InferLine: Prediction Pipeline Provisioning and Management for Tight Latency Objectives

Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo
Joseph E. Gonzalez, Ion Stoica, Alexey Tumanov
UC Berkeley RISELab

10-06-2019

Abstract

Serving prediction pipelines spanning multiple models and hardware accelerators is a key challenge in production machine learning. Optimally configuring these pipelines to meet tight end-to-end latency goals is complicated by the interaction between model batch size, the choice of hardware accelerator, and variation in the query arrival process.

In this paper we introduce InferLine, a system which provisions and executes ML prediction pipelines subject to end-to-end latency constraints by proactively optimizing and reactively controlling per-model configurations in a fine-grained fashion. InferLine leverages automated offline profiling and performance characterization to construct a cost-minimizing initial configuration and then introduces a reactive planner to adjust the configuration in response to changes in the query arrival process. We demonstrate that InferLine outperforms existing approaches by up to 7.6x in cost while achieving up to 34.5x lower latency SLO miss rate on realistic workloads and generalizes across state-of-the-art model serving frameworks.

1 Introduction

Serving predictions from trained machine learning models is emerging as a dominant challenge in production machine learning. Increasingly, pipelines of models and data transformations [34, 5] spanning multiple hardware accelerators are being deployed to deliver interactive services (e.g., speech recognition [9], content filtering [44], machine translation [1]). These computationally intensive prediction pipelines must run continuously with a tight latency budget and in response to stochastic and often bursty query arrival processes.

This paper addresses the problem of how to optimally allocate and configure complex prediction pipelines with tight end-to-end latency constraints at low cost under bursty and unpredictable workloads. Addressing this problem requires addressing two key challenges.

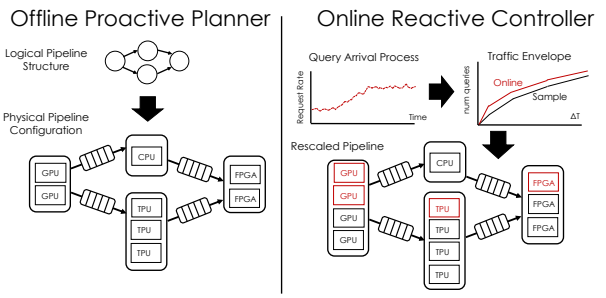


Figure 1: InferLine System Overview

Complex Configuration Space. The optimal configuration for a model requires choosing a hardware accelerator (e.g., CPU, GPUs, and TPUs) and query batch size to balance latency against throughput and cost. Larger batch sizes can improve throughput and reduce the number of costly hardware accelerators needed for a given workload but come at the cost of increased query latency. These configuration decisions are complicated by the combinatorial interaction between each stage in a prediction pipeline and the need to meet end-to-end latency service level objectives (SLOs).

Workload Dependence. Variation in the query arrival process can change the cost optimal configuration for the end-to-end pipeline. A system that has been aggressively tuned to minimize cost under the assumption of a uniform arrival process can destabilize under realistic bursty workloads (as in Fig. 13(a)) resulting in missed latency deadlines. Moreover, previously cost-optimal configurations become under- or over-provisioned as workloads evolve over time (as in Fig. 9).

In this paper we propose InferLine, a high-performance system for provisioning and serving heterogeneous prediction pipelines with strict end-to-end latency constraints (see Fig. 1). InferLine combines a proactive planner that explores the full configuration space *offline* to minimize cost with a reactive scaling controller that reconfigures the

pipeline's replication factors *online* to maintain latency SLOs in response to changing workloads. While a reactive system can react to rapid changes in workloads to ensure SLOs, it cannot explore the combinatorial configuration space efficiently to minimize costs. Conversely, an offline proactive system can afford to explore the configuration space to find an optimal configuration for a given workload, but cannot afford to re-run the offline planner and migrate the pipeline to a new configuration in response to rapid workload changes.

One key enabling assumption in InferLine is that the complex performance characteristics of individual models can be accurately profiled offline using readily available training data and composed to accurately estimate end-to-end system performance. InferLine profiles each stage in the pipeline individually and uses these profiles and a discrete event simulator to accurately estimate end-to-end pipeline latency given hardware and batching configuration parameters. The offline planner uses a hill-climbing algorithm to find the cost-minimizing pipeline configuration while using the simulator to ensure that it only considers configurations that meet the latency objective. The pipeline is then served with InferLine's physical execution engine, utilizing InferLine's centralized batched queuing system to ensure predictable queuing behavior. The online reactive controller leverages traffic envelopes from network calculus to capture the arrival process dynamics across multiple time scales and determine when and how to react to changes. As a consequence, the reactive controller is able to maintain the latency SLO in the presence of transient spikes and sustained variation in the query arrival process.

In summary, the primary contribution of this paper is the hybrid proactive-reactive architecture for cost-efficient provisioning and serving of prediction pipelines with end-to-end latency constraints. This architecture builds on three technical contributions:

1. An accurate end-to-end latency estimation procedure for prediction pipelines spanning multiple hardware and model configurations.

2. A proactive pipeline optimizer that minimizes hardware costs for a given arrival workload and end-to-end latency constraints.

3. An online reactive controller algorithm that monitors and scales each stage in the pipeline to maintain high SLO attainment at low cost.

We evaluate InferLine across a range of pipelines using real ML models subjected to query traces spanning multiple arrival distributions. We compare InferLine to the state-of-the-art model serving baselines that (a) use coarse-grain proactive configuration of the whole pipeline as a unit (e.g., TensorFlow Serving [39], and (b) state-of-the-art coarse-grain reactive mechanisms [14]. We find that InferLine significantly outperforms the baselines by a factor of up to 7.6X on cost, while exceeding two nines of SLO attainment—the highest level of latency SLO attainment achieved in our experiments.

2 Background and Motivation

Prediction pipelines combine multiple machine learning models and data transformations to support complex prediction tasks [35]. For instance, state-of-the-art visual question answering services [2, 26] combine language models with vision models to answer the question.

A prediction pipeline can be formally encoded as a directed acyclic graph (DAG) where each vertex corresponds to a model (e.g., a mapping from images to a list of objects in the image) or some other more basic data transformation (e.g., extracting key frames from a video) and each edge represents a data flow. In this paper we study several (Figure 2) representative prediction pipeline motifs.

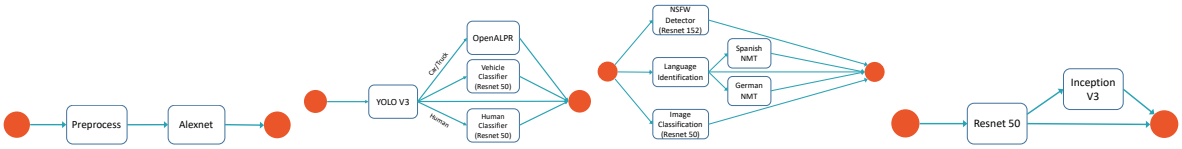
The Image Processing pipeline consists of basic image pre-processing (e.g., cropping and resizing) followed by image classification using a deep neural network. The Video Monitoring pipeline was inspired by [43] and uses an object detection model to identify vehicles and people and then performs subsequent analysis including vehicle and person identification and license plate extraction on any relevant images. The Social Media pipeline translates and categorizes posts based on both text and linked images by combining computer vision models with multiple stages of language model to identify the source language and translate the post if necessary. Finally, the TF Cascade pipeline combines fast and slow models, invoking the slow model only when necessary.

In the Social Media, Video Monitoring, and TF Cascade pipelines, a subset of models are invoked based on the output of earlier models in the pipeline. This conditional evaluation pattern appears in bandit algorithms [23, 4] used for model personalization as well as more general cascaded prediction pipelines [27, 16, 3, 37].

In this work, we demonstrate that InferLine is able to maintain latency constraints with P99 service level objectives (99% of query latencies must be below the constraint) at low cost, even under bursty and unpredictable workloads.

2.1 Systems Challenges

Prediction pipelines present new challenges for the design and provisioning of inference serving systems. We first discuss how the proliferation of specialized hard-



(a) Image Processing (b) Video Monitoring (c) Social Media (d) TF Cascade

Figure 2: Example Pipelines. We evaluate InferLine on four prediction pipelines that span a wide range of models, control flow, and input characteristics.

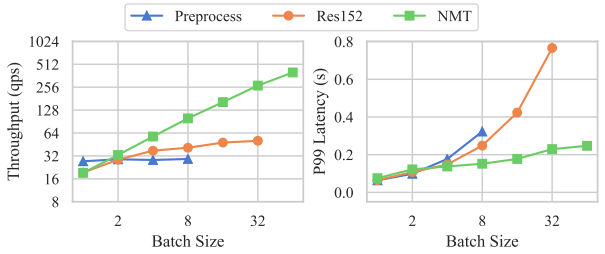


Figure 3: Example Model Profiles on K80 GPU. The preprocess model has no internal parallelism and cannot utilize a GPU. Thus, it sees no benefit from batching. Res152 (image classification) & TF-NMT(text translation model) benefit from batching on a GPU but at the cost of increased latency.

ware accelerators and the need to meet end-to-end latency constraints leads to a combinatorially large configuration space. We then discuss some of the complexities of meeting tight latency SLOs under bursty stochastic query loads. Finally, we contrast this work with ideas from the data stream processing literature, which shares some structural similarities but is targeted at fundamentally different applications and performance goals.

Combinatorial Configuration Space Many machine learning models can be computationally intensive with substantial opportunities for parallelism. In some cases, this parallelism can result in orders of magnitude improvements in throughput and latency. For example, in our experiments we found that TensorFlow can render predictions for the relatively large ResNet152 neural network at 0.6 queries per second (QPS) on a CPU and at 50.6 QPS on an NVIDIA Tesla K80 GPU, an 84x difference in throughput (see Fig. 3). However, not all models benefit equally from hardware accelerators. For example, several widely used classical models (e.g., decision trees [8]) can be difficult to parallelize on GPUs, and often common data transformations (e.g. text feature extraction) cannot be efficiently computed on GPUs.

In many cases, to fully utilize the available parallel hardware, queries must be processed in batches (e.g., ResNet152 required a batch size of 32 to maximize throughput on the K80). However, processing queries in a batch can also increase latency, as we see in Fig. 3. Because most hardware accelerators operate at vector level parallelism, the first query in a batch is not returned until the last query is completed. As a consequence, it is often necessary to set a *maximum* batch size to bound query latency. However, the choice of the maximum batch size depends on the hardware and model and affects the end-to-end latency of the pipeline.

Finally, in heavy query load settings it is often necessary to replicate individual operators in the pipeline to provide the throughput demanded by the workload. As we scale up pipelines through replication, each operator scales differently, an effect that can be amplified by the use of conditional control flow within a pipeline causing some components to be queried more frequently than others. Low cost configurations require fine-grained scaling of each operator.

Allocating parallel hardware resources to a single model presents a complex model dependent trade-off space between cost, throughput, and latency. This trade-off space grows exponentially with each model in a prediction pipeline. Decisions made about the choice of hardware, batching parameters, and replication factor at one stage of the pipeline affect the set of feasible choices at the other stages due to the need to meet *end-to-end* latency constraints. For example, trading latency for increased throughput on one model by increasing the batch size reduces the latency budget of other models in the pipeline and as a consequence the feasible hardware configurations.

Queuing Delays Because prediction pipelines span multiple hardware devices that run at different speeds and batch sizes, buffering in the form of queues is needed between stages. However, queuing adds to the latency of the system. Therefore queuing delays, which depend on the query arrival process and system configuration, must be considered during provisioning.

Stochastic and Unpredictable Workloads Prediction serving systems must respond to bursty, stochastic query streams. At a high-level these stochastic processes can be characterized by their average arrival rate λ and their coefficient of variation, a dimensionless measure of variability defined by $CV = \frac{\sigma^2}{\mu^2}$, where $\mu = \frac{1}{\lambda}$ and σ are the mean and standard-deviation of the query inter-arrival time. Processes with higher CV have higher variability and often require additional over-provisioning to meet query latency

objectives. However, over-provisioning an entire pipeline on specialized hardware can be prohibitively expensive. Therefore, it is critical to be able to identify and provision the bottlenecks in a pipeline to accommodate the bursty arrival process. Finally, as the workload changes, we need mechanisms to quickly detect and re-provision individual stages in the pipeline.

Comparison to Stream Processing Systems Many of the challenges around configuring and scaling pipelines have been studied in the context of generic data stream processing systems [11, 33, 40, 36]. However, these systems focus their effort on supporting more traditional data processing workloads, which include stateful aggregation operators and support for a variety of windowing operations. As a result, the concept of per-query latency is often ill-defined in data pipelines, and instead these systems tend to focus on maximizing throughput while avoiding backpressure, with latency as a second order performance goal (see §8).

3 System Design and Architecture

In this section, we provide a high-level overview of the main system components in InferLine (Fig. 1). The system can be decomposed into *offline planning* where a cost-efficient initial pipeline configuration is selected (§4), and *online serving* where the pipeline is hosted for serving live production traffic and reactively scaled to maintain latency SLOs and remain cost-efficient under changing workloads (§5).

To deploy a new prediction pipeline in InferLine, developers provide a driver program, sample query trace used for planning, and a latency service level objective. The driver function interleaves application-specific code with asynchronous RPC calls to models hosted in InferLine. The sample trace is specified as a list of queries and inter-arrival times, providing information about the distribution of query content and pipeline behavior, as well as expected arrival process dynamics.

Offline Planning: Offline planning begins by extracting the pipeline's dependency structure. Next, the **Profiler** creates performance profiles of all the individual models referenced by the driver program. A performance profile captures model throughput as a function of hardware type and maximum batch size. An entry in the model profile is measured empirically by evaluating the model in isolation in the given configuration using the queries in the sample trace. The last step in offline planning is pipeline configuration, which finds a cost-efficient initial pipeline configuration subject to the end-to-end latency SLO and the specified arrival process. The offline planner sets the three control parameters for each model in the pipeline using a globally-aware, cost-minimizing optimization algorithm. The **Planner** uses the model profiles extracted by the **Profiler** to select cost-minimizing steps in each iteration while relying on the **Estimator** to check for latency constraint violations.

Online Serving: Once planning is complete, the pipeline is deployed to the **Physical Execution Engine** with the planned configuration for live production serving. The serving engine adopts a distributed microservice architecture similar to [10, 39]. In addition, the engine interposes a latency-aware batched queuing system to tightly control how queries are distributed among model replicas. The queuing system both minimizes queue waiting time and ensures that the **Estimator** can accurately simulate the queuing process.

The **Reactive Controller** monitors the dynamic behavior of the arrival process to adjust per-model replication factors and maintain high SLO attainment at low cost. The reactive controller continuously monitors the current traffic envelope [22] to detect deviations from the sample trace traffic envelope at different timescales simultaneously. By analyzing the timescale at which the deviation occurred, the reactive controller is able to take appropriate mitigating action to ensure that SLOs are met without unnecessarily increasing cost.

3.1 Execution Model

Similar to [10, 18], the **Physical Execution Engine** adopts a distributed microservice architecture, allowing models to dictate their environment dependencies and enabling the serving system to be distributed across a cluster of heterogeneous hardware.

To enable the estimator to accurately simulate the system's queuing behavior, the **Physical Execution Engine** interposes a deadline-aware batched queuing system to tightly control how queries are distributed among model replicas. All requests to a model are placed in a centralized queue from which all replicas for that model pull. To ensure that InferLine is always processing the queries that will expire first, InferLine uses earliest deadline first (EDF) priority queues.

When a model replica is ready to process a new batch of inputs, it requests a batch from the centralized queue for that model. The size of the batch is bounded above by the maximum *batch size* for the model as configured by the **Planner**. By employing a pull-based queuing strategy and imposing a maximum batch size, InferLine places an upper bound on the time that a query will spend in the model container itself after leaving the queue.

4 Offline Planning

During offline planning, the **Profiler**, **Estimator** and **Planner** are used to estimate model performance characteristics and optimally provision and configure the system for a given sample workload and latency SLO. In this section, we expand on each of these three components.

4.1 Profiler

The **Profiler** takes a pipeline driver and sample query trace and extracts the logical pipeline structure. It then creates performance profiles for each of the models in the pipeline as a function of batch size and hardware.

Profiling begins with InferLine executing the sample set of queries on the pipeline. This generates input data for profiling each of the component models individually. We also track the frequency of queries visiting each model. This frequency represents the conditional probability that a model will be queried given a query entering the pipeline, independent of the behavior of any other models. We refer to this frequency as the *scale factor*, s, of the model. The scale factor is used by the **Estimator** to estimate the effects of conditional control flow on latency (§4.2) and the **Reactive Controller** to make scaling decisions (§5.1).

The profiler captures model throughput as a function of hardware type and maximum batch size to create per-model performance profiles. An individual model configuration corresponds to a specific value for each of these parameters as well as the model's replication factor. Because the models scale horizontally, profiling a single replica is sufficient.

4.2 Estimator

The **Estimator** is responsible for rapidly estimating the end-to-end latency of a given pipeline configuration for the sample query trace. It takes as input a pipeline configuration, the individual model profiles, and a sample trace of the query workload, and returns accurate estimates of the latency for *each query* in the trace. The **Estimator** is implemented as a continuous-time, discrete-event simulator [6], simulating the entire pipeline, including queuing delays. The simulator maintains a global logical clock that is advanced from one discrete event to the next with each event triggering future events that are processed in temporal order. Because the simulation only models discrete events, we are able to faithfully simulate hours worth of real-world traces in hundreds of milliseconds.

The estimator leverages the deterministic behavior of the batched queuing system to accurately simulate queuing behavior. It combines this with the model profile information which informs the simulator how long a model under a specific hardware configuration will take to process a batch.

Algorithm 1: Find an initial, feasible configuration

```
1  Function Initialize(pipeline, slo):
2      foreach model in pipeline do
3          model.batchsize = 1;
4          model.replicas = 1;
5          model.hw = BestHardware(model);
6      if ServiceTime(pipeline) ≤ slo then
7          return False;
8      else
9          while not Feasible(pipeline, slo) do
10             model = FindMinThru(pipeline);
11             model.replicas += 1;
12         return pipeline;
```

4.3 Planning Algorithm

At a high-level, the offline planning algorithm is an iterative constrained optimization procedure that greedily minimizes cost while ensuring that the latency constraint is satisfied. The algorithm can be divided into two phases. In the first (Algorithm 1), it finds a feasible initial configuration that meets the latency SLO while ignoring cost. In the second (Algorithm 2), the planner greedily modifies the configuration to reduce the cost while using the **Estimator** to identify and reject configurations that violate the latency SLO. The algorithm converges when it can no longer make any cost reducing modifications to the configuration without violating the SLO.

Initialization (Algorithm 1): First, an initial latency-minimizing configuration is constructed by setting the batch size to 1 using the lowest latency hardware available for each model (lines 2-5). If the service time under this configuration (the sum of the processing latencies of all the models on the longest path through the pipeline DAG) is greater than the SLO then the latency constraint is infeasible given the available hardware and the planner terminates (lines 6-7). The **Planner** then iteratively determines the *throughput bottleneck* and increases that model's replication factor until it is no longer the bottleneck (lines 9-11).

Cost-Minimization (Algorithm 2): In each iteration of the cost-minimizing process, the **Planner** considers three candidate modifications for each model: increase the batch size, decrease the replication factor, or downgrade the hardware (line 5), searching for the modification that

Algorithm 2: Find the min-cost configuration

```
1  Function MinimizeCost(pipeline, slo):
2      pipeline = Initialize(pipeline, slo);
3      if pipeline == False then
4          return False;
5      actions = [IncreaseBatch,
         RemoveReplica, DowngradeHW ];
6      repeat
7          best = NULL;
8          foreach model in pipeline do
9              foreach action in actions do
10                 new = action(model, pipeline);
11                 if Feasible(new) then
12                     if new.cost < best.cost then
13                         best = new;
14         if best is not NULL then
15             pipeline = best;
16     until best == NULL;
17     return pipeline;
```

maximally decreases cost while still meeting the latency SLO. It evaluates each modification on each model in the pipeline (lines 8-10), discarding candidates that violate the latency SLO according to the **Estimator** (line 11).

The **batch size** only affects throughput and does not affect cost. It will therefore only be the cost-minimizing modification if the other two would create infeasible configurations. Increasing the batch size does increase latency. The batch size is increased by factors of two as the throughput improvements from larger batch sizes have diminishing returns (observe Fig. 3). In contrast, decreasing the **replication factor** directly reduces cost. Removing replicas is feasible when a previous iteration of the algorithm has increased the batch size for a model, increasing the per-replica throughput.

Downgrading hardware is more involved than the other two actions, as the batch size and replication factor for the model must be re-evaluated to account for the differing batching behavior of the new hardware. It is often necessary to reduce the batch size and increase replication factor to find a feasible pipeline configuration. However, the reduction in hardware price sometimes compensates for the increased replication factor. For example, in Fig. 8, the steep decrease in cost when moving from an SLO of 0.1 to 0.15 can be largely attributed to downgrading the resource allocation of a language identification model from a GPU to a CPU.

To evaluate a hardware downgrade, we first freeze the configurations of the other models in the pipeline and perform the initialization stage for that model using the next cheapest hardware. The planner then performs a localized version of the cost-minimizing algorithm to find the batch size and replication factor for the model on the newly downgraded resource allocation needed to reduce the cost of the previous configuration. If there is no cost reducing feasible configuration the hardware downgrade action is rejected.

At the point of termination, the planner provides the following guarantees: (1) If there is a configuration that meets the latency SLO, then the planner will return a valid configuration. (2) There is no single action that can be taken to reduce cost without violating the SLO.

5 Online Serving

During online serving, the **Physical Execution Engine** provisions model container resources according to the configuration produced by the **Planner** and hosts pipelines for serving live production traffic. The **Reactive Controller** monitors and detects unexpected changes in the arrival process and reactively adjusts the configuration to maintain the latency SLO.

5.1 Reactive Controller

InferLine's offline planner finds an efficient, low-cost configuration that is guaranteed to meet the provided latency objective. However, this guarantee only holds for the sample workload provided during offline planning. Real workloads evolve over time, changing in both arrival rate (change in λ) as well as becoming more or less bursty (change in CV). When the serving workload deviates from the sample, the original configuration will either suffer from queue buildups leading to SLO misses or be over-provisioned and incur unnecessary costs. The **Reactive Controller** both *detects* these changes as they occur and takes the appropriate *scaling action* to maintain both the latency constraint and cost-efficiency of objective.

In order to maintain P99 latency SLOs, the reactive controller must be able to detect changes in the arrival workload dynamics across multiple timescales simultaneously. The proactive planner guarantees that the pipeline is adequately provisioned for the sample trace. The reactive controller's detection mechanism detects when the current request workload exceeds the sample workload. To do this, we draw on the idea of traffic envelopes from network calculus [22] to characterize the workloads.

To create a traffic envelope for a workload, we measure the maximum arrival rate for several different sized windows. For a given window size ΔT_i measured in seconds, we measure q_i as the maximum number of queries that arrived within ΔT_i seconds of each other at any point in the workload. The maximum arrival rate is then sim-

ply $r_i = \frac{q_i}{\Delta T_i}$. The reactive controller computes these rates across exponentially increasing windows. The smallest window size used is the service time of the pipeline and the windows double in size until a max of 60 seconds. The rates computed with smaller windows are a measure of the burstiness of the arrival dynamics, while the rates with larger windows approach the mean arrival rate λ of the workload. By measuring across many different windows we are able to achieve a fine-granularity characterization of an arrival workload that can detect changes in both burstiness and arrival rate.

Initialization Offline, the planner constructs the traffic envelope for the sample arrival trace. The planner also computes the max-provisioning ratio for each model $\rho_m = \frac{\lambda}{\mu_m}$, the ratio of the arrival rate to the maximum throughput of the model in its current configuration. While the max-provisioning ratio is not a fundamental property of the pipeline, it provides a useful heuristic to measure how much "slack" the planner has determined is needed for this model to be able to absorb bursts and still meet the SLO. The planner then provides the reactive controller with the traffic envelope for the sample trace, the max-provisioning ratio ρ_m and single replica throughput μ_m for each model in the pipeline.

In the interactive applications that InferLine targets, failing to scale up the pipeline in the case of an increased workload results in missed latency objectives and degraded quality of service, while failing to scale down the pipeline in the case of decreased workload only results in slightly higher costs. We therefore handle the two situations separately.

Scaling Up The controller continuously computes the traffic envelope for the current arrival workload. This yields a set of arrival rates for the current workload that can be directly compared to those of the sample workload. If any of the current rates exceed their corresponding sample trace rates, the pipeline is underprovisioned and the procedure for adding replicas is triggered.

At this point, not only has the reactive controller detected that rescaling may be necessary, it also knows what arrival rate it needs to reprovision the pipeline for: the current workload rate r_{max} that triggered rescaling. If the overall λ of the workload has not changed but it has become burstier, this will be a rate computed with a smaller ΔT_i, and if the burstiness of the workload is stationary but the λ has increased, this will be a rate with a larger ΔT_i. In the case that multiple rates have exceeded their sample trace counterpart, we take the max rate.

To determine how to reprovision the pipeline, the controller computes the number of replicas needed for each model to process r_{max} as $k_m = \left\lceil \frac{r_{max} s_m}{\mu_m \rho_m} \right\rceil$. s_m is the scale factor for model m, which prevents over-provisioning for a model that only receives a portion of the queries due to conditional logic. ρ_m is the max-provisioning ratio, which ensures enough slack remains in the model to handle bursts. The reactive controller then adds the additional replicas needed for any models that are detected to be underprovisioned.

Scaling Down InferLine takes a conservative approach to scaling down the pipeline to prevent unnecessary configuration oscillation which can cause SLO misses. Drawing on the work in [14], the reactive controller waits for a period of time after any configuration changes to allow the system to stabilize before considering any down scaling actions. InferLine uses a delay of 15 seconds (3x the 5 second activation time of spinning up new replicas), but the precise value only needs to provide enough time for the pipeline to stabilize. Once this delay has elapsed, the reactive controller computes the max request rate λ_{new} that has been observed over the last 30 seconds, using 5 second windows.

The system computes the number of replicas needed for each model to process λ_{new} similarly to the procedure for scaling up, setting $k_m = \left\lceil \frac{\lambda_{new} s_m}{\mu_m \rho_p} \right\rceil$. In contrast to scaling up, when scaling down we use the minimum max provisioning factor in the pipeline $\rho_p = \min(\rho_m \forall m \in \text{models})$. Because the max provisioning factor is a heuristic that has some dependence on the sample trace, using the min across the pipeline provides a more conservative downscaling algorithm and ensures the reactive controller is not overly aggressive in removing replicas.

6 Experimental Setup

To evaluate InferLine we constructed four prediction pipelines (Fig. 2) representing common application domains and using models trained in a variety of machine learning frameworks [29, 38, 30, 28]. We configure each pipeline with varying input arrival processes and latency budgets. We evaluate the latency SLO attainment and pipeline cost under a range of both synthetic and real world workload traces.

Coarse-Grained Baseline Comparison Current prediction serving systems do not provide first-class support for prediction pipelines with end-to-end latency constraints. Instead, the individual pipeline components are each deployed as a separate microservice to a prediction serving system such as [39, 18, 10] and a pipeline is manually constructed by individual calls to each service. Any performance tuning for end-to-end latency or cost treats the entire pipeline as a single black-box service and tunes

it as a whole. We therefore use this same approach as our baseline for comparison. Throughout the experimental evaluation we refer to this as the *Coarse-Grained* baseline. We deploy both the InferLine and coarse-grained pipelines on the **Physical Execution Engine** to eliminate performance variability caused by different execution environments.

We use the techniques proposed in [14] for both provisioning and scaling the coarse-grained pipelines as a baseline for comparison. We profile the entire pipeline as a single black box to identify the single maximum batch size capable of meeting the SLO, in contrast to InferLine's per-model profiling. The pipeline is then replicated as a single unit to achieve the required throughput as measured on the same sample arrival trace used by the InferLine **Planner**. We evaluate two strategies for determining required throughput. *CG-Mean* uses the mean request rate computed over the arrival trace while *CG-Peak* determines the peak request rate in the trace computed using a sliding window of size equal to the SLO. The coarse-grained reactive controller scales the number of pipeline replicas using the scaling algorithm described in [14].

Physical Execution Environment We ran all experiments in a distributed cluster on Amazon EC2. The pipeline driver client was deployed on an `m4.16xlarge` instance which has 64 vCPUs, 256 GiB of memory, and 25Gbps networking across two NUMA zones. We used large client instance types to ensure that network bandwidth from the client is not a bottleneck. The models were deployed to a cluster of up to 16 `p2.8xlarge` GPU instances. This instance type has 8 NVIDIA K80 GPUs, 32 vCPUs, 488.0 GiB of memory and 10Gbps networking all within a single NUMA zone. All instances were running Ubuntu 16.04 with Linux Kernel version 4.4.0.

CPU costs were computed by dividing the total hourly cost of an instance by the number of CPUs. GPU costs were computed by taking the difference between a GPU instance and its equivalent non-GPU instance (all other hardware matches), then dividing by the number of GPUs. This cost model provides consistent prices across different GPU generations and instance sizes.

Workload Setup We generated synthetic traces by sampling inter-arrival times from a Gamma distribution with differing mean μ to vary the request rate, and coefficient of variation CV to vary the workload burstiness. When reporting performance on a specific workload as characterized by $\lambda = \frac{1}{\mu}$ and CV, a trace for that workload was generated once and reused across all comparison points to provide a more direct comparison of performance. We generated separate traces with the same performance characteristics for profiling and evaluation to avoid overfitting to the sample trace.

To generate synthetic time-varying workloads, we evolve the workload generating function between different Gamma distributions over a specified period of time, the transition time. This allows us to generate workloads that vary in mean throughput, CV, or both, and thus evaluate the performance of the **Reactive Controller** under a wide range of conditions.

In Fig. 5 we evaluate InferLine on traces derived from real workloads studied in the AutoScale system [14]. These workloads only report the average request rate each minute for an hour, rather than providing the full trace of query inter-arrival times. To derive traces from these workloads, we followed the approach used by [14] to re-scale the max throughput to 300 QPS, the maximum throughput supported by the coarse-grained baseline pipelines on a 16 node cluster. We then iterated through each of the mean request rates in the workload and sample from a Gamma distribution with CV 1.0 for 30 seconds. We use the first 25% of the trace as the sample for offline planning, and the remaining 75% for evaluation (see Fig. 5).

7 Experimental Evaluation

In this section we evaluate InferLine's end-to-end performance. We start with an end-to-end evaluation that compares InferLine's hybrid proactive-reactive approach to the coarse-grained baseline approaches in use today (§7.1). We demonstrate that InferLine outperforms the baselines on both latency SLO attainment and cost on both synthetic and real-world derived workloads. We then perform a sensitivity analysis through a series of micro-benchmarks aimed at gauging InferLine's robustness to the dynamics of the arrival process (§7.2). We find that InferLine is robust to unplanned changes in the arrival rate as well as unexpected inter-arrival bursts. We then perform an ablation study to show that the system benefits from both (a) offline proactive planning and (b) online reactive control (§7.3). We conclude by showing that InferLine generalizes to other prediction serving systems (§7.4).

7.1 End-to-end Evaluation

We first establish that InferLine's proactive and reactive components outperform state of the art coarse-grain pipeline-level configuration alternatives in an end-to-end evaluation (§7.1). InferLine is able to achieve the same throughput at significantly lower cost, while maintaining zero or close to zero latency SLO miss rate.

Proactive control In the absence of a globally reasoning planner (§4.3), the options are limited to either (a)

provisioning for the peak (CG Peak), or (b) provisioning for the mean (CG Mean). We compare InferLine to these two end-points of the configuration continuum across 2 pipelines (Fig. 4). InferLine meets latency SLOs at the lowest cost. CG Peak meets SLOs, but at much higher cost, particularly for burstier workloads. And CG Mean is not provisioned to handle bursty arrivals and results in high SLO miss rates. This result is exacerbated by higher burstiness and lower SLO. Furthermore, the InferLine planner consistently finds lower cost configurations than both coarse-grained provisioning strategies and is able to achieve up to a *7.6x reduction in cost* by minimizing pipeline imbalance.

Reactive controller InferLine is able to (1) maintain a negligible SLO miss rate, and (2) and reduce cost by up to 4.2x when compared to the state-of-the-art approach [14] when handling unexpected changes in the arrival rate and burstiness. In Fig. 5 we evaluate the *Social Media* pipeline on 2 traces derived from real workloads studied in [14]. InferLine finds a *5x cheaper* initial configuration than the coarse-grained provisioning (Fig. 5(a)). Both systems achieve near-zero SLO miss rates throughout most of the workload, and when the big spike occurs we observe that the InferLine's reactive controller quickly reacts by scaling up the pipeline as described in §5.1. As soon as the spike dissipates, InferLine scales the pipeline down to maintain a cost-efficient configuration. In contrast, the coarse-grained reactive controller operates much slower and, therefore, is ill-suited for reacting to rapid changes in the request rate of the arrival process.

In Fig. 5(b), InferLine scales up the pipeline smoothly and recovers rapidly from an instantaneous spike, unlike the CG baseline. Furthermore, as the workload drops quickly after 1000 seconds, InferLine rapidly responds by shutting down replicas to reduce cluster cost. In the end, InferLine and the coarse-grained pipelines converge to similar costs due to the low terminal request rate which hides the effects of pipeline imbalance, but InferLine has a *34.5x lower SLO miss rate* than the baseline.

We further evaluate the differences between the InferLine and coarse-grained reactive controllers on a set of synthetic workloads with increasing arrival rates in Fig. 6. We observe that the traffic envelope monitoring described in §5.1 enables InferLine to detect the increase in arrival rate earlier and therefore scale up the pipeline sooner to maintain a low SLO miss rate. In contrast, the coarse-grained reactive controller only reacts to the increase in request rate at the point when the pipeline is overloaded and therefore reacts when the pipeline is already in an infeasible configuration. The effect of this delayed reaction is compounded by the longer provisioning time needed to replicate an entire pipeline, resulting in the coarse-grained baselines being unable to recover before the experiment ends. They will eventually recover as we see in Fig. 5 but only after suffering a period of 100% SLO miss rate.

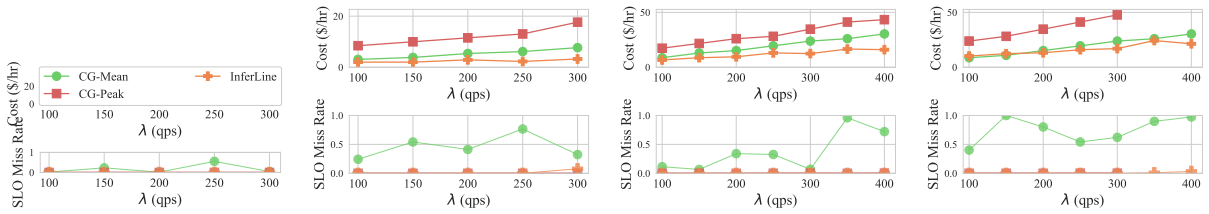
7.2 Sensitivity Analysis

We evaluate the sensitivity and robustness of the **Planner** and the **Reactive Controller**. We analyze the accuracy of the **Estimator** in estimating tail latencies from the sample trace and the proactive **Planner**'s response to varying arrival rates, latency SLOs, and burstiness factors. For the reactive controller, we analyze InferLine's sensitivity to changes in the arrival process.

Proactive Optimizer Sensitivity We first evaluate how closely the latency distribution produced by the estimator reflects the latency distribution of the running system in Fig. 7. We observe that the estimated and measured P99 latencies are close across all four experiments. Further, we see that the estimator has the critical property of ensuring that the P99 latency of feasible configurations is below the latency objective. The near-zero SLO miss rates in Fig. 4 are a further demonstration of the estimator's ability to detect infeasible configurations.

Next, we evaluate the **Planner**'s performance under varying load, burstiness, and end-to-end latency SLOs. We observe three important trends in Fig. 8. First, increasing burstiness (from CV=1 to CV=4) requires more costly configurations as the optimizer provisions more capacity to ensure that transient bursts do not cause the queues to diverge more than the SLO allows. We see this trend over a range of different arrival throughputs. We also see the cost gap narrowing between CV=1 and CV=4 as the SLO increases. As the SLO increases, additional slack in the deadline can absorb more variability in the arrival process. Second, the cost decreases as a function of the latency SLO. While this downward cost trend generally holds, the optimizer occasionally finds sub-optimal configurations, as it makes locally optimal decisions to change a resource assignment. Third, the cost increases as a function of expected arrival rate.

Reactive Controller Sensitivity A common type of unpredictable behavior is a change in the arrival rate. We compare the behavior of the system with and without its reactive controller enabled as the arrival process changes from the provisioned 150qps to 250qps. We vary the rate of arrival throughput change τ. InferLine is able to maintain the SLO miss rate close to zero while matching or beating two alternatives: (a) a proactive-only planner given oracular knowledge of the whole arrival trace a priori, and (b) a proactive-only planner that doesn't respond to change. Indeed, in Fig. 9, InferLine continues to meet the SLO, and increases the cost of the allocation only for



(a) Image Processing CV 1.0 (b) Image Processing CV 4.0 (c) Video Monitoring CV 1.0 (d) Video Monitoring CV 4.0

Figure 4: Comparison of InferLine's proactive planning to coarse-grained baselines (150ms SLO) InferLine outperforms both baselines, consistently providing both the lowest cost configuration and highest SLO attainment (lowest miss rate). CG-Peak was not evaluated on $\lambda > 300$ because the configurations exceeded cluster capacity.

the duration of the unexpected burst. The oracle proactive planner with full knowledge of the workload is able to find the cheapest configuration at the peak because it is equipped with the ability to tune all three control knobs. But it pays this cost for the entire duration of the workload. The proactive-only planner without oracular knowledge starts missing latency SLOs as soon as the ingest rate increases.

A less obvious but potentially debilitating change in the arrival process is an increase in its burstiness, even while maintaining the same mean arrival rate. This type of arrival process change is also harder to detect, as the common practice is to look at moments of the arrival rate distribution, such as the mean or 99th percentile latency. In Fig. 10 we show that InferLine is able to *detect* deviation from expected arrival burstiness and react to meet the latency SLOs.

7.3 Attribution of Benefit

InferLine benefits from (a) offline proactive planning and (b) online reactive scaling. Thus, we evaluate the following comparison points: baseline coarse grain proactive (Baseline Pro), InferLine proactive (InferLine Pro), InferLine proactive with baseline reactive (InferLine Pro + Baseline React), and InferLine proactive with InferLine reactive (InferLine Pro + InferLine React), building up from pipeline-level configuration to the full feature set InferLine provides. InferLine's proactive planning reduces the cost of the initial pipeline configuration by more than 3x (Fig. 11), but starts missing latency SLOs when the request rate increases. Adding the reactive controller (InferLine Pro + Baseline React) adapts the configuration, but too late to completely avoid SLO misses, although it recovers faster than proactive-only options. The InferLine reactive controller has the highest SLO attainment and is the only point that maintains the SLO across the entirety of the workload. This emphasizes the need for both the **Planner** for initial cost-efficient pipeline configuration, and the **Reactive Controller** to promptly and cost-efficiently adapt.

7.4 Extension to TF-Serving

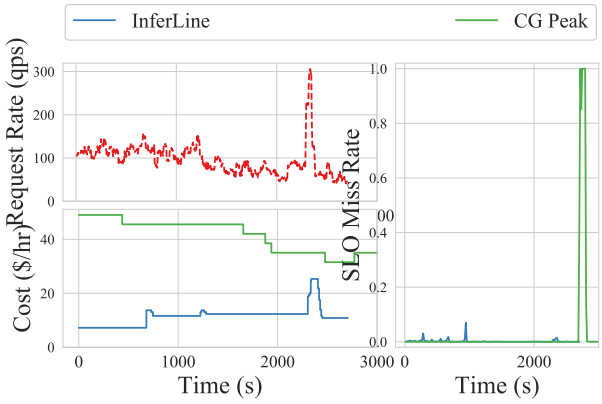
The contributions of this work generalize to different physical serving engines. Here we layer the InferLine planner and queuing system on top of Tensorflow Serving (TFS)—a state-of-the-art model serving framework developed at Google. In this experiment, we achieve the same low latency SLO miss rate as InferLine. This indicates the generality of the planning algorithms used to configure individual models in InferLine. In Fig. 12 we show both the SLO attainment rates and the cost of pipeline provisioning when running InferLine on our in-house physical execution engine and on TFS. The cost of the latter is slightly worse due to the additional overhead of TFS RPC serialization. TFS uses gRPC [15] for communication which requires an additional copy of the inputs and outputs of a model, while our serving engine uses an optimized zero-copy RPC implementation.

8 Related Work

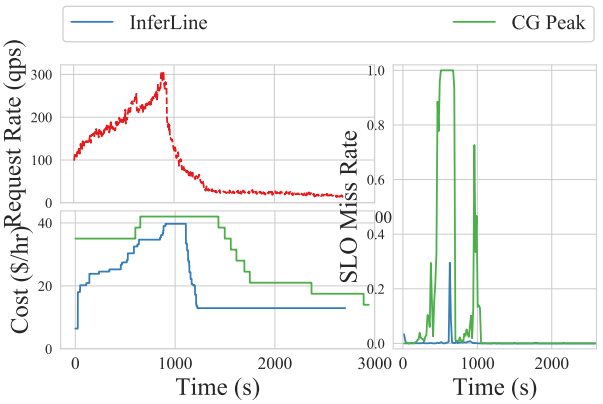
A number of recent efforts study the design of generic prediction serving systems [10, 5, 39]. TensorFlow Serving [39] is a commercial grade prediction serving system primarily designed to support prediction pipelines implemented using TensorFlow [38]. Unlike InferLine, TensorFlow Serving adopts a monolithic design with the pipeline orchestration living within a single process. Thus, TensorFlow Serving is able to introduce performance optimizations like operator fusion across computation stages to reduce coordination between the CPU and GPU at the expense of fine-grain, independent model configuration. However, TensorFlow-Serving does not support latency SLO constraints.

Clipper [10] adopts a more distributed design, similar to InferLine. Like InferLine, each model in Clipper is individually managed, configured, and deployed in separate containers. However, Clipper does not directly support prediction pipelines or reasoning about latency deadlines across models. It also does not dynamically scale to changing workloads.

There are several systems that have explored offline pipeline configuration for data pipelines [19, 7]. However,



(a) AutoScale Big Spike



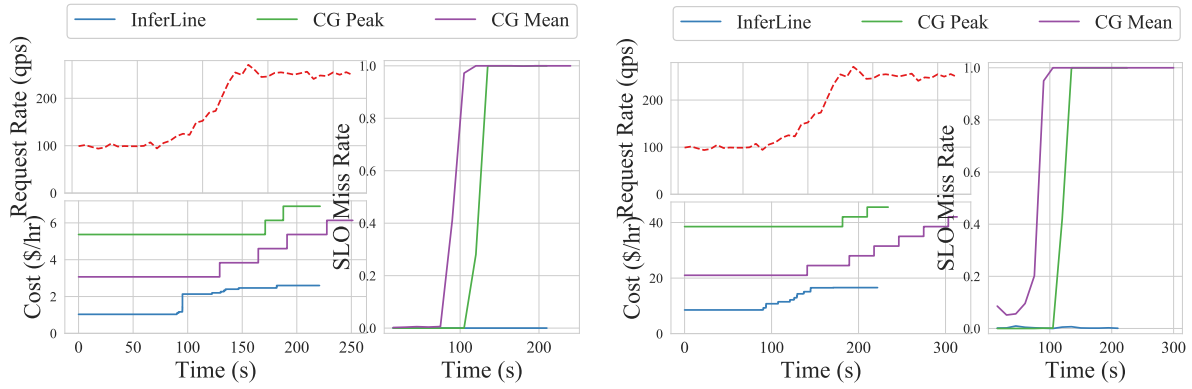
(b) AutoScale Slowly Varying

Figure 5: Performance comparison of the reactive control algorithms on traces derived from real workloads. These are the same workloads evaluated in [14] which forms the basis for the coarse-grained baseline. Both workloads were evaluated on the Social Media pipeline with a 150ms SLO. In Fig. 5(a), InferLine maintains a 99.8% SLO attainment overall at a total cost of $8.50, while the coarse-grained baseline has a 93.7% SLO attainment at a cost of $36.30. In Fig. 5(b), InferLine has a 99.3% SLO attainment at a cost of $15.27, while the coarse-grained baseline has a 75.8% SLO attainment at a cost of $24.63, a 34.5x lower SLO miss rate.

these focus on tuning generic data streaming pipelines. As a result, these systems use black box optimization techniques that require running the pipeline end-to-end to measure performance under each candidate configuration. InferLine instead leverages accurate offline performance profiles of each stage and a simulation-based performance estimator to rapidly explore the configuration space offline, without running the pipeline end-to-end using cluster resources.

Dynamic pipeline scaling is a critical feature in data streaming systems to avoid backpressure and overprovisioning. Systems such as [21, 12] are throughput-oriented with the goal of maintaining a well-provisioned system under changes in the request rate. The DS2 autoscaler in [21] estimates true processing rates for each operator in the pipeline online by instrumenting the underlying streaming system. They use these processing rates in conjunction with the pipeline topology structure to estimate the optimal degree of parallelism for all operators at once. In contrast, [12] identifies a single bottleneck stage at a time, taking several steps to converge from an underprovisioned to a well-provisioned system. Both systems provision for the average ingest rate and ignore any burstiness in the workload which can transiently overload the system. In contrast, InferLine maintains a traffic envelope of the request workload and uses this to ensure that the pipeline is well-provisioned for the peak workload across several timescales simultaneously, including any burstiness (see §5.1).

In Fig. 13 we evaluate the performance of DS2 [21], which is an open source, state-of-the-art autoscaling streaming system, on its ability to meet latency SLOs under a variety of workloads. We deployed the Image Processing pipeline (Fig. 2(a)) in DS2 running on Apache Flink [11] without any batching on a single `m4.16xlarge` AWS EC2 instance. As we can see in Fig. 13(a), provisioning for the average request rate is sufficient to meet latency objectives under uniform workloads. But as CV increases to 4.0, the latency SLO miss rate increases. Part of the reason for this increase is that bursts in the re-



(a) Image Processing

(b) Social Media

Figure 6: Performance comparison of the reactive control algorithms on synthetic traces with increasing arrival rates. We observe that InferLine outperforms both coarse-grained baselines on cost while maintaining a near-zero SLO miss rate for the entire duration of the trace.

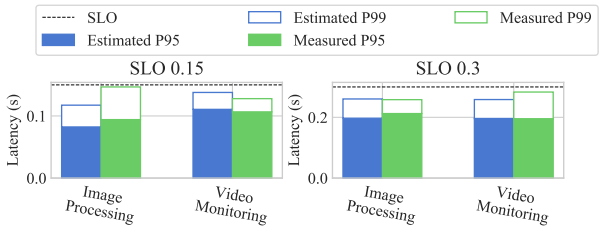


Figure 7: Comparison of estimated and measured tail latencies. We compare the latency distributions produced by the Estimator on a workload with λ of 150 qps and CV of 4, observing that in all cases the estimated and measured latencies are both close to each other and below the latency SLO.

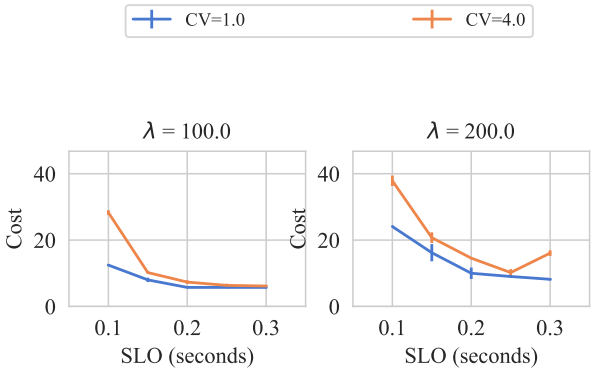


Figure 8: Planner sensitivity: Variation in configuration cost across different arrival processes and latency SLOs for the Social Media pipeline. We observe that 1) cost decreases as SLO increases, 2) burstier workloads require higher cost configurations, and 3) cost increases as λ increases.

quest rate transiently overload the system, causing queries to be delayed in queues until the system recovers. In addition, DS2 occasionally misinterprets transient bursts as changes in the overall request rate and scaled up the pipeline, requiring Apache Flink to halt processing and save state before migrating to the new configuration.

We observe this same degradation under non-stationary workloads in Fig. 13(b) where we measure P99 latency over time for a workload that starts out with a CV of 1.0 and a request rate of 50 qps, then increases the request rate to 100 qps over 60 seconds. It takes nearly 300 seconds after the request rate starts to increase for the system to re-stabilize and the queues to fully drain from the repeated pipeline re-configurations. In contrast, as we see in Fig. 9 and Fig. 10, InferLine is capable of maintaining SLOs under a variety of changes to the workload dynamics. Furthermore, because the operators are all stateless in InferLine, the **Physical Execution Engine** does not need to fully stop processing when scaling but can instead incrementally add and remove replicas. There is ongoing work such as [17] to address live state migration in streaming systems but Apache Flink does not currently support any such mechanisms.

A few streaming autoscaling systems consider latency-oriented performance goals [13, 24]. The closest work to InferLine, [24] from Lohrmann et al. as part of their work on Nephele [25], treats each stage in a pipeline as a single-server queuing system and uses queuing theory to estimate the total queue waiting time of a job under different degrees of parallelism. They leverage this queuing model to greedily increase the parallelism of the stage with the highest queue waiting time until they can meet the latency SLO. However, their queuing model only considers average latency, and provides no guarantees about the behavior of tail latencies. InferLine's reactive controller provisions for worst-case latencies.

VideoStorm [43] explores the design of a streaming video processing system that adopts a distributed design with pipeline operators provisioned across compute nodes and explores the combinatorial search space of hardware and model configurations. VideoStorm jointly optimizes for quality and lag and does not provide latency guarantees.

A large body of prior work leverages profiling for

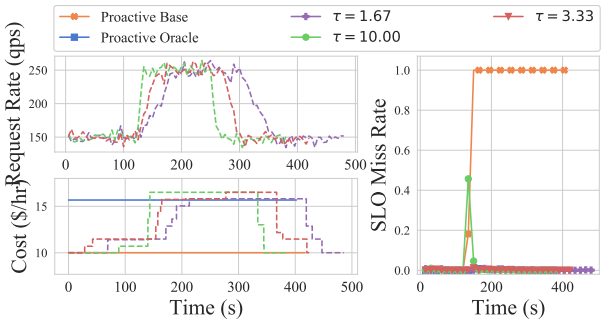


Figure 9: Sensitivity to arrival rate changes (Social Media pipeline). We observe that the Reactive Controller quickly detects and scales up the pipeline in response to increases in λ. Further, the reactive controller finds cost-efficient configurations that either match or are close to those found by the Planner given oracular knowledge of the trace.

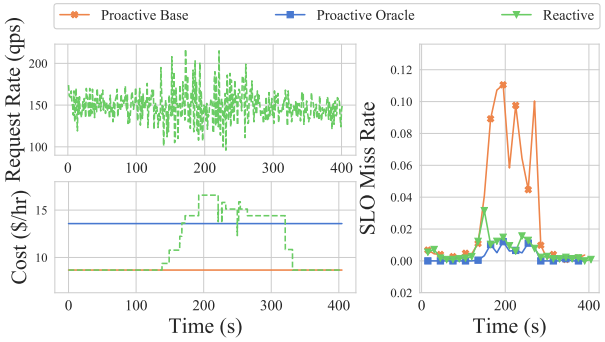


Figure 10: Sensitivity to arrival burstiness changes (Social Media Pipeline). We observe that the network-calculus based detection mechanism of the Reactive Controller detects changes in workload burstiness and takes the appropriate scaling action to maintain a near-zero SLO miss rate.

scheduling, including recent work on workflow-aware scheduling [32, 20]. In contrast, InferLine exploits the compute-intensive and side-effect free nature of ML models to estimate end-to-end pipeline performance based on individual model profiles.

Autoscale [14] offers a comprehensive literature review of a body of work aimed at automatically scaling the number of servers reactively, subject to changing load in the context of web services. Autoscale works well for single model replication without batching as it assumes bit-at-a-time instead of batch-at-a-time query processing. However, we find that the InferLine fine-grained reactive controller far outperforms the coarse-grain baselines that use the Autoscale scaling mechanism on both latency SLO attainment and cost (§7.1).

9 Limitations and Generality

One limitation of the offline planner is its assumption that the available hardware has a total ordering of latency across all batch sizes. As specialized accelerators for ML

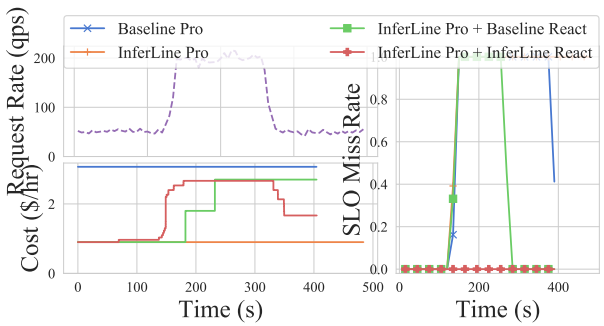
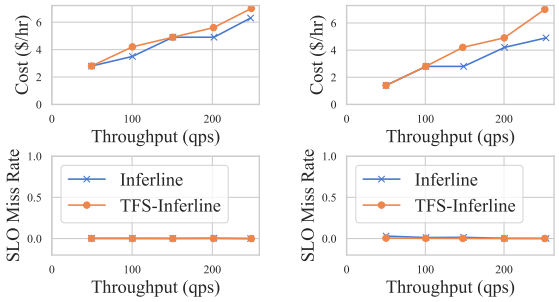


Figure 11: Attribution of benefit between the InferLine proactive and reactive components on the Image Processing pipeline. We observe that the Planner finds a more than 3x cheaper configuration than the baseline. We also observe that InferLine's Reactive Controller is the only combination that maintains the latency SLO throughout the trace.



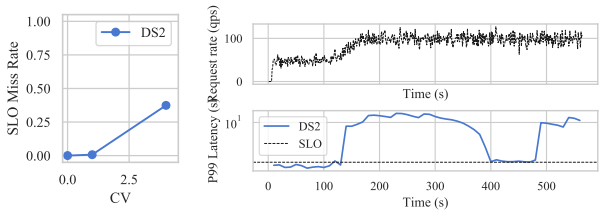
(a) SLO 0.15 (b) SLO 0.3

Figure 12: Comparison of InferLine's planner running on our serving engine and on TensorFlow-Serving on the TF Cascade pipeline.

continue to proliferate, there may be settings where one accelerator is slower than another at smaller batch sizes but faster at larger batch sizes. This would require modifications to the hardware downgrade portion of the planning algorithm to account for this batch-size dependent ordering.

A second limitation is the assumption that the inference latency of ML models is independent of their input. There are emerging classes of machine learning tasks where state-of-the-art models have inference latency that varies based on the input. For example, object detection models [31, 30] will take longer to make predictions on images with many objects in them. Similarly, the inference latency of machine translation models [42, 41] increases with the length of the document. One simple way of modifying InferLine to account for this is to measure this latency distribution during profiling based on the variability in the sample queries and use the tail of the distribution (e.g., 99% or k standard deviations above the mean) as the processing time in the estimator, which will lead to feasible but more costly configurations.

Finally, while we only study machine learning prediction pipelines in this work, there may be other classes of applications that have similarly predictable performance



(a) Increasing CV (b) Increasing Arrival Rate

Figure 13: Performance of DS2 on Bursty and Stochastic Workloads. We observe that DS2 is unable to maintain the latency SLO under (a) bursty workloads, and (b) workloads with increasing request rate.

and can therefore be profiled. We leave the extension of InferLine to applications beyond machine learning as future work.

10 Conclusion

Configuring and serving sophisticated prediction pipelines across heterogeneous parallel hardware while minimizing cost and ensuring SLO attainment presents categorically new challenges in system design. In this paper we introduce InferLine–a system which efficiently provisions and executes prediction pipelines subject to end-to-end latency constraints by proactively optimizing and reactively controlling per-model configurations. We leverage the predictable performance scaling characteristics of machine learning models to enable accurate end-to-end performance estimation and proactive cost minimizing pipeline configuration. In addition, by exploiting the model profiles and network-calculus based detection mechanism we are able to *reactively* adapt pipeline configurations to accommodate changes in the query workload. InferLine builds on three main contribution: (a) the end-to-end latency estimation procedure for prediction pipelines spanning multiple hardware and model configurations (§4.1 and §4.2), (b) the offline proactive planning algorithm for pipeline configuration (§4.3), and (c) the online robust reactive controller algorithm to adapt to unexpected changes in load (§5.1), to achieve the combined effect of *cost-efficient* heterogeneous prediction pipelines that can be deployed to serve applications with a range of tight end-to-end latency objectives. As a result, we achieve up to 7.6x improvement in cost and 34.5x improvement in SLO attainment for the same throughput and latency objectives over state-of-the-art alternatives.

References

[1] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, J. Chen, J. Chen, Z. Chen, M. Chrzanowski, A. Coates, G. Diamos, K. Ding, N. Du, E. Elsen, J. Engel, W. Fang, L. Fan, C. Fougner, L. Gao, C. Gong, A. Hannun, T. Han, L. Johannes, B. Jiang, C. Ju, B. Jun, P. LeGresley, L. Lin, J. Liu, Y. Liu, W. Li, X. Li, D. Ma, S. Narang, A. Ng, S. Ozair, Y. Peng, R. Prenger, S. Qian, Z. Quan, J. Raiman, V. Rao, S. Satheesh, D. Seetapun, S. Sengupta, K. Srinet, A. Sriram, H. Tang, L. Tang, C. Wang, J. Wang, K. Wang, Y. Wang, Z. Wang, Z. Wang, S. Wu, L. Wei, B. Xiao, W. Xie, Y. Xie, D. Yogatama, B. Yuan, J. Zhan, and Z. Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 173–182, New York, New York, USA, 20–22 Jun 2016. PMLR.

[2] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. Deep compositional question answering with neural module networks. *CoRR*, abs/1511.02799, 2015.

[3] A. Angelova, A. Krizhevsky, V. Vanhoucke, A. S. Ogale, and D. Ferguson. Real-Time Pedestrian Detection with Deep Network Cascades. *BMVC*, pages 32.1–32.12, 2015.

[4] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, Jan. 2003.

[5] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. TFX: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1387–1395. ACM, 2017.

[6] A. Beck. Simulation: the practice of model development and use. *Journal of Simulation*, 2(1):67–67, Mar 2008.

[7] M. Bilal and M. Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 189–200, New York, NY, USA, 2017. ACM.

[8] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.

[9] C. Chiu, T. N. Sainath, Y. Wu, R. Prabhavalkar, P. Nguyen, Z. Chen, A. Kannan, R. J. Weiss, K. Rao, K. Gonina, N. Jaitly, B. Li, J. Chorowski, and M. Bacchiani. State-of-the-art speech recognition with sequence-to-sequence models. *CoRR*, abs/1712.01769, 2017.

[10] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium*

on Networked Systems Design and Implementation (NSDI 17), pages 613–627, Boston, MA, 2017. USENIX Association.

[11] Apache Flink. `https://flink.apache.org`.

[12] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow.*, 10(12):1825–1836, Aug. 2017.

[13] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. Drs: Auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):3338–3352, Dec. 2017.

[14] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, Nov. 2012.

[15] GRPC. `https://grpc.io/`.

[16] J. Guan, Y. Liu, Q. Liu, and J. Peng. Energy-efficient Amortized Inference with Cascaded Deep Classifiers. *arXiv.org*, Oct. 2017.

[17] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe. Megaphone: Live state migration for distributed streaming dataflows, 2018.

[18] A. W. S. Inc. Amazon sagemaker: Developer guide. *http://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf*, 2017.

[19] P. Jamshidi and G. Casale. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. *MASCOTS*, pages 39–48, 2016.

[20] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 117–134, Berkeley, CA, USA, 2016. USENIX Association.

[21] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 783–798, Berkeley, CA, USA, 2018. USENIX Association.

[22] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

[23] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *WWW*, 2010.

[24] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 399–410, June 2015.

[25] B. Lohrmann, D. Warneke, and O. Kao. Nephele streaming: Stream processing under qos constraints at scale. *Cluster Computing*, 17, 08 2013.

[26] M. Malinowski, M. Rohrbach, and M. Fritz. Ask your neurons: A neural-based approach to answering questions about images. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1–9, 2015.

[27] M. McGill and P. Perona. Deciding How to Decide: Dynamic Routing in Artificial Neural Networks. *arXiv.org*, Mar. 2017.

[28] Open ALPR. `https://www.openalpr.com/`.

[29] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[30] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[31] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[32] G. P. Rodrigo, E. Elmroth, P.-O. Östberg, and L. Ramakrishnan. Enabling workflow-aware scheduling on hpc systems. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 3–14, New York, NY, USA, 2017. ACM.

[33] Apache Samza. `http://samza.apache.org`.

[34] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[35] E. Sparks. *End-to-End Large Scale Machine Learning with KeystoneML*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2016.

[36] Apache Storm. `http://storm.apache.org`.

[37] Y. Sun, X. Wang, and X. Tang. Deep Convolutional Network Cascade for Facial Point Detection. *CVPR*, pages 3476–3483, 2013.

[38] TensorFlow. `https://www.tensorflow.org`.

[39] TensorFlow Serving. `https://tensorflow.github.io/serving`.

[40] Timely Dataflow. `https://github.com/TimelyDataflow/timely-dataflow`.

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[42] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[43] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, 2017. USENIX Association.

[44] S. Zhang, L. Yao, and A. Sun. Deep learning based recommender system: A survey and new perspectives. *CoRR*, abs/1707.07435, 2017.