InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines

Daniel Crankshaw
Microsoft Research
dacranks@microsoft.com

Gur-Eyal Sela
UC Berkeley
ges@berkeley.edu

Xiangxi Mo
UC Berkeley, Anyscale
xmo@berkeley.edu

Corey Zumar
Databricks
czumar@berkeley.edu

Ion Stoica
UC Berkeley, Anyscale
istoica@berkeley.edu

Joseph Gonzalez
UC Berkeley
jegonzal@berkeley.edu

Alexey Tumanov
Georgia Tech
atumanov@gatech.edu

ABSTRACT

Serving ML 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.

持续，包括

复杂的

In this paper we introduce InferLine, a system which provisions and manages the individual stages of prediction pipelines to meet end-to-end tail latency constraints while minimizing cost. InferLine consists of a low-frequency combinatorial planner and a high-frequency auto-scaling tuner. The low-frequency planner leverages stage-wise profiling, discrete event simulation, and constrained combinatorial search to automatically select hardware type, replication, and batching parameters for each stage in the pipeline. The high-frequency tuner uses network calculus to auto-scale each stage to meet tail latency goals 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.

低频组合规划器

SoCC '20, October 19–21, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421285

CCS CONCEPTS

• **General and reference** → *Reliability*; Performance; • **Computer systems organization** → **Availability**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

inference, serving, machine learning, autoscaling

ACM Reference Format:
Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421285

1 INTRODUCTION

Cloud applications as well as cloud infrastructure providers today increasingly rely on ML inference over multiple models linked together in a dataflow DAG. Examples include a digital assistant service (e.g., Amazon Alexa), which combines audio pre-processing with downstream models for speech recognition, topic identification, question interpretation and response and text-to-speech to answer a user's question. The natural evolution of these applications leads to a growth in the complexity of the prediction pipelines. At the same time, their latency-sensitive nature dictates tight tail latency constraints (e.g., 200-300ms). As the pipelines grow and the models used become increasingly sophisticated, they present a unique set of systems challenges for provisioning and managing these pipelines.

① dataflow DAG
↓
complex

② latency constraint

Each stage of the pipeline must be assigned the appropriate hardware accelerator (e.g., CPU, GPU, TPU) — a task complicated by increasing hardware heterogeneity. Each model must

③ hardware heterogeneity

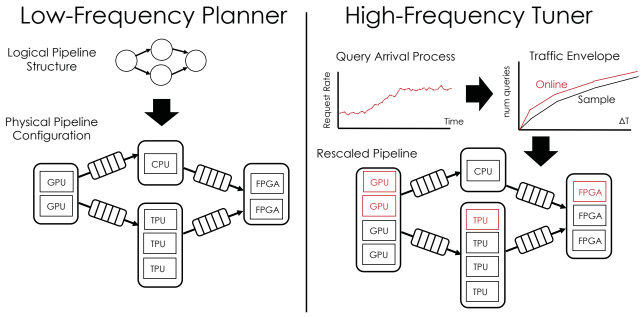


Figure 1: InferLine System Overview

•④ batch size

•⑤ replicated

be configured with the appropriate query batch size — necessary for optimal utilization of the hardware. And each pipeline stage can be replicated to meet the application throughput requirements. Per-stage decisions with respect to the hardware type and batch size affect the latency contributed by each stage towards the end-to-end pipeline latency bound by the application-specified Service Level Objective (SLO). This creates a combinatorial search space with three control dimensions per model (hardware type, batch size, number of replicas) and constraints on aggregate latency.

Problem
{ hardware batch size replicas | & SLO

A number of prediction serving systems exist today, including Clipper [9], TensorFlow Serving [37], and NVIDIA TensorRT Inference Server [36] that optimize for single model serving. This pushes the complexity of coordinating cross-model interaction and, particularly, the questions of per-model configuration to meet application-level requirements, to the application developer. To the best of our knowledge, no system exists today that automates the process of pipeline provisioning and configuration, subject to specified tail latency SLO in a cost-aware manner. Thus, the goal of this paper is to address the problem of configuring and managing multi-stage prediction pipelines subject to end-to-end tail latency constraints cost efficiently.

Exist work
→ single model ~

Goal

We propose InferLine — a system for provisioning and management of ML inference pipelines. It composes with existing prediction serving frameworks, such as Clipper and TensorFlow Serving. It is necessary for such a system to contain two principal components: a low-frequency planner and a high-frequency tuner. The low-frequency planner is responsible for navigating the combinatorial search space to produce per-model pipeline configuration relatively infrequently to minimize cost. It is intended to run periodically to correct for workload drift or fundamental changes in the steady-state, long-term query arrival process. It is also necessary for integrating new models added to the repository and to integrate new hardware accelerators. The high frequency component is intended to operate at time scales three orders of magnitude faster. It monitors instantaneous query arrival traffic and tunes the running pipeline to accommodate unexpected query spikes cost efficiently to maintain latency SLOs under bursty and stochastic workloads.

注意
包含 ← TF-serving or Clipper ...

A. planner

B. tuner

To enable efficient exploration of the combinatorial configuration space, 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 the hardware configuration and batchsize parameters. The low-frequency *planner* uses a constrained greedy search algorithm to find the cost-minimizing pipeline configuration that meets the end-to-end tail latency constraint determined using the discrete event simulator on a sample planning trace.

planner:
profiling
+
greedy search

The InferLine high-frequency *tuner* leverages traffic envelopes built using network calculus tools to capture the arrival process dynamics across multiple time scales and determine when and how to react to changes in the arrival process. As a consequence, the tuner is able to maintain the latency SLO in the presence of transient spikes and sustained variation in the query arrival process.

tuner:
traffic envelops

In summary, the primary contribution of this paper is a system for provisioning and managing machine learning inference pipelines for latency-sensitive applications cost efficiently. It consists of two key components that operate at time scales orders of magnitude apart to configure the system for near-optimal performance. The planner builds on a high-fidelity model-based networked queueing simulator, while the tuner uses network calculus techniques to rapidly adjust pipeline configuration, absorbing unexpected query traffic variation cost efficiently.

Contribution

per model config
↗
planner → cost
tune → SLO
↓
unexpected burst

We apply InferLine to provision and manage resources for multiple state-of-the-art prediction serving systems. We show that InferLine significantly outperforms alternative pipeline configuration baselines by a factor of up to 7.6X on cost, while exceeding 99% latency SLO attainment—the highest level of attainment achieved in relevant prediction serving literature.

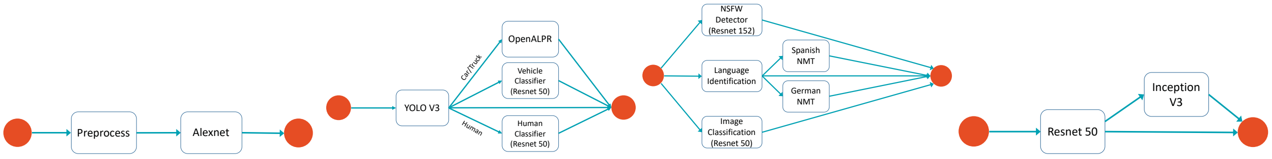
2 BACKGROUND AND MOTIVATION

Prediction pipelines combine multiple machine learning models and data transformations to support complex prediction tasks [32]. For instance, state-of-the-art visual question answering services [1, 23] combine language models with vision models to answer the question.

A prediction pipeline can be represented as a directed acyclic graph (DAG), where each vertex corresponds to a model (e.g., mapping images to objects in the image) or a data transformation (e.g., extracting key frames from a video) and edges represent dataflow between vertices.

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

pipeline
↓
dataflow DAG
↓
example fig.2


Preprocess
Alexnet
YOLO V3
Car/Truck
Human
OpenALPR
Vehicle Classifier (Resnet 50)
Human Classifier (Resnet 50)
NSFW Detector (Resnet 152)
Language Identification
Spanish NMT
German NMT
Image Classification (Resnet 50)
Resnet 50
Inception V3

(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 range of models, control flow, and input characteristics.

network. The Video Monitoring pipeline was inspired by [40] 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 models to identify the source language and translate the post if necessary. The TensorFlow (TF) Cascade pipeline combines fast and slow TensorFlow 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 [3, 20] used for model personalization as well as more general cascaded prediction pipelines [2, 14, 24, 34].

We show that for such pipelines 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 Challenges

Prediction pipelines present new challenges for the design and provisioning of prediction serving systems. First, we discuss how the proliferation of specialized hardware accelerators and the need to meet end-to-end latency constraints leads to a combinatorially large configuration space. Second, we discuss some of the complexities of meeting tight latency SLOs under bursty stochastic query loads. Third, 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 (Fig. 3). However, not all models benefit equally from hardware accelerators. For example, several widely used classical models (e.g., decision trees [7]) can be difficult to parallelize on GPUs, and common data transformations (e.g. text feature extraction) often cannot be efficiently computed on GPUs.

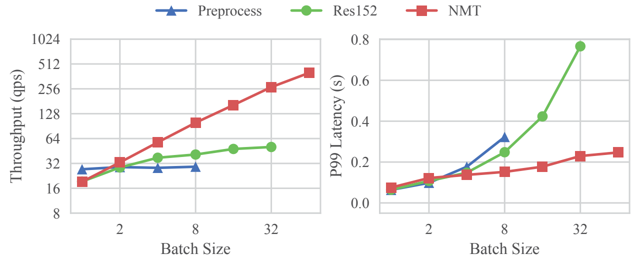


Figure 3: Example Model Profiles on K80 GPU. The preprocess model has no internal parallelism and cannot utilize a GPU. Thus, it sees no benefit from batching. Res152 (image classification) & TF-NMT(text translation model) benefit from batching on a GPU but at the cost of increased latency.

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, constrains feasible hardware configurations as well.

Queueing Delays: As stages of a pipeline may operate at different speeds, due to resource and model heterogeneity, it is necessary to have a queue per stage. Queueing also allows to absorb query inter-arrival process irregularities and can be a significant end-to-end latency component. Queueing delay must be explicitly considered during pipeline configuration, as it directly depends on the relationship between the inter-arrival process and system configuration.

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 $CV_A^2 = \frac{\sigma^2}{\mu^2}$, where $\mu = \frac{1}{\lambda}$ and σ are the mean and standard-deviation of the query inter-arrival time. Processes with higher CV_A^2 have higher variability and often require additional over-provisioning to meet latency objectives. Clearly, over-provisioning the whole 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 monitor, quickly detect, and *tune* 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 [8, 10, 30, 33, 38]. 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. These systems tend to focus on maximizing throughput while avoiding back-pressure, with latency as a second order performance goal (§8). Even those that consider latency directly such as [8] do not manage tail latency.

3 SYSTEM DESIGN

In this section, we provide a high-level overview of the main system components in InferLine (Fig. 1). The system requires a *planner* that operates infrequently and re-configures the whole pipeline w.r.t. all of the control parameters and a *tuner* that makes adjustments to the pipeline configurations in response to dynamically observed query traffic patterns.

InferLine runs on top of any prediction serving system that meets a few simple requirements. The underlying serving system must be able to 1) deploy multiple replicas of a model and scale the number of replicas at runtime across a cluster of worker machines, 2) allow for batched inference with the ability to configure a maximum batch size, and 3) use a centralized batched queueing system to distribute batches among model replicas. The first two properties are necessary for InferLine to configure the serving engine, and a centralized queueing system provides deterministic queueing behavior that can be accurately simulated by the **Estimator**. In our experimental evaluation, we run InferLine with both Clipper [9] and TensorFlow Serving [37]. Both systems needed only minor modifications to meet these requirements.

Using InferLine: To deploy a new prediction pipeline managed by 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 calls to models hosted in the underlying serving system to execute the pipeline.

The **Planner** runs as a standalone Python process that runs periodically independent of the prediction serving framework. The **Tuner** runs as a standalone process implemented in C++. It observes the incoming arrival trace streamed to it by the centralized queueing system and triggers model addition/removal executed by serving-framework-specific APIs.

Low-Frequency Planning: The first time planning is performed, InferLine uses the **Profiler** to create 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 **Planner** finds a cost-efficient initial pipeline configuration subject to the end-to-end latency SLO and the specified arrival process. It uses a globally-aware, cost-minimizing optimization algorithm to set the three control parameters for each model in the pipeline. In each iteration of the optimization algorithm, the Planner uses the model profiles to select a cost-minimizing step while relying on the **Estimator** to check for latency constraint violations. After the initial configuration is generated and the pipeline is deployed to serve live traffic, the Planner is re-run periodically (hours to days) on the most recent arrival history to find a cost-optimal configuration for the current workload.

High-Frequency Tuning: The **Tuner** monitors the dynamic behavior of the arrival process to adjust per-model replication factors and maintain high SLO attainment at low cost. InferLine only adjusts per-model replication factors during tuning to avoid expensive hardware migration operations during live serving and to ensure that scaling decisions can be made quickly to maintain latency SLOs even during sharp bursts. The Tuner continuously monitors the current traffic envelope [19] to detect deviations from the planning trace traffic envelope at different timescales simultaneously. By analyzing the timescale at which the deviation occurred, the Tuner is able to take appropriate mitigating action within seconds to ensure that SLOs are met without unnecessarily increasing cost. It ensures that latency SLOs are maintained during unexpected changes to the arrival workload in between runs of the Planner.

4 LOW-FREQUENCY PLANNING

During 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** 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, called the *scale factor*, s. The scale factor represents the conditional probability that a model will be queried given a query entering the pipeline, independent of the behavior of any other models. It is used by the **Estimator** to simulate the effects of conditional control flow on latency (§4.2) and the **Tuner** to make scaling decisions (§5).

The Profiler captures model throughput as a function of hardware type and 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. Profiling only needs to be performed once for each hardware and batch size pair and is re-used in subsequent runs of the Planner.

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 [5], simulating the entire pipeline, including queueing delays and conditional control flow (using the scale factor s). 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.

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) $\geq$ 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;
```

The Estimator simulates the deterministic behavior of queries flowing through a centralized batched queueing system. It combines this with the model profile information which informs the simulator how long a model running on a specific hardware configuration will take to process a batch of a given size.

4.3 Planning Algorithm

At a high-level, the 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), it 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). Otherwise, 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

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;
```

(line 5), searching for the modification that 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. 10, the steep decrease in cost when moving from an SLO of 0.1 to 0.15 can be largely attributed to downgrading the hardware of a language identification model from a GPU to a CPU.

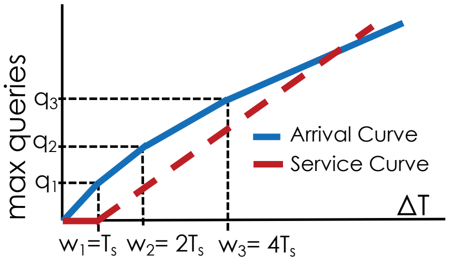


Figure 4: Arrival and Service Curves. The arrival curve captures the maximum number of queries to be expected in any interval of time x seconds wide. The service curve plots the expected number of queries processed in an interval of time x seconds wide.

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

5 HIGH-FREQUENCY TUNING

InferLine's Planner finds an efficient, low-cost configuration that is guaranteed to meet the provided latency objective. However, this guarantee only holds for the sample planning workload provided to the planner. Real workloads evolve over time, changing in both arrival rate (change in λ) as well as becoming more or less bursty (change in CV_A^2). 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 **Tuner** both *detects* these changes as they occur and takes the appropriate *scaling action* to maintain both the latency constraint and cost-efficiency objective.

In order to maintain P99 latency SLOs, the Tuner must be able to detect changes in the arrival workload dynamics across multiple timescales simultaneously. The Planner guarantees that the pipeline is adequately provisioned for the sample trace. The Tuner'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 [19] to characterize the workloads.

A traffic envelope for a workload is constructed by sliding a window of size ΔT_i over the workload's inter-arrival

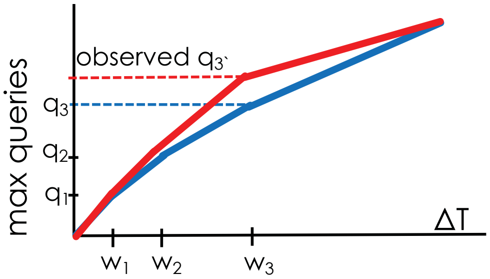


Figure 5: Observed traffic envelope exceeds sample envelope. The observed traffic envelope (in red) exceeds the sample trace traffic envelope (in blue) at window w_3, triggering the Tuner to scale up the pipeline. The pipeline will be re-scaled for the new arrival rate $r_{max} = \frac{q_{3'}}{w_3}$.

process and capturing the maximum number of queries seen anywhere within this window (see Fig. 4). Thus, each $x = \Delta T_i$ is mapped to $y = q_i$ (number of queries) for all x over the duration of a trace. This powerful characterization captures how much the workload can burst in any given interval of time. In practice, we discretize the x-axis by setting the smallest ΔT_i to T_s, the service time of the system, and then double the window size up to 60 seconds. For each such interval, the maximum arrival rate r_i for this interval can be computed as $r_i = \frac{q_i}{\Delta T_i}$. By measuring r_i across all ΔT_i *simultaneously* we capture a fine-grain characterization of the arrival workload that enables simultaneous detection of changes in both short term (burstiness) and long term (average arrival rate) traffic behavior.

Initialization: During planning, 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 λ to the maximum throughput of the model μ 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 Tuner 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 low-latency 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 (Algorithm 3): The Tuner 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 (as in Fig. 5). If any of the current rates exceed their corresponding sample rates (lines 3-6), the pipeline is underprovisioned and the Tuner checks whether it should add replicas for any models in the pipeline.

Algorithm 3: Reactively scale up the pipeline

```
1  Function CheckScaleUp():
2      $r_{\max}$ = -1;
3      for $i$ in Windows.size do
4          $r_{\text{obs}}$ = MaxQueries [i]/Windows [i];
5          if $r_{\text{obs}} >$ SampleRates [i] then
6              $r_{\max}$ = Max($r_{\max}$, $r_{\text{obs}}$);
7      if $r_{\max} > 0$ then
8          foreach model in Pipeline do
9              $k_m = r_{\max}$ * model.scalefactor /
                 (model.throughput * model.$\rho$);
10             reps = Ceil($k_m$) - model.replicas;
11             if reps > 0 then
12                 AddReps(model, reps);
13                 LastUpdate = Now();
```

At this point, not only has the Tuner 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 Tuner computes the number of replicas needed for each model to process r_{max} as $k_m = \left\lceil \frac{r_{max} s_m}{\mu_m \rho_m} \right\rceil$ (lines 9-10). 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 Tuner then adds the additional replicas needed for any models that are underprovisioned (lines 11-12).

Scaling Down (Algorithm 4): 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 [13], the Tuner 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 in the underlying prediction serving frameworks), but the precise value is unimportant as long as it provides enough time for the pipeline to stabilize after a scaling action. Once this delay has elapsed (line 2), the

Algorithm 4: Reactively scale down the pipeline

```
1 Function CheckScaleDown():
2     if (Now() - LastUpdate) > 15 then
3         $\lambda_{\text{new}}$ = Max(RecentLambdas);
4         foreach model in Pipeline do
5             $k_m = \lambda_{\text{new}}$ * model.scalefactor /
                (model.throughput * $\rho_{\min}$);
6             extraReps = model.replicas - Ceil($k_m$);
7             if extraReps > 0 then
8                 RemoveReps(model, extraReps);
```

Tuner continuously computes the max request rate λ_{new} that has been observed over the last 30 seconds, using 5 second windows (line 3).

The Tuner computes the number of replicas needed for each model to process λ_{new} similarly to the procedure for scaling up, setting $k_m = \left\lceil \frac{\lambda_{new} s_m}{\mu_m \rho_p} \right\rceil$ (lines 5-6). 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 Tuner is not overly aggressive in removing replicas. If the workload has dropped substantially, the next time the Planner runs it will find a new lower-cost configuration that is optimal for the new workload.

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 [25–27, 35]. 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 functionality for provisioning and managing prediction pipelines with end-to-end latency constraints. Instead, the individual pipeline components are each deployed as a separate micro-service to a prediction serving system such as [9, 15, 36, 37] 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 pipelines configured with both InferLine and the coarse-grained baseline to the same underlying prediction-serving framework. All experiments used Clipper [9] as the prediction-serving framework except for those in Fig. 14 which compare InferLine running on Clipper and TensorFlow Serving [37]. Both prediction-serving frameworks were modified to add a centralized batched queueing system.

We use the techniques proposed in [13] to do both low-frequency planning and high-frequency tuning for the coarse-grained pipelines as a baseline for comparison. In this baseline, 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 **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 tuning mechanism scales the number of pipeline replicas using the scaling algorithm introduced in [13].

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. 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 ran 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 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 CV_A^2 to vary the workload burstiness. When reporting performance on a specific workload as characterized by $\lambda = \frac{1}{\mu}$ and CV_A^2, 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_A^2, or both, and thus evaluate the performance of the **Tuner** under a wide range of conditions.

In Fig. 7 we evaluate InferLine on traces derived from real workloads studied in the AutoScale system [13]. 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 [13] to re-scale the max throughput to 300 QPS, the maximum throughput supported by the coarse-grained baseline pipelines on a 16 node (128 GPU) cluster. We then iterated through each of the mean request rates in the workload and sample from a Gamma distribution with CV_A^2 1.0 for 30 seconds. We use the first 25% of the trace as the sample for the Planner, and the remaining 75% as the live serving workload (see Fig. 7).

7 EXPERIMENTAL EVALUATION

In this section we evaluate InferLine's performance. First, we evaluate end-to-end performance of InferLine relative to current state of the art methods for configuring and provisioning prediction pipelines with end-to-end latency constraints (§7.1). We show that InferLine outperforms the baselines on latency SLO attainment and cost for synthetic and real-world derived workloads with both stable and unpredictable workload dynamics. Second, we demonstrate that InferLine is robust to unplanned dynamics of the arrival process (§7.2): changes in the arrival rate as well as unexpected inter-arrival bursts, as the **Tuner** rapidly re-scales the pipeline in response to these changes. Third, we perform an ablation study to show that the system benefits from both the low-frequency planning and high-frequency tuning. We conclude by showing that InferLine composes with multiple underlying prediction-serving frameworks (§7.4).

7.1 End-to-end Evaluation

We first establish that InferLine's planning and tuning components outperform state-of-the-art 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 near-zero latency SLO miss rate.

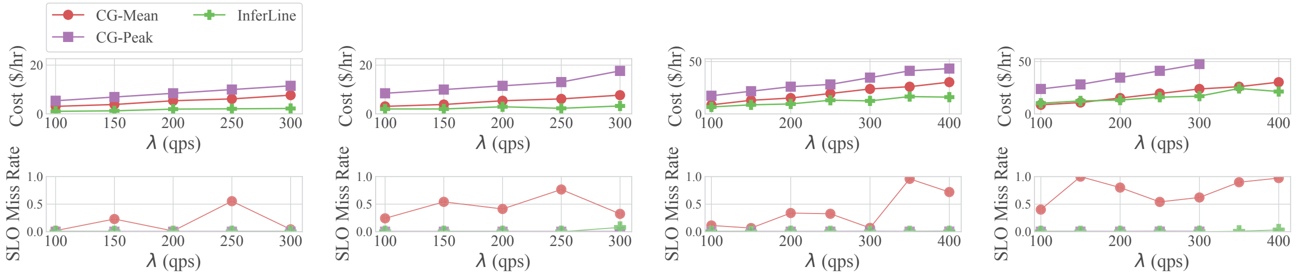
Low-Frequency Planning: In the absence of a workload-aware planner (§4.3), the options are limited to either (a) provisioning for the peak (CG Peak), or (b) provisioning for the mean (CG Mean) request rate. We compare InferLine to these two end-points of the configuration continuum across 2 pipelines (Fig. 6). 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 bursts which results in high SLO miss rates.

The **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. Finally, we observe that the Planner consistently finds configurations that meet the SLO for workloads with the same characteristics as the sample trace used for planning. Next, we evaluate the Tuner's ability to meet SLOs during *unexpected* changes in workload.

High-Frequency Tuning: 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 [13] when handling unexpected changes in the arrival rate and burstiness. In Fig. 7 we evaluate the *Social Media* pipeline on 2 traces derived from real workloads studied in [13]. The **Planner** finds a *5x cheaper* initial configuration than coarse-grained provisioning (Fig. 7(a)). Both systems achieve near-zero SLO miss rates throughout most of the workload, and when the big spike occurs we observe that InferLine's **Tuner** quickly reacts by scaling up the pipeline as described in §5. As soon as the spike dissipates, InferLine scales the pipeline down to maintain a cost-efficient configuration. In contrast, the coarse-grained tuning mechanism operates much slower and, therefore, is ill-suited for reacting to rapid changes in the request rate of the arrival process.

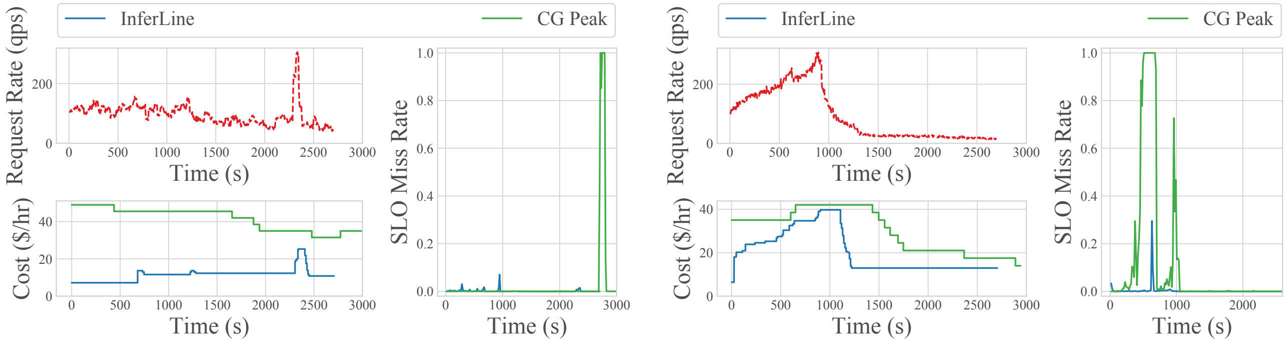
In Fig. 7(b), InferLine scales up the pipeline smoothly and recovers rapidly from an instantaneous spike, unlike the CG baseline. As the workload drops quickly after 1000 seconds, InferLine rapidly responds by shutting down replicas to reduce cluster cost. In the end, InferLine and the coarse-grained 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 tuning algorithms on a set of synthetic workloads with increasing arrival rates in Fig. 8. We observe that the traffic envelope monitoring described in §5 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 baseline 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. 7 but only after suffering a period of 100% SLO miss rate.



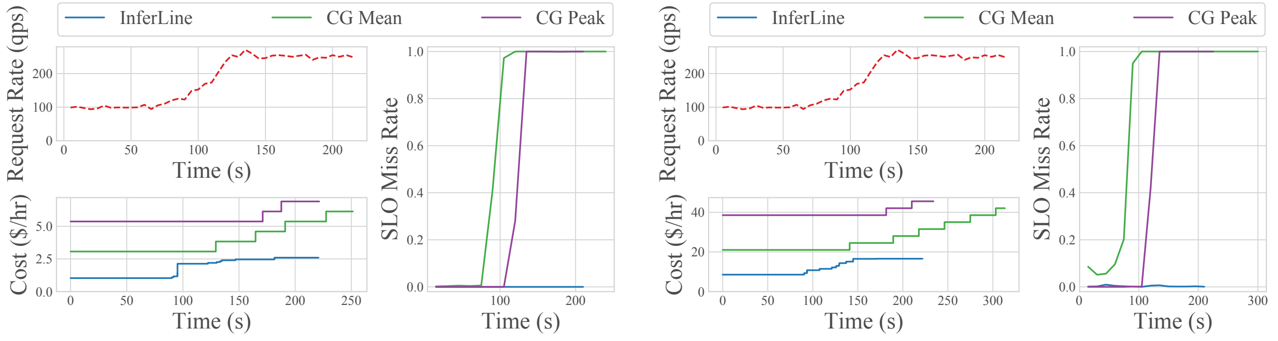
(a) Image Processing CV_A^2 1.0 (b) Image Processing CV_A^2 4.0 (c) Video Monitoring CV_A^2 1.0 (d) Video Monitoring CV_A^2 4.0

Figure 6: Comparison of InferLine's Planner 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.



(a) AutoScale Big Spike (b) AutoScale Slowly Varying

Figure 7: Performance comparison of the high-frequency tuning algorithms on traces derived from real workloads [13]. Both workloads were evaluated on the Social Media pipeline with a 150ms SLO. In Fig. 7(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. 7(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.



(a) Image Processing (b) Social Media

Figure 8: Performance comparison of the high-frequency tuning 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.

7.2 Sensitivity Analysis

We evaluate the sensitivity and robustness of the **Planner** and the **Tuner**. We analyze the accuracy of the **Estimator** in estimating tail latencies from the sample trace and the **Planner**'s response to varying arrival rates, latency SLOs, and burstiness factors. We also analyze the **Tuner**'s sensitivity to changes in the arrival process and ability to re-scale individual pipeline stages to maintain latency SLOs during these unexpected changes to the workload.

Planner Sensitivity: We first evaluate how closely the latency distribution produced by the **Estimator** reflects the latency distribution of the running system in Fig. 9. We observe

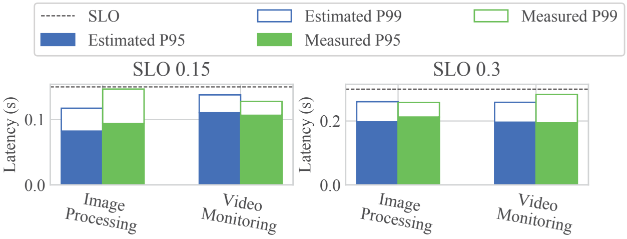


Figure 9: Comparison of estimated and measured tail latencies. We compare the latency distributions produced by the Estimator on a workload with λ of 150 qps and CV_A^2 of 4, observing that in all cases the estimated and measured latencies are both close to each other and below the latency SLO.

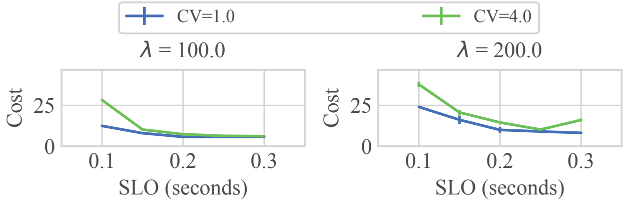


Figure 10: Planner sensitivity: Variation in configuration cost across different arrival processes and latency SLOs for the Social Media pipeline. We observe that 1) cost decreases as SLO increases, 2) burstier workloads require higher cost configurations, and 3) cost increases as λ increases.

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. 6 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. 10. First, increasing burstiness (from CV_A^2=1 to CV_A^2=4) requires more costly configurations as the Planner provisions more capacity to ensure that transient bursts do not cause the queues to diverge more than the SLO allows. We also see the cost gap narrowing between CV_A^2=1 and CV_A^2=4 as the SLO increases. As the SLO increases, additional slack in the deadline can absorb more variability in the arrival process and therefore fewer pipeline replicas are needed to process transient bursts within the SLO. 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, as more queries require more replicas.

Tuner Sensitivity: A common type of unpredictable behavior is a change in the arrival rate. We compare the behavior of InferLine with and without its Tuner enabled as the arrival rate changes from the planned-for 150 QPS to 250 QPS. 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 pipeline with only the Planner enabled but given full oracle knowledge of the arrival trace, and (b) a pipeline with only the Planner enabled and provided only the sample planning trace. Neither of these baselines responds to changes in workload during live serving. As we see in Fig. 11, InferLine continues to meet the SLO, and increases the cost of the pipeline only for the duration of the unexpected burst. The oracle 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 configure batch size and hardware type along with replication factor. But it pays this cost for the entire duration of the workload. The Planner without oracular knowledge starts missing latency SLOs as soon as the ingest rate increases as it is unable to respond to unexpected changes in the workload without the Tuner.

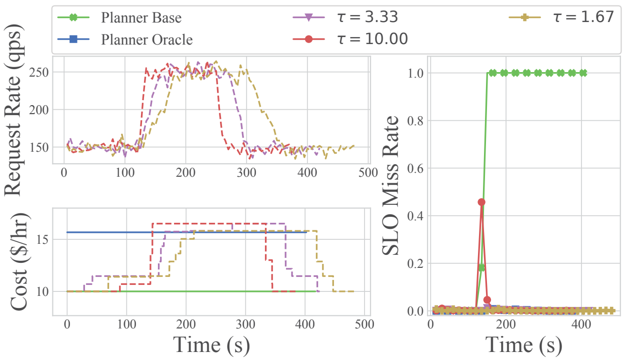


Figure 11: Sensitivity to arrival rate changes (Social Media pipeline). We observe that the Tuner quickly detects and scales up the pipeline in response to increases in λ. Further, the Tuner finds cost-efficient configurations that either match or are close to those found by the Planner given full oracle knowledge of the trace.

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 λ. This change is also harder to detect, as common practice is to look at moments of the arrival rate distribution, such as the mean or 99%. In Fig. 12 we show that **Tuner** is able to *detect* deviation from expected arrival burstiness and react to meet the latency SLOs by employing the traffic-envelope detection mechanism described in §5.

7.3 Attribution of Benefit

InferLine benefits from (a) low-frequency planning and (b) high-frequency tuning. Thus, we evaluate the following comparison points: baseline coarse grain planning (Baseline Plan), InferLine's planning (InferLine Plan), InferLine planning with baseline tuning (InferLine Plan + Baseline Tune), and

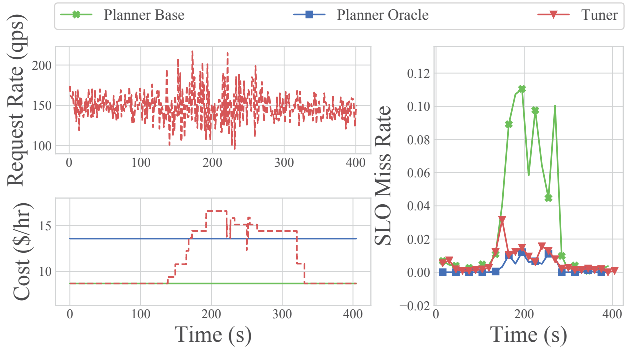


Figure 12: **Sensitivity to arrival burstiness changes (Social Media Pipeline). We observe that the network-calculus based detection mechanism of the Tuner detects changes in workload burstiness and takes the appropriate scaling action to maintain a near-zero SLO miss rate.**

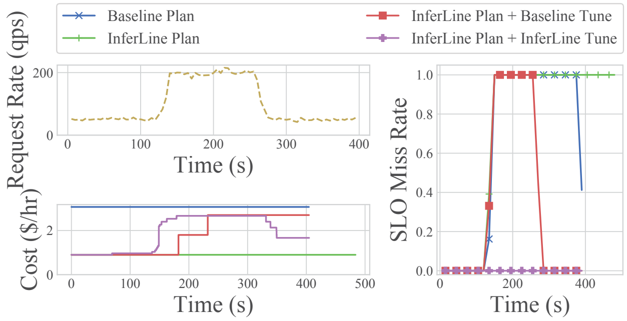


Figure 13: **Attribution of benefit between the InferLine low-frequency Planner and high-frequency Tuner on the Image Processing pipeline. We observe that the Planner finds a more than 3x cheaper configuration than the baseline. We also observe that InferLine's Tuner is the only alternative that maintains the latency SLO throughout the workload.**

InferLine planning with InferLine tuning (InferLine Plan + InferLine Tune), building up from pipeline-level configuration to the full feature set InferLine provides. InferLine's Planner reduces the cost of the initial pipeline configuration by more than 3x (Fig. 13), but starts missing latency SLOs when the request rate increases. Adding the baseline tuning mechanism (InferLine Plan + Baseline Tune) adapts the configuration, but too late to completely avoid SLO misses, although it recovers faster than planning-only alternatives. The InferLine Tuner has the highest SLO attainment and is the only alternative 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 Tuner to promptly and cost-efficiently adapt to unexpected workload changes.

7.4 Multiple Prediction-Serving Frameworks

The contributions of this work generalize to different underlying serving frameworks. Here, we evaluate the InferLine **Planner** running on top of both Clipper and TensorFlow Serving (TFS). In this experiment, we achieve the same low latency SLO miss rate for both prediction-serving frameworks. This indicates the generality of the planning algorithms used to configure individual models in InferLine. In Fig. 14 we show both the SLO attainment rates and the cost of pipeline provisioning when running InferLine on the two serving frameworks. The cost for running on TFS is slightly higher due to some additional RPC serialization overheads not present in Clipper.

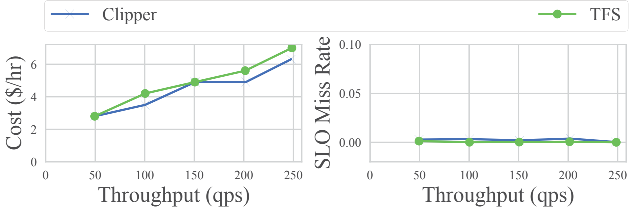


Figure 14: **Comparison of the InferLine Planner provisioning the TF Cascade pipeline in the Clipper and TensorFlow Serving (TFS) prediction-serving frameworks. The SLO is 0.15 and the CV_A^2 is 1.0.**

8 RELATED WORK

A number of recent efforts study the design of generic prediction serving systems [4, 9, 36, 37]. TensorFlow Serving [37] is a commercial grade prediction serving system primarily designed to support prediction pipelines implemented using TensorFlow [35], but does not provide any automatic provisioning or support latency constraints. Clipper adopts a containerized design allowing each model to be individually managed, configured, and deployed in separate containers, but does not support prediction pipelines or reasoning about latency deadlines across models.

Several systems have explored offline pipeline configuration for data pipelines [6, 16]. However, these target generic data streaming pipelines. They use black box optimization techniques that require running the pipeline end-to-end to measure the performance of each candidate configuration. InferLine instead leverages performance profiles of each stage and a simulation-based performance estimator to explore the configuration space without needing to run the pipeline.

Dynamic pipeline scaling is a critical feature in data streaming systems to avoid backpressure and over-provisioning. Systems such as [11, 18] are throughput-oriented with the goal of maintaining a well-provisioned system under changes in the request rate. The DS2 autoscaler in [18] estimates true processing rates for each operator in the pipeline 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, [11] identifies a single bottleneck stage at a time, taking several steps to converge to a well-provisioned system. Both systems provision for the average ingest rate and ignore any burstiness 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).

A few streaming autoscaling systems consider latency-oriented performance goals [12, 21, 22]. The closest work to InferLine, [21] treats each stage in a pipeline as a single-server queueing system and uses queueing theory to estimate the total queue waiting time of a job under different degrees of parallelism. They leverage this queueing model to greedily increase the parallelism of the stage with the highest queue waiting time until they can meet the latency SLO. However, their queueing model only considers average latency and ignores tail latency. InferLine's Tuner automatically provisions for worst-case latencies.

SEDA [39] studies dynamically controlling pipeline systems connected by queues. However, SEDA focuses on managing multi-threaded single-process systems using techniques such as thread pool management and adaptive load-shedding to remain performant even when overloaded. In contrast, InferLine is a distributed system that maintains end-to-end latency objectives and scales resource usage to avoid overload.

VideoStorm [40] explores the design of a streaming video processing system with 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.

Nexus [31] configures DNN inference pipelines for video-streaming applications. Similar to InferLine, it uses model profiles to understand model batching behavior and provisions pipelines for end-to-end latency objectives. However, they do not configure which hardware to use, instead assuming a homogeneous GPU cluster. They also rely on admission control to reject queries while InferLine's Tuner quickly re-scales the pipeline to maintain SLOs without rejecting queries.

A large body of prior work leverages profiling for scheduling, including recent work on workflow-aware scheduling [17, 29]. 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 [13] surveys 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 processing. However, we find that the InferLine Tuner outperforms the coarse-grain baselines using the Autoscale mechanism on both latency SLO attainment and cost (§7.1).

9 LIMITATIONS

One assumption in the Planner is that the available hardware has a total ordering of latency across all batch sizes. As specialized accelerators for ML 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 Planner to account for this batch-size dependent ordering.

A second assumption is that the inference latency of ML models is independent of their input. There are emerging classes of ML tasks [27, 28] where state-of-the-art models have inference latency that varies based on the input. One way 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 as the processing time in the estimator, which will lead to feasible but more costly configurations.

10 CONCLUSION

In this paper we studied the problem of provisioning and managing prediction pipelines to meet end-to-end tail latency requirements at low cost and across heterogeneous parallel hardware. We introduced InferLine- -a system which efficiently provisions prediction pipelines subject to end-to-end latency constraints. InferLine combines a low-frequency Planner that finds cost-optimal configurations with a high-frequency Tuner that rapidly re-scales pipelines to meet latency SLOs in response to changes in the query workload. The low-frequency Planner combines profiling, discrete event simulation, and constrained combinatorial optimization to find the cost minimizing configuration that meets the end-to-end tail latency requirements without ever instantiating the system (§4). The high-frequency Tuner uses network-calculus to quickly autoscale each stage of the pipeline to accommodate changes in the query workload (§5). In combination, these components achieve the combined effect of *cost-efficient* heterogeneous prediction pipeline provisioning that can be deployed to a variety of prediction-serving frameworks 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 provisioning alternatives.

11 ACKNOWLEDGEMENTS

In addition to NSF CISE Expeditions Award CCF-1730628 and NSF CAREER Award 1846431, this research is supported by gifts from Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

REFERENCES

[1] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2015. Deep Compositional Question Answering with Neural Module Networks. *CoRR* abs/1511.02799 (2015). arXiv:1511.02799 http://arxiv.org/abs/1511.02799

[2] Anelia Angelova, Alex Krizhevsky, Vincent Vanhoucke, Abhijit S Ogale, and Dave Ferguson. 2015. Real-Time Pedestrian Detection with Deep Network Cascades. *BMVC* (2015), 32.1–32.12.

[3] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. 2003. The Nonstochastic Multiarmed Bandit Problem. *SIAM J. Comput.* 32, 1 (Jan. 2003), 48–77. https://doi.org/10.1137/S0097539701398375

[4] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. 2017. 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)*. ACM, 1387–1395.

[5] Andrew Beck. 2008. Simulation: the practice of model development and use. *Journal of Simulation* 2, 1 (01 Mar 2008), 67–67. https://doi.org/10.1057/palgrave.jos.4250031

[6] Muhammad Bilal and Marco Canini. 2017. Towards Automatic Parameter Tuning of Stream Processing Systems. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/3127479.3127492

[7] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Wadsworth Publishing Company, Belmont, California, U.S.A.

[8] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 401âĂŞ412. https://doi.org/10.14778/2735496.2735503

[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] flink 2020. Apache Flink. https://flink.apache.org.

[11] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-regulating Stream Processing in Heron. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1825–1836. https://doi.org/10.14778/3137765.3137786

[12] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2017. DRS: Auto-Scaling for Real-Time Stream Analytics. *IEEE/ACM Trans. Netw.* 25, 6 (Dec. 2017), 3338–3352. https://doi.org/10.1109/TNET.2017.2741969

[13] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4, Article 14 (Nov. 2012), 26 pages. https://doi.org/10.1145/2382553.2382556

[14] Jiaqi Guan, Yang Liu, Qiang Liu, and Jian Peng. 2017. Energy-efficient Amortized Inference with Cascaded Deep Classifiers. *arXiv.org* (Oct. 2017). arXiv:1710.03368v1 [cs.LG]

[15] Amazone Web Services Inc. 2017. Amazon SageMaker: Developer Guide. *http://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf* (2017).

[16] Pooyan Jamshidi and Giuliano Casale. 2016. An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems. *MASCOTS* (2016), 39–48.

[17] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 117–134. http://dl.acm.org/citation.cfm?id=3026877.3026887

[18] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. 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* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 783–798. http://dl.acm.org/citation.cfm?id=3291168.3291226

[19] Jean-Yves Le Boudec and Patrick Thiran. 2001. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg.

[20] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. 2010. A Contextual-bandit Approach to Personalized News Article Recommendation. In *WWW*.

[21] B. Lohrmann, P. Janacik, and O. Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 399–410. https://doi.org/10.1109/ICDCS.2015.48

[22] BjÃűrn Lohrmann, Daniel Warneke, and Odej Kao. 2013. Nephele Streaming: Stream Processing Under QoS Constraints At Scale. *Cluster Computing* 17 (08 2013). https://doi.org/10.1007/s10586-013-0281-8

[23] Mateusz Malinowski, Marcus Rohrbach, and Mario Fritz. 2015. Ask Your Neurons: A Neural-Based Approach to Answering Questions about Images. *2015 IEEE International Conference on Computer Vision (ICCV)* (2015), 1–9.

[24] Mason McGill and Pietro Perona. 2017. Deciding How to Decide: Dynamic Routing in Artificial Neural Networks. *arXiv.org* (March 2017). arXiv:1703.06217v2 [stat.ML]

[25] openalpr 2020. Open ALPR. https://www.openalpr.com/.

[26] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[27] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[28] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR* abs/1506.01497 (2015). arXiv:1506.01497 http://arxiv.org/abs/1506.01497

[29] Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. 2017. Enabling Workflow-Aware Scheduling on HPC Systems. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (Washington, DC, USA) *(HPDC '17)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/3078597.3078604

[30] samza 2020. Apache Samza. http://samza.apache.org.

[31] 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)*. ACM, New

York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[32] Evan Sparks. 2016. *End-to-End Large Scale Machine Learning with KeystoneML*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-200.html

[33] storm 2020. Apache Storm. http://storm.apache.org.

[34] Yi Sun, Xiaogang Wang, and Xiaoou Tang. 2013. Deep Convolutional Network Cascade for Facial Point Detection. *CVPR* (2013), 3476–3483.

[35] tensorflow 2020. TensorFlow. https://www.tensorflow.org.

[36] tensorrtserver 2020. TensorRT Inference Server. https://github.com/NVIDIA/tensorrt-inference-server.

[37] tfserving 2020. TensorFlow Serving. https://tensorflow.github.io/serving.

[38] timely 2020. Timely Dataflow. https://github.com/TimelyDataflow/timely-dataflow.

[39] Matt Welsh, David E Culler, and Eric A Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SOSP* (2001), 230–243.

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