.0000%	0.1249%
800	0.0793%	0.0000%	0.0042%	0.0292%	0.0000%	0.0042%	0.0000%	0.6353%	0.0084%
1200	0.0056%	0.0111%	0.0000%	0.0194%	0.0000%	0.0000%	0.2137%	0.0000%	0.0000%
1600	0.0021%	0.0728%	0.0812%	0.0354%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%
2000	0.0000%	0.0033%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0000%	0.0033%
2400	0.0028%	0.0000%	0.0000%	0.0014%	0.0153%	0.0028%	0.2767%	0.0000%	0.0042%
2800	0.0048%	0.0000%	0.0000%	0.0072%	0.0072%	0.2219%	0.0000%	0.6573%	0.0131%
3200	0.0073%	0.0000%	0.0000%	0.0000%	0.0042%	0.0063%	0.0000%	0.0000%	0.0000%
3600	0.0065%	0.0000%	0.0000%	0.0019%	0.0000%	0.0223%	0.0000%	0.0000%	0.0009%
4000	0.0050%	0.0000%	0.2287%	0.0084%	0.0034%	0.0000%	0.0008%	0.0000%	0.0000%

Table 4. Latency SLO violation rate for RAMSIS and baselines queried under constant query load on the image classification (top) and text classification (bottom) task.

to §4.4.2. Recall that during interval B, the number of arrivals to worker w is 0 (§4.4.2). Thus, $k_B^w = 0$. Further, recall the number of arrivals to worker w during interval C is at least 0 and at most n'. Therefore, $k_C^w \in [0, n']$. Finally, recall that n' queries must arrive between s and s' to worker w. Thus $k_C^w + k_B^w + k_D^w = n'$ and $k_D^w = n' - k_C^w$.

With shortest-queue-first load balancing, given *PF* is a Poisson distribution with arrival rate λ at the central queue,

the arrival distribution to worker w, $PF_w(k, T|s)$, can be approximated as a conditional Poisson distribution with arrival rate $\lambda_w(n)$ [18]. Recall current state $s = (n, T_j)$ where n is the number of queued queries at worker w. We can approximate $\lambda_w(n)$ using an established approach [18]:

$$\lambda_w(n) = \begin{cases} (\frac{\lambda}{K\mu})^K \mu & n \geq 3 \\ \frac{\lambda}{K} & 0 \leq n \leq 2 \end{cases}$$

where μ is the mean inference latency of worker w. We conservatively assume μ is the maximum inference latency of the model that meets the query load given the latency SLO. Thus, we set $\mu = max_{m \in M^*_w} l_w(m, 1)$ such that $\exists b. l_w(m, b) \leq SLO/2 \wedge \frac{b}{l_w(m,b)} \geq \frac{\lambda}{K}$. M^*_w denotes the set of models on the Pareto Front of accuracy and latency on worker w (4.3.3). Recall that $l_w(m, b)$ denotes the inference latency of model m with batch size b on worker w and K is the total number of workers. Note $\lambda_w(n)$ for $n \geq 3$ is an approximation introduced in prior work [18], and $\lambda_w(n)$ for $0 \leq n \leq 2$ is the time-average arrival rate to worker w with shortest-queue-first load balancing [18]. Note that for when PF is not a Poisson distribution, PF_w can be empirically estimated using simulation.