Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling

Sohaib Ahmad
University of Massachusetts, Amherst
sohaib@cs.umass.edu

Hui Guan
University of Massachusetts, Amherst
huiguan@cs.umass.edu

Brian D. Friedman
Nokia Bell Labs
brian.friedman@nokia-bell-labs.com

Thomas Williams
Nokia Bell Labs
thomas.williams@nokia-bell-labs.com

Ramesh K. Sitaraman
University of Massachusetts, Amherst
ramesh@cs.umass.edu

Thomas Woo
Nokia Bell Labs
thomas.woo@nokia-bell-labs.com

Abstract

Existing machine learning inference-serving systems largely rely on hardware scaling by adding more devices or using more powerful accelerators to handle increasing query demands. However, hardware scaling might not be feasible for fixed-size edge clusters or private clouds due to their limited hardware resources. A viable alternate solution is accuracy scaling, which adapts the accuracy of ML models instead of hardware resources to handle varying query demands. This work studies the design of a high-throughput inference-serving system with accuracy scaling that can meet throughput requirements while maximizing accuracy. To achieve the goal, this work proposes to identify the right amount of accuracy scaling by jointly optimizing three sub-problems: how to select model variants, how to place them on heterogeneous devices, and how to assign query workloads to each device. It also proposes a new adaptive batching algorithm to handle variations in query arrival times and minimize SLO violations. Based on the proposed techniques, we build an inference-serving system called Proteus and empirically evaluate it on real-world and synthetic traces. We show that Proteus reduces accuracy drop by up to 3× and latency timeouts by 2-10× with respect to baseline schemes, while meeting throughput requirements.

CCS Concepts: • **Computer systems organization → Cloud computing**; *Neural networks.*

Keywords: Inference serving, model serving, machine learning, autoscaling

ACM Reference Format:
Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. 2024. Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3617232.3624849

1 Introduction

The growing popularity of machine learning (ML) has led to the development of inference-serving systems[1], where cloud providers execute pre-trained ML models on their infrastructure to provide fast and accurate responses to inference queries. In such a system, the provider guarantees certain service level objectives (SLOs) to users in terms of latency deadlines. Meanwhile, the provider wants to maximize their hardware utilization by increasing throughput in order to serve as many queries as possible in a short amount of time.

Recent years have seen considerable interest in developing high-throughput inference-serving systems. Prior work has focused on enhancing system throughput through techniques such as multi-tenancy [28, 35], batching [8, 9, 36], and low-level optimizations [14, 42, 46]. While these optimizations are effective, they primarily rely on *hardware scaling*, i.e., adding more devices or using more powerful accelerators, to accommodate higher query demands (in terms of queries per second or QPS). However, hardware scaling may not be feasible for private clouds owned by enterprises or edge clusters due to the limited availability of hardware resources. Purchasing and maintaining more devices to cater to peak demands can be expensive and cost-ineffective.

To address the limitations of hardware scaling, this paper focuses on an orthogonal perspective, *accuracy scaling*, which adapts model accuracy instead of hardware resources to meet varying query demands. Accuracy scaling is motivated by the fact that ML models can offer varying degrees of accuracy depending on the time spent computing the answer. Less time spent on computation leads to less accurate

[1]Inference-serving systems have also been referred to as model-serving systems in the literature.

results but higher throughput. When the demand is high, an inference-serving system can serve queries using less accurate model variants to avoid SLO violations. Similarly, when the demand drops, it can serve queries using more accurate model variants to improve accuracy. These different accuracy levels are provided by different variants of deep neural networks (DNNs), which can be created through existing compression techniques such as quantization [39] and pruning [6], or as part of the neural network architecture design. The creation of these variants is complementary to our work. Throughout the paper, we use the term "model variants" to refer to DNN models with varying accuracy and performance profiles.

In this paper, our goal is to build an inference-serving system on a heterogeneous cluster via accuracy scaling such that the system can serve varying query demands with high system accuracy. System accuracy refers to the averaged accuracy of queries that are served by the system. We assume that every query prefers the most accurate model variant if resource permits, but is willing to accept lower accuracy in exchange for timely response when resources are tight. This assumption is practical in many cases such as recommendation systems and real-time applications where a timely response can be more critical than an accurate yet sluggish response, or even worse, dropped requests [13, 23].

To achieve this goal, we need to address two challenges raised by accuracy scaling when managing resources on an inference-serving system. The first challenge is *to determine the right amount of accuracy scaling to serve a target query demand with high system accuracy.* The system can overshoot the query demand by hosting less accurate model variants but introduce unnecessary accuracy drops to end-users. On the other hand, if the system undershoots the target query demand by hosting more accurate model variants, users will experience SLO timeouts.

To address the first challenge, our key insight is that identifying the right amount of scaling requires *jointly* optimizing three sub-problems when allocating system resources. Missing any of these sub-problems results in a sub-optimal solution that leads to high SLO violations and/or high accuracy loss. The first sub-problem, *model selection*, aims to choose the appropriate set of model variants and the number of replicas for each model variant. As the accuracy-throughput trade-off for a model variant varies across different device types, the second sub-problem, *model placement*, determines the placement of each selected model variant on a particular device of a heterogeneous cluster. In addition, an inference-serving system is usually shared by multiple applications, each corresponding to a query type. The last sub-problem *query assignment* specifies the percentage of queries from each query type that can be assigned to a model variant hosted on a particular device.

Based on this insight, we formulate the problem using a mixed integer linear programming (MILP) framework to derive the optimal resource allocation given a target query demand. Since the target query demand changes over time, it is necessary to keep adjusting the plan accordingly. Ideally, each query arrival could trigger the MILP solver; however, in this case, solving the MILP problem would lie in the critical path of query execution, introducing significant run-time overhead. This motivates us to decouple the control path that manages resources from the data path that serves queries. We trigger the MILP solver in response to *macro-scale* changes in query demand over a period of time to ensure that the system has sufficient capacity to serve incoming queries. We then rely on per-device query execution to handle the *micro-scale* variations in the arrival times of queries.

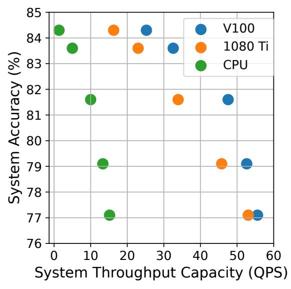
This raises the second challenge: *how to adaptively batch queries on each device to handle variations in query arrival times such that SLO violations are minimized?* Query execution on individual devices typically batches multiple inference queries together to improve throughput. When queries arrive at a uniform rate, it is easy to determine a fixed batch size that can maximize throughput without incurring SLO violations. However, when there is variation in the query inter-arrival times, batching needs to be adaptive due to two reasons: (i) the number of queries arriving per second may change over time, requiring a different batch size, and (ii) delaying query execution slightly can improve throughput and absorb micro-scale query arrival variations. We present a non-work-conserving approach to adaptive batching that can handle *micro-scale* query load fluctuations effectively without requiring any changes to the underlying ML framework. It can delay requests momentarily if it results in higher throughput and dynamically determines suitable batch sizes based on the queue status and query latency requirements.

Based on the proposed techniques, we build an inference-serving system called Proteus[2]. We evaluate Proteus against three state-of-the-art inference-serving systems, INFaaS [35], Sommelier [17], and Clipper [9], using query workloads derived from a real-world Twitter trace and synthetic traces. Our experiments show that compared to baselines that do not scale accuracy, Proteus improves system throughput by 60% and reduces SLO violations by 10× due to accuracy scaling; and compared to baselines that also scale accuracy, Proteus minimizes accuracy drop by up to 3.2× and reduces SLO violations by up to 4.3× because of better resource allocation and batching algorithms.

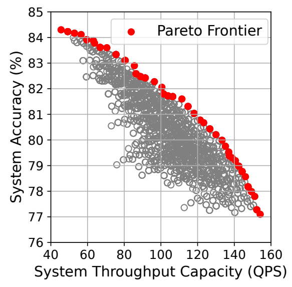
Our contributions. We make the following key contributions.

- We present a theoretical framework for resource management of an inference-serving system that exploits *accuracy scaling* to ensure that the system throughput is sufficient to meet the query demand while maximizing system accuracy.

[2]Proteus is named after a Greek god of the same name who had the ability to change shape and form to avoid capture by the enemy.



(a)



(b)

Figure 1. Illustration on how different resource allocation configurations affect system throughput capacity and accuracy. Models are EfficientNet variants and batch size is one. (a) Accuracy-throughput trade-off on three different devices for EfficientNet variants. (b) Accuracy-throughput trade-offs for all possible configurations with five EfficientNet variants and five devices.

- We propose a proactive adaptive batching algorithm that can handle query load fluctuations effectively via a non-work-conserving approach. The algorithm is easy to implement without requiring modifications to the underlying ML framework and reduces SLO violations by up to 4× compared to baselines.
- We design Proteus, a high-throughput inference-serving system with accuracy scaling. Proteus decouples the control and data paths of inference-serving to perform optimal resource allocation asynchronously from query serving. To the best of our knowledge, Proteus is the first system to study accuracy scaling in a cluster setting.
- We evaluate Proteus on a production system that is currently used by actual users within a large enterprise and perform trace-based simulations of the system. We show that Proteus reduces the accuracy drop by up to 3.2× and SLO violations by 2.8-10× compared to state-of-the-art baselines while meeting throughput requirements. We also show that our simulation results closely match the results from our production system.

2 Motivation and Challenges

This section first explains the motivation for accuracy scaling and then elaborates on the main challenges to effectively capitalize on accuracy scaling to build a high-throughput inference-serving system with minimal SLO violations.

2.1 Motivation

Many existing inference-serving systems rely on scaling out to more devices or scaling up to faster devices to handle the increasing query demands. However, in certain scenarios, such hardware scaling is not practical. For example, an edge cloud-based serving system may have a limited number of devices available, and an enterprise may maintain a fixed-size cluster to serve internal users.

A viable alternative in these cases is accuracy scaling, which is motivated by the natural trade-off between throughput and accuracy of inference workloads. This trade-off exists because models with lower accuracy are usually smaller neural networks, e.g., having a smaller number of parameters and layers, etc., therefore requiring less computation, allowing more requests to be served in a given amount of time, and providing higher throughput. Figure 1a shows the accuracy of EfficientNet variants [40] and their throughput in queries per second (QPS) on three different types of devices. The batch size is set to one for simplicity of demonstration. The figure shows that for a given device type, a model variant with lower accuracy can achieve higher throughput. An inference-serving system can replace hosted model variants with their less accurate counterparts in order to serve an increasing query load, and later switch to more accurate model variants once the query load returns to normal.

2.2 Challenges and Opportunities

Building an inference-serving system has two main challenges as detailed below.

Challenge 1: "Right" amount of accuracy scaling. The first challenge lies in determining the right amount of accuracy scaling to meet a target query demand. With traditional hardware scaling, an inference-serving system would provision the appropriate amount of hardware resources sufficient to meet a target query demand. Hardware under-provisioning would lead to performance degradation while over-provisioning would introduce unnecessary hardware costs. Similarly, with accuracy scaling, one can *under-scale* or *over-scale*. In one extreme case, the system can always host the most accurate model variants to maximize accuracy but will suffer from relatively low throughput and risk high SLO violation rates. In another extreme, the system can always host the least accurate model variants to maximize throughput, but will suffer from unnecessary accuracy drops. The right amount of accuracy scaling means that the system hosts the right set of model variants on the right devices and assigns the right amount of queries to each device. Solving the above resource allocation problem is complicated because of the huge configuration space resulting from three factors: many model variants, heterogeneous devices, and many applications. Missing any factor would lead to a sub-optimal allocation, as we show empirically in our evaluation.

Factor 1: Many model variants. A query can be served by many variants of the same model, resulting in a wide range of throughput and accuracy. Figure 1a demonstrates this problem. Even if we consider a single device, e.g., NVIDIA GTX 1080 Ti, five different selections of EfficientNet variants result in a 3.3× difference in throughput and around 8% difference in accuracy. The problem becomes more difficult when we have a cluster of devices, resulting in an even wider range of possibilities. With M model variants and N devices, there are M^N possible configurations, each of them resulting in different system throughput and accuracy. Selecting the right model variant for each of the devices depends on the query demands on the system.

Factor 2: Heterogeneous devices. Modern computing clusters typically consist of different types of hardware devices, such as different types of CPUs and GPUs. This is due to the fact that enterprises typically upgrade their hardware incrementally, for example, adding newer and faster GPUs to their existing fleet of hardware as they become available, instead of completely replacing old ones. As each model variant has a different throughput profile across devices, the configuration space becomes even larger. For a given selection of model variants, there are $N!$ ways to place them on N different devices, leading to a much broader set of possibilities for overall system throughput and accuracy. Figure 1b illustrates the system throughput capacity and accuracy for all possible mappings between 5 EfficientNet variants and 5 devices. We assume that all devices serve the maximum number of queries feasible without SLO violations. Out of all 3125 possible configurations, we are only interested in those at the Pareto frontier, due to the fact that for a given throughput requirement, configurations at the Pareto frontier yield the highest possible system accuracy compared to other configurations. The results imply that identifying the right amount of accuracy scaling requires co-optimizing model selection and model placement.

Factor 3: Many applications. The optimization problem is further complicated by the multi-tenant nature of inference-serving systems. An inference-serving system usually serves queries from many applications, each corresponding to a particular query type. In this case, the model variants that are selected for each query type and their mapping to devices further depend on how many and what kind of devices are allocated to each query type. When the query demand for any query type changes, the best resource allocation plan to achieve the highest accuracy also changes.

Opportunities. The above challenge can be formulated as three sub-problems for resource allocation: *model selection*, *model placement*, and *query assignment*, that can be jointly solved using mixed integer linear programming (MILP). We elaborate on the MILP formulation in Section 4. Solving the MILP problem takes time, while queries have strict latency requirements. Triggering the MILP solvers on query arrivals would place the resource allocation problem on the critical path of serving queries, adding a significant run-time overhead to queries' response time. To avoid the problem, we decouple resource allocation from query serving. Resources are re-allocated when the macro-scale query demand changes to ensure that the system has enough throughput capacity to handle incoming queries. As it comes down to each device to serve individual queries, each device can handle non-uniform query arrivals with adaptive batching to avoid SLO violations. This motivates us to design an inference-serving system that separates the control path that manages resources from the data path that serves queries (see Section 3).

Challenge 2: Adaptive Batching. Decoupled resource allocation and query serving rely heavily on adaptive batching to handle micro-scale fluctuations in query arrival times. Batching multiple queries together can improve the throughput of an inference-serving system. However, it can also increase the waiting time for queries since all queries that are executed in a batch must finish processing before the results are returned to the user. If the batch size is increased beyond a certain point, requests will start to miss their latency SLOs. Since queries arrive non-uniformly, we need to dynamically adjust batch sizes based on query arrival times to minimize SLO violations.

Although adaptive batching is studied in prior work, we find their approaches inefficient. Clipper [9] adapts its batch size reactively based on whether the current batch size causes timeouts. Nexus [36] presents a proactive early dropping approach but performs poorly when query load fluctuates due to its work-conserving approach. Lazy Batching [8] requires significant changes to the underlying ML framework.

Opportunities. We find that a non-work-conserving approach for adaptive batching is better than a work-conserving approach in stabilizing the system when query inter-arrivals are non-uniform. In a work-conserving approach, a worker immediately executes the next batch of queries after the current batch finishes. This approach is adopted by Nexus and demonstrated to suffer from low throughput and high SLO violations on bursty workloads in our empirical evaluation (Section 6.4). In contrast, a non-work-conserving approach waits as long as possible to accumulate more queries before executing a batch. Although it may leave the device idle at times if the load is low, we show that it improves system throughput and reduces SLO violations. Moreover, we find that a proactive approach to batch size adaptation incurs significantly fewer latency SLO timeouts than a reactive approach, such as Clipper's, as detailed in Section 6.4.

We present a novel adaptive batching algorithm following a proactive, non-work-conserving approach without requiring modifications to the ML framework in Section 5.

3 Overview of Proteus

We present Proteus, a high-throughput inference-serving system that leverages accuracy scaling to handle varying query demands. This section presents an overview of the system architecture while Sections 4 and 5 will elaborate on the core modules of Proteus.

Figure 2 illustrates the overall system architecture of Proteus. It has three major components: Controller, Load Balancers, and Workers. These components are involved differently in two types of interactions with the system.

The first type of interaction is for developers to register an application and its model variants. The pipeline is marked with dotted arrows in Figure 2. After the controller

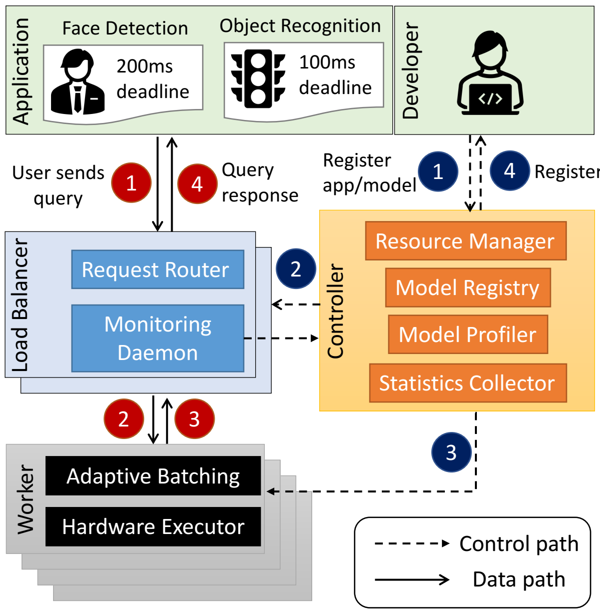


Figure 2. System architecture of Proteus.

receives an application register command, it creates a new load balancer for that application and sets up workers to serve queries from that application. Proteus will automatically manage which model variant to use to serve each query and where to place the model variants. This model-less interface is similar to the recent work, INFaaS [35]. *Our core system design contribution lies in how resources are managed when serving queries, i.e., the second type of interaction described next.*

The second type of interaction is for registered applications to send inference queries and receive query responses. The data path is marked with solid arrows in Figure 2. A query from a registered application is directly sent to the application's assigned load balancer. The load balancer then routes the query to an appropriate model variant that is hosted on a worker machine for query execution. Proteus responds to the application with the inference results. Since queries from each application are handled by its specific load balancer, Proteus avoids a single-point performance bottleneck when supporting many applications. The separation of the controller and load balancer is a design choice that makes Proteus more flexible and robust, allowing it to perform resource allocation without being on the critical path to inference-serving.

Controller. The controller receives the registration of the application and model variants and confirms the registration status. It has four modules: (1) a *resource manager* that determines the resource allocation strategy including model selection, placement, and query assignment when the query demands change, (2) a *model registry* that handles application and model variant registration, (3) a *model profiler* that profiles the performance of each model variant on different types of devices, and (4) a *statistics collector* that collects query demand statistics from all load balancers, used to determine when to re-allocate resources. If re-allocation is needed, these statistics are used as inputs to the resource manager to derive a new optimal resource allocation configuration. Moreover, the resource manager also consults the model registry and model profiler to make re-allocation decisions, in order to identify model variants that can meet demand with high accuracy. The model profiler is invoked every time new models are registered, as well as periodically to poll the inference latency of currently hosted model variants running on the workers, and stores the profiling information in an in-memory key-value store, keyed by the 3-tuple (model variant, device type, batch size) to ensure a $O(1)$ lookup time.

Load Balancer. A load balancer receives inference queries from its designated application and responds to them with model execution results. Load balancers for different registered applications could be distributed across multiple machines to avoid network congestion. Each load balancer includes two modules: (1) a *request router* that dispatches queries to model variants hosted on worker machines based on a query assignment policy, and (2) a *monitoring daemon* that monitors query demands at runtime and reports the statistics to the controller periodically. The query assignment policy is determined by the controller. If the load balancer detects a burst of requests or overload on any of its workers, it calls the controller to re-allocate resources. There is one load balancer for each query type (i.e., application), and usually multiple workers for each query type.

Workers. Each worker executes its hosted model variant to serve inference queries assigned by the request router. A worker includes two modules: (1) an *adaptive batching* module that dynamically determines the suitable batch size to improve throughput while meeting query latency constraints, and (2) a *hardware executor* that manages the deployment and execution of model variants.

We next explain the two core modules, the Resource Manager in Section 4 and Adaptive Batching in Section 5.

4 Resource Management

The Resource Manager identifies the optimal model selection, model placement, and query assignment solution to meet a target query demand while maximizing system accuracy by solving an MILP optimization. Depending on the solution, it could terminate some instances of model variants currently hosted on devices and start instances of other model variants. It also propagates a new query assignment policy to the request routers. Under stable query demand conditions, the Resource Manager is invoked periodically. However, if the demand changes rapidly in a short period of time, the monitoring daemon in the load balancers invokes the Resource Manager to respond to the burst.

Note that the Resource Manager responds to *macro-scale* changes in the workload, measured by the incoming demand in terms of QPS, whereas the adaptive batching, as explained

Constants/subscripts:	
q	the q-th query type. Each type is a registered application.
m	the m-th model variant for a query type
d	the d-th device
A_m	the accuracy of model variant m
$B_{m,q}$	true if model variant m serves query type q; false otherwise
$P_{d,m,q}$	the peak throughput capacity of model variant m on device d serving query type q
Inputs:	
s_q	throughput in QPS required for the query type q
Optimization variables:	
$x_{d,m}$	true if model variant m is hosted on device d; false otherwise
$y_{d,q}$	percentage of queries of type q routed to device d
Intermediate variables:	
a_q	sum of accuracy of all queries of type q
$z_{d,q}$	number of queries served by device d of query type q

Table 1. Notation used for the optimization

in Section 5, responds to *micro-scale* changes in terms of varying query inter-arrival times.

The resource management problem. We now formulate the resource management problem with accuracy scaling using MILP. The objective is to maximize accuracy while meeting a target serving throughput. We first explain two optimization variables collectively representing the resource allocation plan and then define the system serving throughput and system accuracy based on the two variables. Lastly, we present the MILP formulation of the resource management problem. Table 1 summarizes the notations.

Optimization Variables: Let $\{x_{d,m}\}$ be Boolean variables indicating the model selection and placement policy, together called model allocation. $x_{d,m}$ is True if model variant m is hosted on the device d. Let $\{y_{d,q}\}$ be a query assignment plan where $y_{d,q} \in [0, 1]$ indicates the percentage of queries of type q routed to device d. Each query type corresponds to one registered application and can only be served by the registered set of model variants.

The two variables must meet three constraints. First, in this work, we consider that each device hosts at most one model variant to avoid interference (Eq. 1). Second, the total ratio of queries for a given type routed to all devices can never be larger than one (Eq. 2). Third, a query assignment must ensure that the model variant hosted on a device supports the assigned query type (Eq. 3). Let $B_{m,q}$ be the Boolean constant that denotes whether model variant m can serve query type q. We formalize the three constraints as follows:

$$\sum_m x_{d,m} \leq 1 \qquad \forall d \qquad (1)$$

$$\sum_d y_{d,q} \leq 1 \qquad \forall q \qquad (2)$$

$$\sum_m \sum_d B_{m,q} \cdot x_{d,m} \cdot y_{d,q} = \sum_d y_{d,q} \qquad \forall q \qquad (3)$$

Serving Throughput: Let $z_{d,q}$ be the number of queries per second of query type q served by the device d. System serving throughput is the number of queries served by all devices: $\sum_d \sum_q z_{d,q}$. Note that $\sum_q z_{d,q}$ cannot be larger than the total number of queries assigned to device d (Eq. 4) or the peak throughput capacity of that device (Eq. 5). Furthermore, we require that all incoming demand be served by the system (Eq. 6). Let s_q be the number of queries per second (QPS) for the query type q. The total number of queries assigned to device d is: $\sum_q y_{d,q} . s_q$. Let $P_{d,m,q}$ be the peak throughput capacity of the model variant m profiled on device d for the query type q. The peak throughput of device d for query type q is then $\sum_m P_{d,m,q} . x_{d,m}$. The serving throughput of a device d follows three constraints:

$$\sum_q z_{d,q} \leq \sum_q y_{d,q} . s_q \qquad \forall d \qquad (4)$$

$$z_{d,q} \leq \sum_m P_{d,m,q} . x_{d,m} \qquad \forall d, q \qquad (5)$$

$$\sum_d z_{d,q} = s_q \qquad \forall q \qquad (6)$$

Effective Accuracy (also called System Accuracy): For each model variant m, the number of queries of query type q served by it is $\sum_d \sum_q x_{d,m} . z_{d,q} . s_q$. Let A_m be the accuracy of model variant m. We can get the accuracy of all queries of type q as: $a_q = \sum_m A_m \cdot (\sum_d x_{d,m} . z_{d,q} . s_q))$. Effective accuracy is the average accuracy of all queries served as $\sum_q a_q$.

MILP Formulation: The resource management problem identifies the optimal model selection and placement $\{x_{d,m}\}^*$ and the query assignment $\{y_{d,q}\}^*$ to maximize effective accuracy $\sum_q a_q$ while reaching a target serving throughput $\sum_d \sum_q z_{d,q}$ high enough to serve the incoming queries $\{s_q\}$. The problem can be formulated as:

$$\max_{\{x_{d,m}\},\{y_{d,q}\}} \sum_q a_q \quad s.t. \text{ Constraints Eqs. 1-6} \qquad (7)$$

Solving the MILP. The Resource Manager solves the MILP exactly to identify a global optimal model allocation and query assignment policy. Note that the time overhead to solve the MILP does not lie on the critical path of query serving as the MILP is called asynchronously. We provide overhead details in Section 6.8. When solving the MILP, s_q is set to be the demand by default. However, if demand increases beyond a certain point, even using the lowest accuracy model variants for every query type might still not meet throughput demand. In this case, the MILP solver immediately reports that the constraints are infeasible, and we solve the MILP again by decreasing s_q by a small value.

Estimation of throughput capacity. Solving the MILP problem requires us to estimate each $P_{d,m,q}$, the throughput capacity of each model variant on a device for a query type. Increasing the batch size improves the throughput of a model variant but also increases the processing latency. So we first estimate the maximum batch size that we can use for each model variant without violating a query's latency SLO and then profile the throughput capacity using that batch size. Specifically, [36] observes that to prevent latency timeouts, the maximum inference latency for any model cannot exceed half of its latency SLO since in the worst case, a query arriving just after a batch starts executing must be executed with the next batch, so the response time for the query is at most twice the processing latency. Using this observation, we calculate the maximum batch size for each (d, m, q) pair that meets the SLO requirement. Note that in addition to latency constraints, the maximum batch size is also bounded by the memory constraint of each device, since larger batch sizes require more memory. Hence, the maximum allowed batch size is the minimum of the following: (i) the maximum batch size that meets SLO, (ii) the maximum batch size that fits in the memory of d.

We use the maximum allowed batch size of each (d, m, q) pair, along with the profiled latency of model variant m on device d for query type q, to calculate the throughput capacity $P_{d,m,q}$ of that pair.

$$P_{d,m,q} = \frac{\text{Maximum allowed batch size for } d, m, q}{\text{Profiled latency (seconds)}}$$

5 Adaptive Batching

While the Resource Manager makes model allocation and query assignment decisions based on a target serving throughput, it is the responsibility of each device to serve queries assigned to it without violating the latency constraints. Adaptive batching dynamically determines the optimum batch size to use based on queue conditions to minimize SLO violations.

The Proteus adaptive batching algorithm is based on two key ideas. Firstly, it is a *proactive* algorithm: it ensures that no queries in the queue timeout unnecessarily since we proactively start processing the queries just before the first query in the queue is in danger of violating its latency SLO. Secondly, it is *non-work-conserving*: it may leave the device idle at times if this helps to accumulate more queries before starting batched execution. This allows the algorithm to improve throughput as much as possible on a given device without violating latency SLOs. This also helps to smooth out non-uniform query inter-arrivals in order to handle micro-scale query demand variations.

Figure 3 illustrates the approach. Suppose that we have q queries in the queue and that the first query will expire at $T_{exp}(1)$. To process a batch of q+1 queries, time $T_{process}(q+1)$ is required. We define $T_{max\_wait}(q+1) = T_{exp}(1) - T_{process}(q+1)$, or in other words, the maximum time that we will wait for the q+1st query to arrive. If we have not reached $T_{max\_wait}(q+1)$ yet, we can wait for more queries to arrive in the queue since we are not in danger of violating any query's latency SLO. While waiting until $T_{max\_wait}(q+1)$ to fill up the batch, there can be two possibilities:

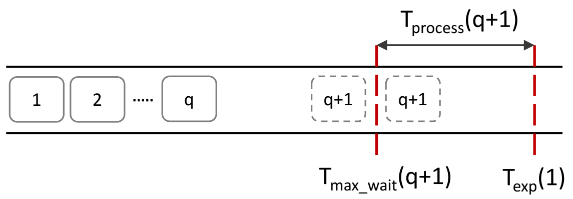


Figure 3. Adaptive batching in Proteus

Case 1: We do not receive any query until $T_{max\_wait}(q+1)$. In this case we will start executing the current queries in the queue with a batch size of q at $T_{max\_wait}(q+1)$, because if any query arrives after this time and we were to execute with a batch size of $q+1$, the first query in the queue would expire by the time the batch finishes processing.

Case 2: We receive the $q+1$st query before $T_{max\_wait}(q+1)$. In this case, we calculate $T_{max\_wait}(q+2)$. Note that $T_{max\_wait}(q+2) < T_{max\_wait}(q+1)$ since $T_{process}(q+2) > T_{process}(q+1)$. If we are already past $T_{max\_wait}(q+2)$, that means we cannot wait for the $q+2$nd query; otherwise, our first query will expire, so we execute with a batch size of $q+1$ which will not result in any timeouts since we execute before $T_{max\_wait}(q+1)$. If we are not past $T_{max\_wait}(q+2)$, then we wait to accumulate more queries in the queue and repeat the same procedure with $q' = q+1$.

As we will see in Section 6.4, the proposed batching algorithm outperforms re-active approaches, e.g., Clipper's AIMD batching, and even proactive work-conserving approaches, e.g., Nexus's early drop batching.

6 Evaluation

This section evaluates the efficacy of Proteus. We begin by describing the experimental setup common to all experiments (Section 6.1). We provide an end-to-end quantitative analysis on the performance of Proteus and baselines (Section 6.2). We also measure the responsiveness of each of these approaches to bursty workloads (Section 6.3). We evaluate Proteus's adaptive batching algorithm individually to Clipper and Nexus's batching algorithm (Section 6.4). We then perform an ablation study of Proteus to quantify the benefit of its individual components (Section 6.5). We also report the effect of varying latency SLOs (Section 6.6) and the performance breakdown for different model families (Section 6.7). Finally, we quantify the overheads of Proteus's decision-making (Section 6.8).

6.1 Experimental Setup

6.1.1 Baselines. We select a spectrum of baselines ranging from fully static (Clipper), to partially dynamic (Sommelier), to fully dynamic (INFaaS). We describe each of them below and implement them in our simulator and cluster system.

Clipper: Clipper is one of the fundamental works in inference-serving, however, it does not dynamically place models in a heterogeneous cluster or perform any accuracy scaling. This baseline pre-loads a static resource allocation at the start of the experiment. We extend Clipper to use our MILP to get this

Feature	Clipper	Sommelier	INFaaS	Proteus
Model placement	Static	Static	Heuristic	MILP
Model selection	Static	Heuristic	Heuristic	MILP
Accuracy scaling	No	Limited[3]	No[4]	Yes
Adaptive batching	Yes	No	Yes	Yes

Table 2. Comparing Proteus features with baselines

initial allocation. Since Clipper cannot scale, we implement two versions, one that maximizes throughput by using the least accurate model variants: **Clipper-HT** (High Throughput), and another that maximizes serving accuracy by using the most accurate model variants: **Clipper-HA** (High Accuracy). Clipper uses an additive-increase multiplicate-decrease (AIMD) heuristic to perform adaptive batching. We first evaluate Clipper end-to-end with our system and then separately evaluate its batching algorithm against ours in Section 6.4. Note that Clipper is also representative of other static inference-serving systems, such as TensorFlow-Serving [32] and NVIDIA Triton Inference Server [5] since none of these systems dynamically adapt the resource allocation of model variants in a cluster and depend on the application developer to handle it.

Sommelier: Sommelier can suggest alternate models to meet throughput requirements for a single server device and switch out a high accuracy model variant for a low accuracy variant when high throughput is needed. We refer to Sommelier as being partially dynamic as it only works with a single device and model but does not perform resource allocation at the cluster level. Moreover, Sommelier does not include adaptive batching by default as it is primarily a model repository that can interface with an inference-serving system and is not designed as an inference-serving system itself. Therefore we extend it by using our adaptive batching algorithm and MILP to get its initial model placement.

INFaaS: INFaaS is the most dynamic baseline we use as it dynamically changes model selection and model placement. However, INFaaS does not perform accuracy scaling by default, since it treats accuracy as a constraint to be met and minimizes the cost of running instances. In our setting, since the cost of the cluster is fixed, we tweak INFaaS slightly to use accuracy drop as its objective to minimize while using the cost of running instances as a constraint, i.e., it cannot exceed the fixed size of the cluster. We call this baseline *INFaaS-Accuracy*, a version of INFaaS optimized for accuracy scaling. *INFaaS-Accuracy* minimizes accuracy drop using the allocation strategy from INFaaS, thereby maximizing serving accuracy and giving us a version of INFaaS directly comparable to Proteus. Note that INFaaS-Accuracy just swaps accuracy and cost in INFaaS as its objective and constraint, respectively.

[3]Sommelier can suggest alternate models for a single device but does not perform allocation at the cluster level.

[4]INFaaS does not perform accuracy scaling by default, however we tweak it slightly to get a version of INFaaS that can scale accuracy.

Model Family	**Model Variants**
ResNet (classification) [20]	18, 34, 50, 101, 152
DenseNet (classification) [22]	121, 161, 169, 201
ResNest (classification) [47]	14, 26, 50, 269
EfficientNet (classification) [40]	b0-b7
MobileNet (classification) [21]	1.0, 0.75, 0.5, 0.25
YOLOv5 (object detection) [25]	n, s, m, l, x
BERT (sentiment analysis) [11]	RoBERTa-base, large; [29], ALBERT-base, large, xlarge, xxlarge [26]; BERT-base, tiny, mini, small, medium, large [41]
T5 (translation) [34]	small, base, large, 3b, 11b
GPT-2 (question answering) [33]	base, medium, large, xl

Table 3. Model families and their variants used

6.1.2 Model Variants. Our inference-serving system supports queries belonging to different query types (i.e., applications). We assume one query type corresponds to one DNN family (e.g., ResNets). Each query type can be served by different variants of the same family (e.g., ResNet-18 or ResNet-34). Table 3 shows the model families and their respective variants that we use in our experiments. We obtain the pre-trained computer vision model variants from the ONNX Model Zoo [4] and the GluonCV Model Zoo [16], and the NLP model variants from the HuggingFace repository [43]. All the models are converted to ONNX format before usage. Since our models belong to a wide variety of inference applications and their performance profiles vary significantly from one another, setting the same latency SLO for all of them is not practical. Therefore, we set latency SLOs similar to [35, 36]: for each model family, we profile its fastest model variant that can run on a CPU with the smallest batch size, and set the latency SLO of the model family to be 2× of its profiled latency. We explore the sensitivity of Proteus and the baselines to SLO in Section 6.6. Since the accuracy of each application is measured using a different metric, we normalize the accuracy of each model variant by the accuracy of the most accurate variant in its model family. This normalized accuracy varies from 80% to 100% for the variants listed in Table 3.

6.1.3 Workloads. We use a mix of real-world and synthetic workloads to evaluate our system and the baselines.

Real-world trace. Since there exists no public trace of a production inference-serving system to date, we use a public trace from Twitter [1] collected over a month-long period. As noted by previous work [35], the trace is representative of workloads that an inference-serving system would expect to see since tweets are likely to be passed through many inference pipelines before they are published. Moreover, the trace also contains diurnal patterns and spikes that an inference-serving system is likely to see over its execution. Since the queries in the trace are aggregated at the granularity of seconds, we use a Poisson process to determine inter-arrival times for queries within each second. Like prior work [35], we use a Zipf distribution to divide requests across model families with an α value of 1.001. Finally, we speed up the trace by a constant factor without modifying the shape of the trace, and we choose the constant factor such that the trace can overload our system to test its performance.

Synthetic traces. We generate synthetic traces to stress-test systems in response to burstiness, both at the macro-scale in order to evaluate resource allocation, and on the micro-scale to evaluate adaptive batching. We provide details for both in Sections 6.3 and 6.4 respectively.

6.1.4 Evaluation Metrics. We compare performance against baselines using several metrics. *(i) Throughput* is the number of queries per second served (QPS). *(ii) Effective Accuracy* is the averaged accuracy for all the queries successfully served by the system. *(iii) Maximum Accuracy Drop* for an approach represents the maximum drop in its effective accuracy over the entire trace. As the accuracy of all models is normalized as mentioned before, we measure the drop from 100%. *(iv) SLO Violation Ratio* is the number of latency SLO violations divided by the total number of queries. Since effective accuracy measures accuracy only for the queries that are successfully served by the system, it needs to be looked at in conjunction with both throughput and SLO violation ratio to get the full picture of performance.

6.1.5 Implementation. We describe our simulator-based and cluster-based implementations below.

Simulator-based Implementation. We implement Proteus in a simulator in ~6000 lines of Python code. It uses an event queue and a timer to record the arrival and processing of inference queries. We use the average profiled latencies of the model variants as the processing time for the queries. We show that the results from our simulator match closely with results from our cluster-based implementation, with a slight discrepancy mainly due to the variance in processing time of queries when they run on the actual hardware accelerators and the networking effects between the hosts. Our simulator code and workload traces are available on GitHub[5].

Cluster-based Implementation. We implement Proteus on top of an enterprise inference-serving system to test its performance and feasibility on real-world workloads and hardware accelerators. The system is used internally by the enterprise to serve inference queries for its employees who use it for a wide variety of applications and use cases, ranging from standard image classification and voice recognition models to complex pipelines for autonomous vehicles and augmented reality. Note that while we use the software infrastructure and the hardware accelerators of the enterprise in our cluster-based implementation, we do not use any proprietary models or workload traces.

Our inference-serving system consists of a cluster of 20 Intel(R) Xeon(R) Gold 6126 @ 2.60GHz CPU workers, 10 NVIDIA GeForce GTX 1080 Ti GPU workers, and 10 NVIDIA V100 GPU workers. We use Docker containers [31] to host the models in our cluster on top of Kubernetes. We extend Kubernetes with a resource optimizer plugin to implement our Resource Manager which runs on one of the CPUs. Inside the Docker containers, we run the models using the ONNX runtime [10] with the CUDA execution provider for GPUs and the CPU execution provider for CPUs.

[5]https://github.com/UMass-LIDS/Proteus

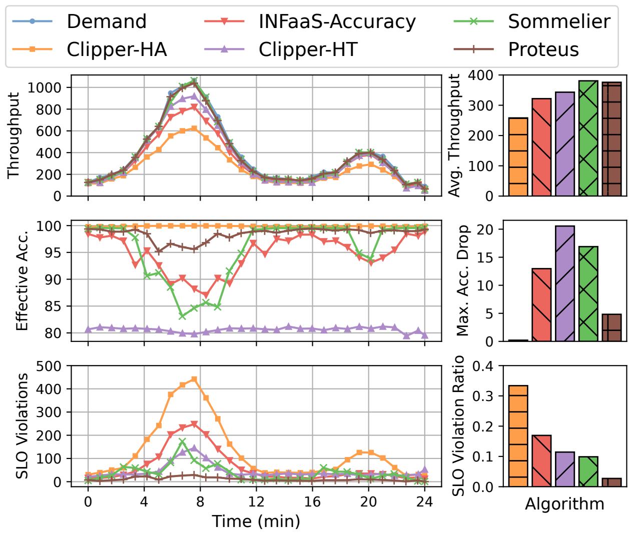


Figure 4. End-to-end performance comparison. Proteus offers the lowest accuracy drop amongst scaling approaches as well as the lowest SLO violation ratio while meeting query demand. Against the non-scaling approach, Proteus improves throughput by 60% while reducing SLO violations by 10×.

To solve our MILP in the simulator as well as the cluster-based system, we use the Python interface for Gurobi [18]. We set the invocation period of the MILP to be 30 seconds as it works well for our evaluation.

6.2 End-to-End Performance Comparison

We first show results from the end-to-end evaluation of Proteus and the baselines on our cluster testbed. Figure 4 shows the timeseries graphs for demand and throughput, effective accuracy, and the SLO violations, as well as the averaged SLO violation ratios across the trace.

We see that *Clipper* performs the worst amongst all the approaches. Even though *Clipper-HA* provides the highest effective accuracy as it never swaps out high accuracy models for low accuracy models, it suffers the highest number of SLO misses throughout the trace. Moreover, its throughput also drops significantly at the diurnal peaks during the day. For instance, during the first peak, its throughput drops to half of the query demand. On the other hand, *Clipper-HT* has significantly lower SLO violations and higher throughput than *Clipper-HA*, though it comes at the cost of the lowest effective accuracy and the maximum accuracy drop (20%).

Sommelier performs better than *Clipper* since it scales accuracy, but still suffers from a high accuracy drop (16% at peak) due to its inability to perform model placement dynamically. *INFaaS* achieves the lowest maximum accuracy drop across the baselines (13.7%) as it performs both dynamic model selection and placement. However, since it uses a greedy heuristic to do so which is prone to get stuck in local optima, it suffers from a significant drop in throughput and increase in SLO violations around peak times. INFaaS has to use a greedy heuristic because its resource allocation decision must be made on a query's arrival. Therefore, it needs a faster solution than a MILP to identify a new allocation policy in time. Proteus avoids this problem by removing MILP solvers from the critical path of serving a query.

We see that Proteus drops the least accuracy while closely meeting query demands. Its maximum accuracy drop is 4.85% at the highest peak, which is 2.8× lower than *INFaaS* and 3.2× lower than *Sommelier*. We attribute this to its optimal resource allocation obtained from the MILP. It also experiences the lowest number of SLO violations, 2.8× lower than *Sommelier*, 4.3× lower than *INFaaS*, and more than 10× lower compared to *Clipper*. We attribute this to its adaptive batching algorithm. To understand the performance of Proteus's adaptive batching further, we do a deep dive in Section 6.4.

The results from our simulator match the behavior of the cluster testbed very closely. In particular, we observe an average difference of 0.12% and 0.82% between the simulator and the cluster testbed in terms of the effective accuracy and system throughput respectively. We attribute the decrease in throughput to background traffic on the cluster, container startup delays, and other background effects not captured by the simulator. We also report an average difference of 0.5% in the SLO violation ratios. We expect this to be due to the variance in runtimes of queries from the profiled runtimes.

Therefore, we show simulation-based experiments for the remaining subsections to perform a deep dive into these results in a time and cost-efficient manner.

6.3 Bursty Workload

We now evaluate Proteus and the baselines on a synthetic *macro-scale* bursty workload to evaluate the responsiveness of resource allocation. We generate the trace by interleaving periods of flat, low query demand with periods of high query demand, having query inter-arrivals sampled through a Poisson process to introduce *macro-scale bursts.* As we are only interested in studying resource allocation in this experiment, we show timeseries plots in Figure 5 and skip the summarized SLO violation chart in the interest of space.

We note that *INFaaS* reacts the fastest to the burst in demand, due to the fact that its resource allocation lies on the critical path to inference-serving, so it can detect this change sooner and change its allocation accordingly.

Proteus first incurs a small number of SLO violations when the burst starts. The increase in query demand triggers its

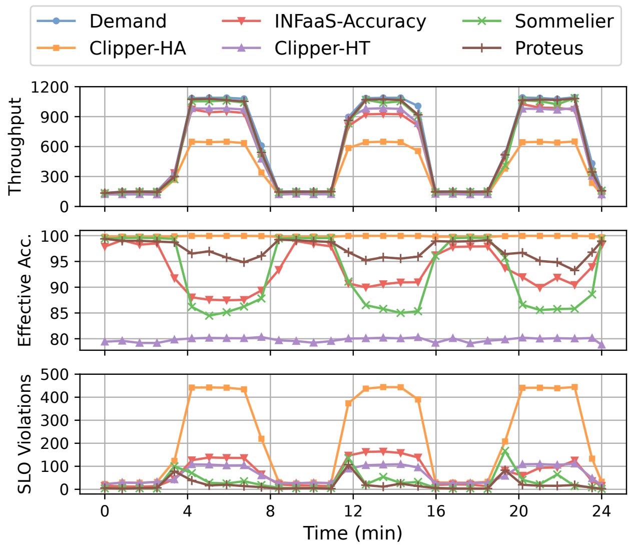


Figure 5. Responsiveness to bursty workload for Proteus vs. baselines. Proteus adjusts allocation after an initial spike in SLO violations, but provides the lowest SLO violations and accuracy drop once it re-allocates.

Resource Manager to adjust allocation, resulting in the SLO violations decreasing again as it performs accuracy scaling to meet this burst. Thus Proteus's decoupling of resource allocation from the critical path of inference-serving comes at the cost of a slightly slower response to sudden changes in workloads. Once *INFaaS* and Proteus adjust their respective allocations in response to the burst, Proteus continues to have significantly lower SLO violations and higher effective accuracy.

6.4 Adaptive Batching

Proteus's adaptive batching dynamically chooses batch sizes based on the arrival time of queries and queue status on a device to minimize SLO violations. We compare it with Clipper's AIMD batching and Nexus's [36] early-drop batching in terms of SLO Violation Ratio. To separate the effect of batching from resource allocation and query assignment, we implement each of these batching algorithms on top of Proteus in order to study them in isolation without other free variables. Since the batching of *INFaaS* is tied with its resource allocation, it cannot be evaluated separately by implementing it on top of Proteus as a module. We study its batching as part of its resource allocation as in Section 6.2.

We evaluate adaptive batching on three synthetic traces with query inter-arrivals following uniform, Poisson, and Gamma (shape 0.05) distributions. The Gamma arrival trace has highly bursty query inter-arrivals, thus introducing *micro-scale bursts.* We set the incoming load (QPS) to be the same throughout these traces as we are interested in how the batching algorithms react to query inter-arrivals, when resource allocation is unchanged. Note that burstiness in query inter-arrivals is different from the bursty workload in Figure 6.3,

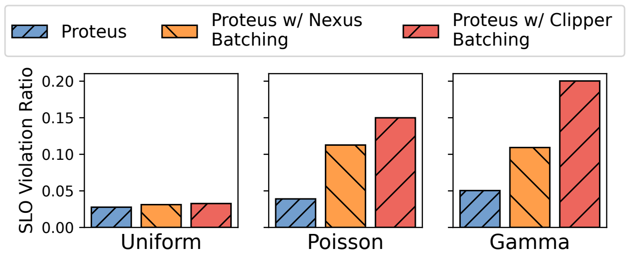


Figure 6. Comparison of Nexus and Clipper batching with Proteus. All do well on uniform traces, but Proteus reduces SLO violations by 2-4× on Poisson and Gamma traces.

where the overall demand is bursty but the query inter-arrivals have a Poisson distribution (*macro-scale bursts*).

Figure 6 shows the SLO violation ratios using three batching approaches: (i) Proteus with its original adaptive batching, (ii) Proteus with Clipper's AIMD batching, (iii) Proteus with Nexus's early-drop batching. We note that all batching algorithms perform well with uniform arrivals as the optimal batch size remains same throughout the trace, thus no adaptive adjustment is required. We see that *Nexus* experiences about 2-3× higher SLO violations on the Poisson and Gamma traces. The difference in performance between Proteus and *Nexus* batching is due to the work-conserving nature of *Nexus*. *Nexus* dispatches an inference batch to each device as soon as the previous batch is finished. As the query inter-arrivals get more and more non-uniform, it is more advantageous to wait for queries to arrive and a non-work-conserving approach works better in that case, so Proteus's adaptive batching outperforms that of *Nexus*.

On the other hand, *Clipper* experiences 3.8-4× higher SLO violations on the Poisson and Gamma traces. Though it is very easy to implement as a simple heuristic in any system, its simplicity comes at the price of poor performance. As AIMD is reactive, it does consider the current queue length nor when any of the queries currently in the queue would expire. It keeps incrementally increasing batch size until it experiences latency timeouts, at which point it backs off multiplicatively. While this works for a uniform trace where the optimal batch size does not change over time, it does not work well for Poisson or Gamma (bursty) arrivals.

6.5 Ablation Study

We present an ablation study of Proteus in Figure 7 to quantify the performance benefit of each of its components in isolation.

Proteus w/o MP (Model Placement) places the models on accelerators initially using the MILP, but does not change this placement over time. However, it can perform model selection to do accuracy scaling by changing model variants without moving them around. This baseline is the same as the *Sommelier* baseline in Section 6.2, which suffers from the largest maximum accuracy drop (16%). *Proteus w/o MS (Model Selection)* represents a baseline that performs model placement and query assignment optimally using the MILP

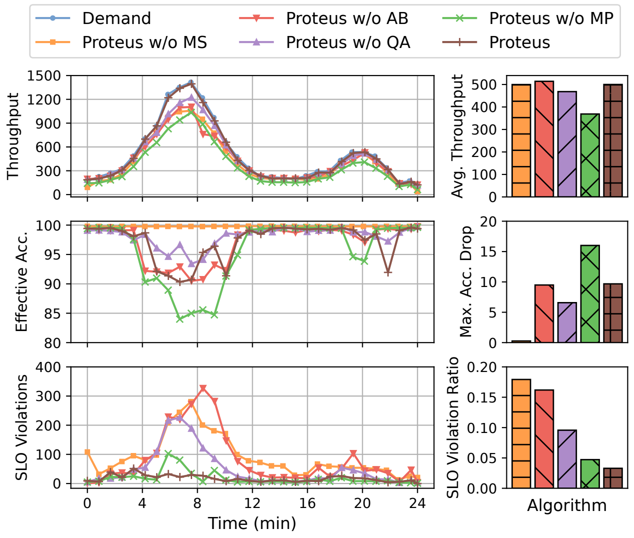


Figure 7. Proteus ablation study. Removing model selection increases SLO violations the most, while removing model placement degrades effective accuracy most.

and performs adaptive batching, but does not scale accuracy in response to workload changes. Since it does not drop any accuracy, its effective accuracy stays at 100% throughout the experiment, however, it experiences the largest overall SLO violation ratio at 0.18, due to its inability to adapt to varying demand by scaling accuracy. *Proteus w/o AB (Adaptive Batching)* represents our approach while setting batch size statically to 1. This approach has the highest absolute number of SLO violations during the peak as it is unable to batch a large number of queries together to increase throughput while meeting SLO, however this decreases when demand is low since executing with a small batch at low loads does not backlog the queues. *Proteus w/o QA (Query Assignment)* distributes queries uniformly to all devices that host the respective model variants, regardless of their serving capacity. It also suffers from a high SLO violation ratio of 0.1.

Overall, we note that with respect to SLO violations, removing model selection causes the highest degradation as the system is not able to scale accuracy to meet the increased demand (and hence, is forced to drop queries), followed by adaptive batching, query assignment, and then model placement. In terms of effective accuracy, removing model placement causes the highest degradation, followed by adaptive batching, query assignment, and then model selection.

6.6 Sensitivity to Latency SLO

We now vary the latency SLO and study its effect on all approaches, as shown in Figure 8. As mentioned before, our system hosts models from diverse tasks that have significantly different performance profiles from each other. Therefore, instead of setting the same latency SLO for all of them, we set the SLO for a query type to be a multiple of the profiled runtime of fastest model variant that runs on a CPU with a batch size of 1. We vary this multiple from 1× to 3.5× with

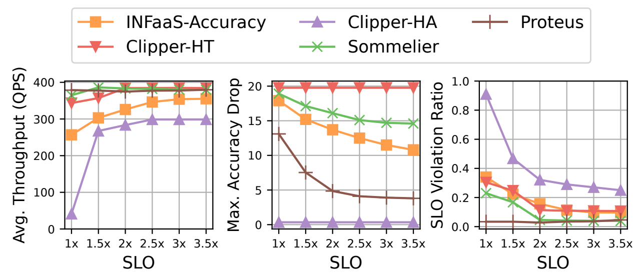


Figure 8. Effect of varying SLOs on Proteus and baselines.

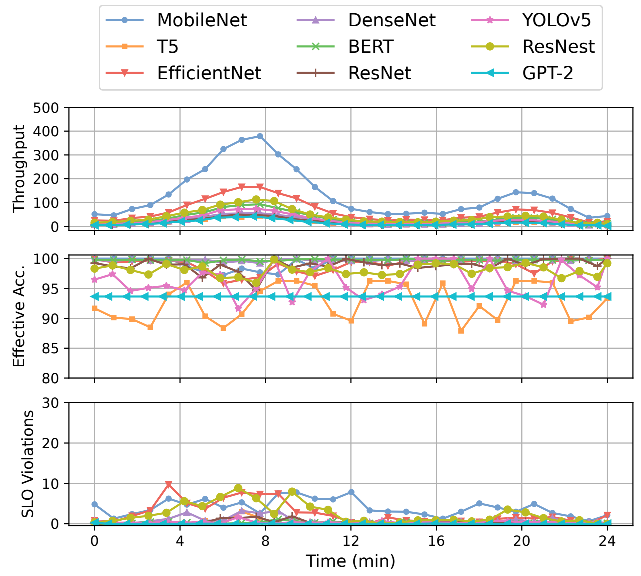


Figure 9. Breakdown of Proteus performance w.r.t. model families.

an interval of 0.5 and study its effect using the real-world Twitter trace. To show results from a large number of experiments, we show (i) averaged throughput across time, (ii) maximum accuracy drop across time, (iii) average SLO violation ratio across time.

Overall, Proteus consistently offers the lowest accuracy drop across all baselines that perform scaling, and the lowest SLO violation ratios. As the SLO increases, the SLO violation ratio decreases, and throughput increases for all baseline approaches. The maximum accuracy drop decreases for all approaches except *Clipper* as it does not dynamically scale accuracy and thus suffers from high SLO violations. Proteus can maintain the same high throughput and low SLO violation ratios across all values of SLO. Its maximum accuracy drop decreases as SLO values increase, since higher accuracy model variants with slower runtimes can meet larger SLOs.

6.7 Performance Breakdown

We now present a breakdown of the end-to-end performance of Proteus with respect to different model families on the Twitter trace. Figure 9 shows that every model family experiences different throughput as the workload has a Zipf distribution with respect to model families as described in Section 6.1. Since Proteus optimizes effective accuracy at the system level, we note variations across model families. We first observe that T5 experiences the highest variation across time, which we attribute to the fact that it has a lower incoming request rate compared to other model families, and hence carries a lower weight in the overall system effective accuracy. We also observe that GPT-2 has the lowest variation in effective accuracy since it is the most heavyweight model and cannot fit into most accelerators, hence the system loads it once in the accelerator with the highest amount of memory. In terms of SLO violations, we see relatively consistent behavior across model families, since adaptive batching is responsible for minimizing SLO violations and it executes on a per-device level. We discuss the fairness implications of these results in Section 7.

6.8 Decision Overhead

We measure the overhead of Proteus in two ways: (i) overhead of the Request Router on the critical path of queries, (ii) overhead of the Resource Manager. Since the Request Router lies on the critical path for serving every query, it is imperative that it should incur only a small delay to route the queries. Recall that the Request Router stores a routing table with an entry for each model variant as well as accelerator. We measure the latency of the Request Router to search this table to be less than 1ms. Note that there is also a space overhead of this routing table that is $O(D \times M \times Q)$ where D is the number of server devices, M is the number of model variants, and Q is the number of query types.

The overhead for solving the MILP inside the Resource Manager is also important as we need to make allocation changes quickly in order to be able to adapt to fast workload changes. We show how the runtime of the MILP scales with respect to its input parameters in Figure 10. We show the increase in the average run-time when increasing one of our input parameters (d, m, q) while keeping the other parameters constant. We also show the 95% confidence interval for each value as an error bar. As our evaluation uses an invocation period of 30 seconds for the MILP under stable conditions, we show run-times up to 60 seconds since we observe that beyond this point, our workload changes significantly and the time taken to solve the MILP makes it infeasible to adapt to workload changes quickly, forcing the system to rely on heuristic solutions. We note that the MILP can scale up to 160 devices, 450 model variants, or 17 query types while being solvable in under 60 seconds.

In our case, we measure the average time to solve our MILP to be 4.2 seconds. At this speed, the Resource Manager is able to change the resource allocation quickly without incurring any significant overhead on the system since it does not lie on the critical path to inference.

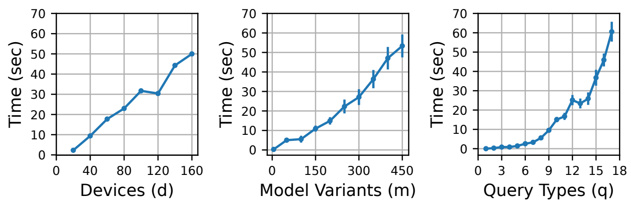


Figure 10. Scalability of MILP with respect to input parameters (d, m, q)

7 Discussion

We now discuss the limitations of our work and potential future work opportunities.

Varying Input Sizes. We consider models from a wide variety of domains in this work, such as computer vision and natural language processing. While computer vision models typically have fixed-size inputs, many natural language models can have variable-size inputs and their throughput may change depending on the size of input passed to the model for inference. While our MILP does not consider variable input sizes of queries when calculating the throughput of a model variant ($P_{d,m,q}$), adaptive batching does take into account the real-time query execution. However, for the scope of this work, we do not consider the effect of variable-size inputs on inference-serving and leave it open for future work.

Fairness. We consider accuracy scaling at a system level in this work. This can mean that despite most requests meeting the latency SLO and throughput requirements, some requests may experience better accuracy than others at times. This opens the door to fairness as an alternate objective for optimization in an inference-serving system. However, it is not straightforward to optimize fairness simultaneously with accuracy. The system can improve fairness by reallocating resources from a query type experiencing a low degree of accuracy drop to a query type experiencing a higher degree of accuracy drop; however, this would decrease overall effective accuracy at the system level. Thus, there is a trade-off between fairness and system-wide effective accuracy, which remains an interesting direction to explore for future work.

Hardware Scaling. In this work, we present accuracy scaling as an alternative to hardware scaling for resource-constrained inference-serving systems. However, accuracy scaling can also work in tandem with hardware scaling for systems that have the ability to add hardware resources. Since hardware scaling involves provisioning and starting new servers which takes time, accuracy scaling can be used on a finer-grained timescale to absorb sudden unexpected bursts of load by dropping accuracy for a short time while waiting for new servers to be allocated.

8 Related Work

There has been much work recently on inference-serving systems. Representative production systems include TensorFlow-Serving [32], Amazon SageMaker [2], Triton Inference Server [5], and Azure ML [3]. Research prototypes include Clipper [9], PRETZEL [27], Clockwork [14], DLIS [37], and INFless [45]. They aim to provide a unified abstraction to the user to hide details of the underlying ML frameworks, data pre-processing, and performance optimization. Cocktail [15] uses model ensembling to improve accuracy and meet latency requirements at minimal cost. Some other work focuses specifically on serving heterogeneous DNNs via GPU spatial sharing [7, 12], dynamic model placement [38], computation merging [24], and adaptive scheduling [30]. However, none of these works perform accuracy scaling to improve system throughput for resource-constrained inference-serving.

The closest works to Proteus are INFaaS [35, 44], Sommelier [17], and Tolerance Tiers [19]. INFaaS [35] presents a model-less inference-serving system that leverages model graph optimizers to generate model variants and automates selection of model variants for each query to minimize cost. However, it assumes that all model variants generated can meet the accuracy requirement and thus fails to simultaneously optimize for effective accuracy. INFaaS also suffers from local optimum in model-to-device assignment due to its heuristic assignment algorithm. Sommelier [17] is a model repository that can interface with inference-serving systems to suggest model variants with lower accuracy to handle periods of high load. Tolerance Tiers [19] allows developers to trade accuracy off for latency using programming APIs, but restricts an application to use only one accuracy tier statically throughout the inference-serving process instead of adapting accuracy dynamically as a scaling approach. Further, it doesn't consider the scenario of serving heterogeneous DNNs on heterogeneous devices.

9 Conclusion

We presented Proteus, a high-throughput inference-serving system that leverages accuracy scaling to handle query workload variations. We formulated the resource management in Proteus as a mixed integer linear programming optimization to adapt to macro-scale variations, and solved it to guarantee the optimal solution. We also proposed a novel adaptive batching algorithm to improve throughput and absorb micro-scale variations. We evaluated Proteus on real-world workload traces and showed that it outperforms state-of-the-art baselines in reducing SLO violations and improving system throughput while dropping minimal accuracy.

Acknowledgments

We thank our shepherd, Peter Pietzuch, and the anonymous reviewers for their helpful feedback. This material is based upon work supported by the National Science Foundation under grants 1763617, 2106299, 2312396, 2220211, and 2224054. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

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

[2] 2020. Amazon SageMaker. Build, train, and deploy machine learning models at scale. https://aws.amazon.com/sagemaker/. Accessed: 2021-06-23.

[3] 2022. Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning/.

[4] 2022. The ONNX Model Zoo. https://github.com/onnx/models. Accessed: 2022-06-06.

[5] 2022. Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.

[6] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the State of Neural Network Pruning?. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 129–146. https://proceedings.mlsys.org/paper_files/paper/2020/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2021. Multi-model Machine Learning Inference Serving with GPU Spatial Partitioning. *CoRR* abs/2109.01611 (2021). arXiv:2109.01611 https://arxiv.org/abs/2109.01611

[8] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. 2021. Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 493–506. https://doi.org/10.1109/HPCA51647.2021.00049

[9] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[10] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/. Version: 1.8.1.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[12] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 492–506. https://doi.org/10.1145/3419111.3421284

[13] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (New Delhi, India) *(MobiCom '18)*. Association for Computing Machinery, New York, NY, USA, 115–127. https://doi.org/10.1145/3241539.3241559

[14] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[15] 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

[16] Jian Guo, He He, Tong He, Leonard Lausen, Mu Li, Haibin Lin, Xingjian Shi, Chenguang Wang, Junyuan Xie, Sheng Zha, et al. 2020. GluonCV and GluonNLP: deep learning in computer vision and natural language processing. *J. Mach. Learn. Res.* 21, 23 (2020), 1–7.

[17] Peizhen Guo, Bo Hu, and Wenjun Hu. 2022. Sommelier: Curating DNN Models for the Masses. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1876–1890. https://doi.org/10.1145/3514221.3526173

[18] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[19] Matthew Halpern, Behzad Boroujerdian, Todd W. Mummert, Evelyn Duesterwald, and Vijay Janapa Reddi. 2019. One Size Does Not Fit All: Quantifying and Exposing the Accuracy-Latency Trade-off in Machine Learning Cloud Service APIs via Tolerance Tiers. *CoRR* abs/1906.11307 (2019). arXiv:1906.11307 http://arxiv.org/abs/1906.11307

[20] 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

[21] 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. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[22] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4700–4708.

[23] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. 2015. TencentRec: Real-Time Stream Recommendation in Practice *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/2723372.2742785

[24] Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, Yunseong Lee, and Byung-Gon Chun. 2020. Accelerating Multi-Model Inference by Merging DNNs of Different Weights. *arXiv preprint arXiv:2009.13062* (2020).

[25] Glenn Jocher, Ayush Chaurasia, Alex Stoken, Jirka Borovec, NanoCode012, Yonghye Kwon, Kalen Michael, TaoXie, Jiacong Fang, imyhxy, Lorna, Zeng Yifu, Colin Wong, Abhiram V, Diego Montes, Zhiqiang Wang, Cristi Fati, Jebastin Nadar, Laughing, UnglvKitDe, Victor Sonck, tkianai, yxNONG, Piotr Skalski, Adam Hogan, Dhruv Nair, Max Strobel, and Mrinal Jain. 2022. *ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation*. https://doi.org/10.5281/zenodo.7347926

[26] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. *CoRR* abs/1909.11942 (2019). arXiv:1909.11942 http://arxiv.org/abs/1909.11942

[27] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 611–626. https://www.usenix.org/conference/osdi18/presentation/lee

[28] Matthew LeMay, Shijian Li, and Tian Guo. 2020. PERSEUS: Characterizing Performance and Cost of Multi-Tenant Serving for CNN Models. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*. 66–72. https://doi.org/10.1109/IC2E48712.2020.00014

[29] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[30] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: Towards High-Performance Multi-Tenant Deep Learning Services via Adaptive Compilation and Scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 388–401. https://doi.org/10.1145/3503222.3507752

[31] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[32] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Workshop on ML Systems at NIPS 2017*.

[33] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[35] 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)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[36] 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

[37] Jonathan Soifer, Jason Li, Mingqin Li, Jeffrey Zhu, Yingnan Li, Yuxiong He, Elton Zheng, Adi Oltean, Maya Mosyak, Chris Barnes, Thomas Liu, and Junhua Wang. 2019. Deep Learning Inference Service at Microsoft. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. USENIX Association, Santa Clara, CA, 15–17. https://www.usenix.org/conference/opml19/presentation/soifer

[38] Piyush Subedi, Jianwei Hao, In Kee Kim, and Lakshmish Ramaswamy. 2021. AI Multi-Tenancy on Edge: Concurrent Deep Learning Model Executions and Dynamic Model Placements on Edge Devices. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 31–42. https://doi.org/10.1109/CLOUD53861.2021.00016

[39] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[40] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

[41] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-Read Students Learn Better: The Impact of Student Initialization on Knowledge Distillation. *CoRR* abs/1908.08962 (2019). arXiv:1908.08962 http://arxiv.org/abs/1908.08962

[42] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 717–729. https://doi.org/10.1145/3445814.3446763

[43] 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. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[44] Neeraja J. Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. 2019. A Case for Managed and Model-Less Inference Serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 184–191. https://doi.org/10.1145/3317550.3321443

[45] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 768–781. https://doi.org/10.1145/3503222.3507709

[46] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex Bronstein, Ronald Dreslinski, Trevor Mudge, and Nishil Talati. 2023. GRACE: A Scalable Graph-Based Approach to Accelerating Recommendation Model Inference *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 282–301. https://doi.org/10.1145/3582016.3582029

[47] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Haibin Lin, Zhi Zhang, Yue Sun, Tong He, Jonas Mueller, R. Manmatha, Mu Li, and Alexander Smola. 2022. ResNeSt: Split-Attention Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 2736–2746.

A Artifact Appendix for Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling

A.1 Abstract

This artifact describes the complete workflow to setup the simulation experiments for Proteus. We describe how to obtain the code, and then describe two methods to install the simulator. We explain how to run the experiments and the expected results from simulation. Finally, we also publicize all the workload traces used in our paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Combinatorial optimization using mixed integer linear programming, adaptive batching.
- **Hardware:** Docker container for `linux_amd64` platform provided. Source code can be used on any hardware.
- **Metrics:** Throughput, inference accuracy, latency SLO violations.
- **Output:** Log files are output by the simulator which are then used by the plotting scripts to generate graphs for results.
- **Experiments:** End-to-end evaluation of Proteus and baselines as well as an evaluation of the responsiveness of Proteus against baselines.
- **How much disk space required (approximately)?:** Docker container requires approximately 1.5GB of disk space.
- **How much time is needed to prepare workflow (approximately)?:** 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1-2 hours depending on hardware platform.
- **Publicly available?:** Yes. See below for access details.

A.3 Description

A.3.1 How to access. The simulator code and workload traces can be accessed at https://github.com/UMass-LIDS/Proteus. It is also accessible at the following DOI: https://doi.org/10.5281/zenodo.10428550.

A.3.2 Hardware dependencies. None for the simulator.

A.3.3 Software dependencies. We provide two installation methods. The first method only requires Docker, while the second method requires the following:

- Gurobi optimization software
- A Gurobi license
- `conda`
- A local installation of Python

A.3.4 Data sets. The artifact has two types of datasets:

1. A *real-world* Twitter workload dataset to test end-to-end system performance.
2. Several *synthetic* workload datasets to test performance of isolated components as described in the paper

A.3.5 Models. A mix of computer vision models for image classification and object detection is used, as well as natural language models for text translation, sentiment analysis, and question answering. The complete list of all the DNN models and their variants used is given in Table 3 in the paper.

A.4 Installation

We provide two methods of installation:

1. **Docker:** In order to quickly evaluate Proteus, we provide a Docker container with usage instructions at https://github.com/UMass-LIDS/Proteus/blob/main/DOCKER.md. We recommend this method for artifact evaluation and quick testing as it does not require obtaining a Gurobi license. For any other use cases, please use the second method described below.
2. **Local installation:** For extensive evaluation and general simulator usage, we recommend locally installing Proteus and obtaining a Gurobi license. The instructions are provided at https://github.com/UMass-LIDS/Proteus/blob/main/README.md.

A.5 Experiment workflow

The simulator requires a JSON configuration file as input to set up the experiment. We have provided several example configuration files in the `config` folder of our GitHub repo. A configuration file describes the workload trace to use, the resource allocation algorithm, the adaptive batching algorithm, as well as any hyper-parameters (e.g., a β value of 1.05 for Proteus is used by default).

Depending on the installation method used, the experiments can be run using either of the following set of instructions:

1. **Docker:** Follow instructions at https://github.com/UMass-LIDS/Proteus/blob/main/DOCKER.md.
2. **Local installation:** Follow instructions at https://github.com/UMass-LIDS/Proteus/blob/main/EXAMPLES.md.

A.6 Evaluation and expected results

The simulator produces log files in the `logs` folder. These log files contain snapshots of the system at regular intervals containing not only aggregated information about user demand, system capacity, requests served/dropped/late, and accuracy seen by the requests, but they also contain detailed logs for all system events in `logs/per_predictor`.

These log files are then ingested by the plotting scripts provided in the `plotting` folder to generate two graphs: an end-to-end evaluation of Proteus against the baselines on the Twitter trace, similar to the one in Section 6.3, as well as an evaluation of the responsiveness of Proteus vs. the baselines on a bursty trace, as in Section 6.4.

A.7 Experiment customization

The experiments can be customized by editing the sample configuration files in the `config` folder.

The following resource allocation algorithms can be used in the `model_allocation` field: `ilp` for Proteus, `infaas_v2` for INFaaS-Accuracy, `clipper` for Clipper, and `sommelier` for Sommelier.

The following batching algorithms can be used in the `batching` field: `accscale` for Proteus adaptive batching, `aimd` for Clipper's AIMD batching, and `nexus` for Nexus's batching algorithm.

A.8 Notes

Please refer to https://github.com/UMass-LIDS/Proteus for complete reference and instructions.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html