Inferline:

Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines

By Arkash Jain

Introduction

Prediction serving systems optimize single model serving, leaving cross-model interaction and configurations to meet application requirements to the developer.

Multiple Models	Hardware Heterogeneity	Variable Batch Sizes
Cloud applications rely on ML inference over multiple models linked in a dataflow DAG (like Amazon's Alexa, case study in future slide)	Hardware heterogeneity makes assigned tasks in each stage of the pipeline cumbersome. Each model must be configured to query a certain batch size for optimal performance.	Per-stage decisions with respect to hardware and batch size affect latency contributed by each stage towards the end-to-end pipeline latency, bound by the application SLO.

There is a need for a system to automate pipeline provisioning and configuration subject to tail-latency SLOs in a cost effective manner.

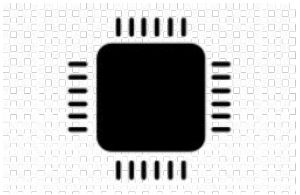
Motivation

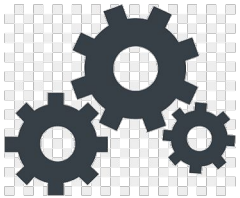
Sparse literature exists on tackling combinatorially large configuration spaces, stochastic and bursty workloads and queuing delays altogether.

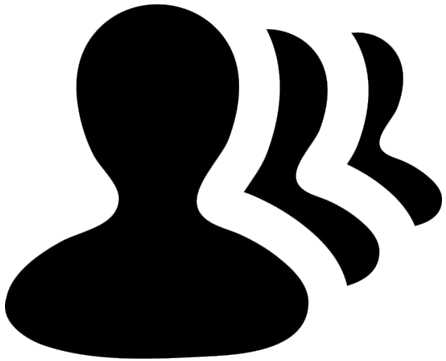
Large Configuration Space

Queuing Delays

Unpredictable Workloads



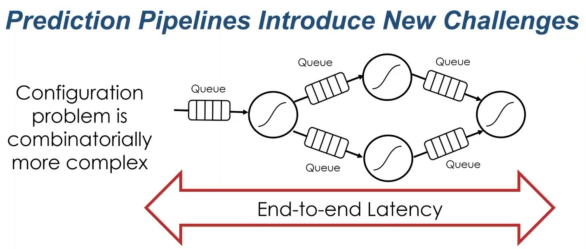




Configuration Space

- ML models can be parallelized to maximize throughput but not all are suitable for hardware accelerators.

- Many cases involved setting batch sizes which increases latency. In vector parallelization, the first query isn't returned until the last one.
- To bound query latency, setting a max. batch size becomes important but is challenged by the hardware type and model.



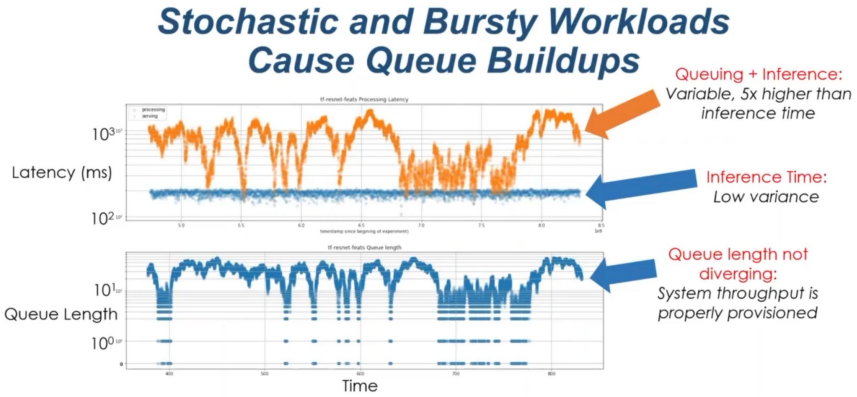
- In heavy query loads, operators need to be set and changed in conditional logic flow.
- There needs to be a low cost method for fine-tuning these operators.

- **Allocating parallel resources to single models leds to a trade off because if you increase the throughput on one model by trading latency, you reduce the latency budget for other models.**
- **End-to-End Latency is thus impacted.**

Queuing Delay and Stochastic Workloads

Queuing

- Use queues because each stage operates at a different speed.
- It absorbs inter-arrival processing irregularities as well and needs to be set explicitly.



Stochastic Workloads

- Systems must respond to bursty, stochastic query streams, characterized by their average arrival rate λ and their coefficient of variation, a dimensionless measure of variability defined below.

$$CV_A^2 = \sigma^2/\mu^2$$

- Processes with higher *CVA*2 have higher variability and require additional over-provisioning to meet latency objectives (expensive).
- Changing workloads require mechanisms to monitor and *tune* individual stages in the pipeline.

System Design

Inferline uses a low-frequency planner and a high frequency tuner.

- To deploy a new prediction pipeline, 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

Planner

- During first time planning, the **Profiler** creates performance profiles of all individual models referenced by the driver program, capturing model throughput as a function of hardware type and maximum batch size.
- Per-iteration, the Planner uses model profiles to select a cost-minimizing step while relying on the **Estimator** to check for latency constraint violations. After deployment, the Planner is re-run periodically (hours to days) to find cost optimal configuration

Tuner

- **Tuner** runs as a standalone process by observing 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 Planner

Profiler

- **Creates performance profiles** for each model in the pipeline as a function of batch size and hardware.
- Starts with **executing the sample set of queries** on the pipeline to generates input data for profiling individual components. The scale factor s (frequency of queries) to each model is the conditional probability that a model will be queried given a query entering the pipeline.
- Profiling **one replica is sufficient** since models scale horizontally and is reused in subsequent runs of the Planner.

Estimator

- Responsible for **rapidly estimating the end-to-end latency** of a given pipeline configuration for the sample query trace. Combines output with profile information to tell simulator running time of a model of a certain batch size.
- Implemented as a **continuous-time, discrete-event simulator (of deterministic behavior), simulating the entire pipeline**. This simulator maintains a global logical clock, advancing discrete events in temporal order.

Planning Algorithm

- The planning algorithm is an **iterative constrained optimization** procedure that **greedily minimizes cost** while ensuring that the latency constraint is satisfied.
- Algorithm 1 - finds a feasible initial configuration that meets the latency SLO **while ignoring cost**
- Algorithm 2 - **greedily modifies the configuration** to reduce the cost while using the Estimator to identify and reject configurations that violate the latency SLO

High Frequency Tuner

- While the planner finds a config for the sample planning workload, changing arrival rates and variability lead to either over provisioning or missing SLOs.
- The tuner constantly runs to maintain the latency objective and minimize costs by scaling actions.
- A traffic envelope for a workload is constructed by sliding a window of size ΔTi over the workload's inter-arrival process and capturing the maximum number of queries seen anywhere within this window

- $x = \Delta Ti \leftrightarrow y = qi$ (# queries) for all *x* over the duration of a trace, capturing how much the workload can burst in any given interval of time.
- The x-axis is discretized by setting the smallest ΔTi to *Ts* (service time of the system), and then double the window size up to 60 seconds.
- For each such interval, the maximum arrival rate *ri* for this interval can be computed as $ri = qi$. By measuring *ri* across all ΔTi *simultaneously*, ΔTi captures 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
- The planner constructs the sample trace for the traffic envelope

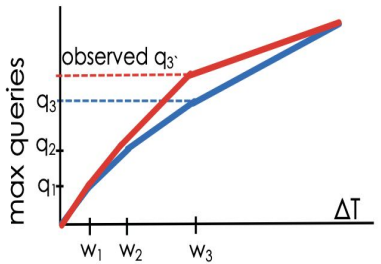
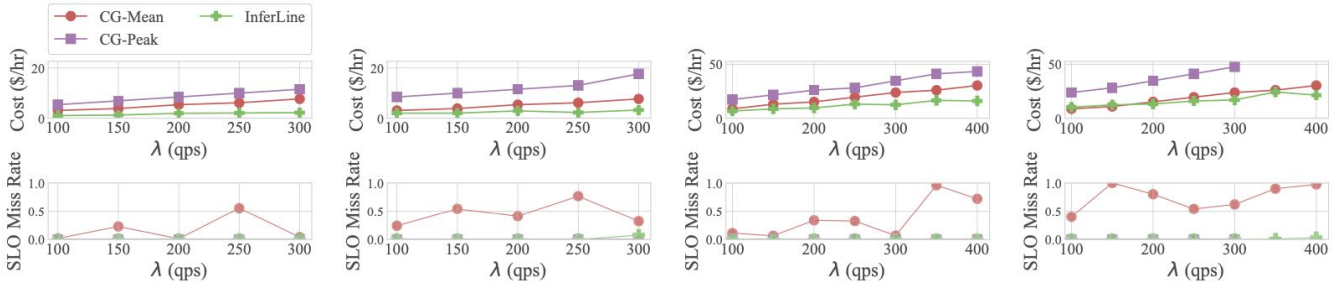


Figure 5: Observed traffic envelope exceeds sample envelope. The observed traffic envelope (in red) exceeds the sample trace traffic envelope (in blue) at window w_3, triggering the Tuner to scale up the pipeline. The pipeline will be re-scaled for the new arrival rate $r_{max} = \frac{q_{3'}}{w_3}$.

Conclusion

- 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
- InferLine has a *34.5x lower SLO miss rate* than the baseline.



(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.

Appendix

Planning Algorithms (#'s 1 & 2)

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;
```

- Batch size = 1 using lowest latency hardware available (lines 2-5).
- If sum of processing latencies of all models > slo, planner terminates (lines 6-7).
- Otherwise, throughput bottleneck is determined and model replication is increased until it is no longer a bottleneck.

- If the pipeline is initialize properly, the cost minimizing process considers 3 optimizations - increased batch size, removing replicas and downgrading hardware (lines 2-5)
- It applies each one till it finds the best cost algorithm
- Batch Size - increasing doesn't affect cost but increases throughput and decreases latency.
- Removing replicas and using downgraded hardware reduces costs

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;
```

Scaling Algorithms (#'s 3 and 4)

Failing to scale down = higher costs, whereas failing to scale up = missing latency objectives.

Algorithm 3: Reactively scale up the pipeline

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

- By continuously computing the traffic envelope, the tuner gets the set of arrival rates, which trigger rescaling if necessary
- It also knows the current workload rate r_max.
  - If the overall λ of the workload has not changed but it has become burstier this will be a rate computed with a smaller Δ*Ti*,
  - and if the burstiness of the workload is stationary but the λ has increased, this will be a rate with a larger Δ*Ti*.
- The tuner finds the # of replicas needed for rescaling (lines 9-19)

- InferLine takes a conservative approach to scaling down the pipeline to prevent unnecessary configuration oscillation which can cause SLO misses.
- It waits for a period of time after any configuration changes to allow the system to stabilize and uses a delay of 15 seconds
- Once the time elapses the tuner continuously computes the max request rate λ*new* that has been observed over the last 30 seconds, using 5 second windows and computes the number of replicas

Algorithm 4: Reactively scale down the pipeline

```
1  Function CheckScaleDown():
2      if (Now() - LastUpdate) > 15 then
3          λ_new = Max(RecentLambdas);
4          foreach model in Pipeline do
5              k_m = λ_new * model.scalefactor /
                 (model.throughput * ρ_min);
6              extraReps = model.replicas - Ceil(k_m);
7              if extraReps > 0 then
8                  RemoveReps(model, extraReps);
```

Related Work

- The closest work to InferLine, 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.

Collection of screenshots

Low-Frequency Planner

1. **Profiler:** Profile each model in pipeline individually
   - Understand ***inference*** performance under different configurations

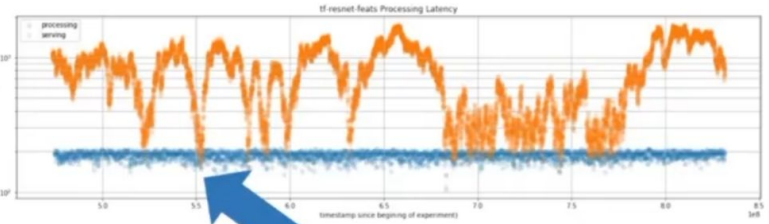
2. **Latency Estimator**: Use profiles and sample trace in discrete event simulator that estimates end-to-end pipeline latency
   - Model ***queuing behavior*** and effects of variable batch sizes

3. **Planner:** Solve a constrained optimization problem to find cost-minimizing pipeline configuration that meets SLOs

Model Profiles: Batch Size

Model inference time depends on **batch size**

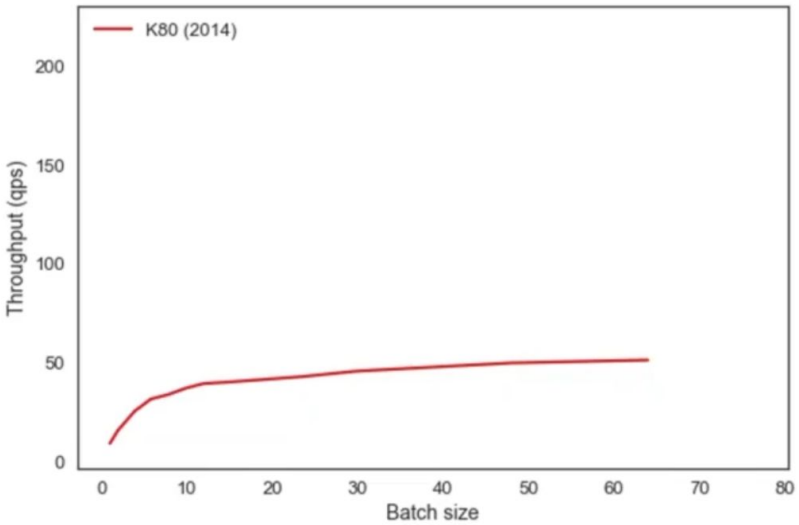
Latency (ms)



Time

Inference Time:
Low variance

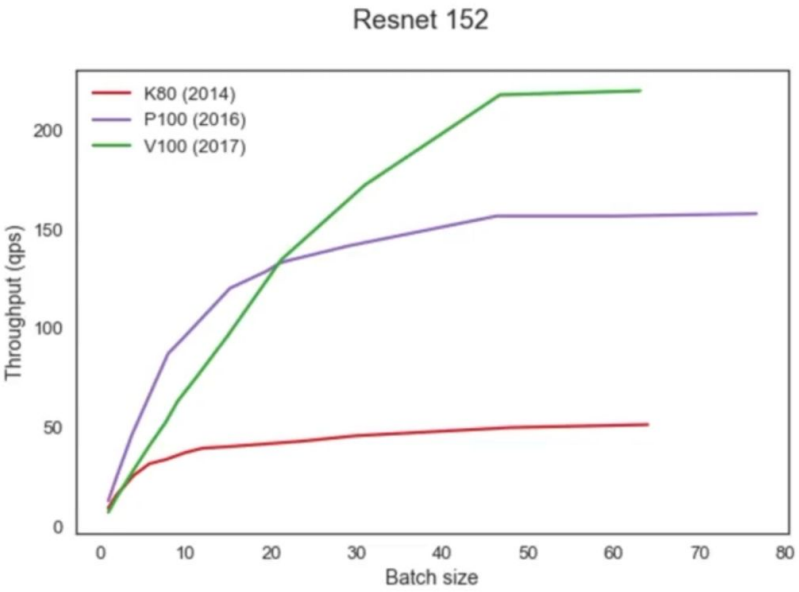
Resnet 152



Throughput increases as function of batch size with diminishing returns

Model Profiles: Hardware

Model inference time depends on batch size and **compute hardware**



GPUs

TPUs

CPUs

Custom ASICS

Optimization

Three candidate configuration changes:

- Increase Batch Size
- Reduce Replication Factor
- Downgrade Hardware

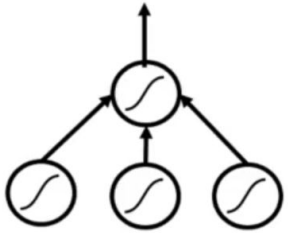
In each iteration:

- Evaluate the three configuration changes on each model
- Use simulation-based latency estimator to eliminate configurations that violate latency constraint
- Greedily minimize cost by taking the lowest cost candidate configuration
  - Applies one config change to one model per iteration

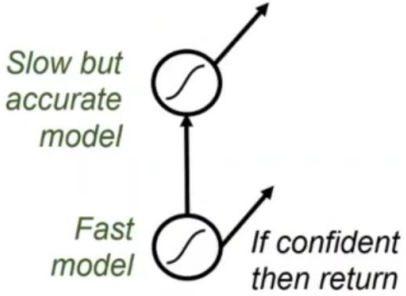
High-Frequency Tuner: Overview

1. Runs continuously during serving
2. Network-calculus traffic envelopes detect workload changes across many timescales simultaneously
3. Offline profiles determine how much to scale each model
4. Delay downscaling actions to avoid oscillation

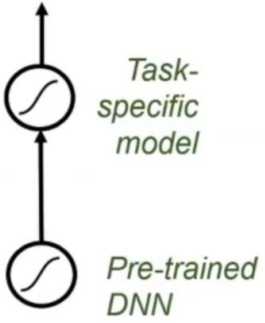
Prediction Pipelines



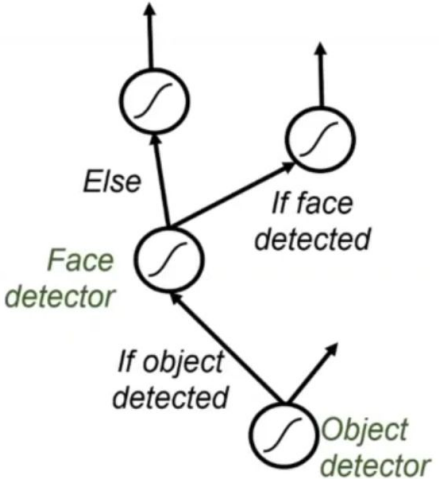
Ensembles can improve accuracy



Faster inference with prediction cascades



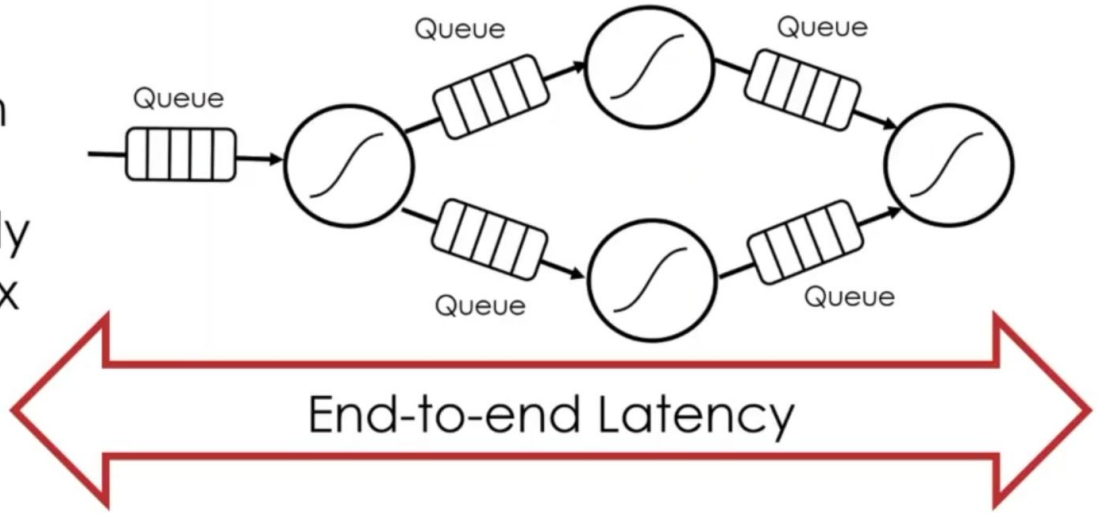
Faster development through model-reuse



Model specialization

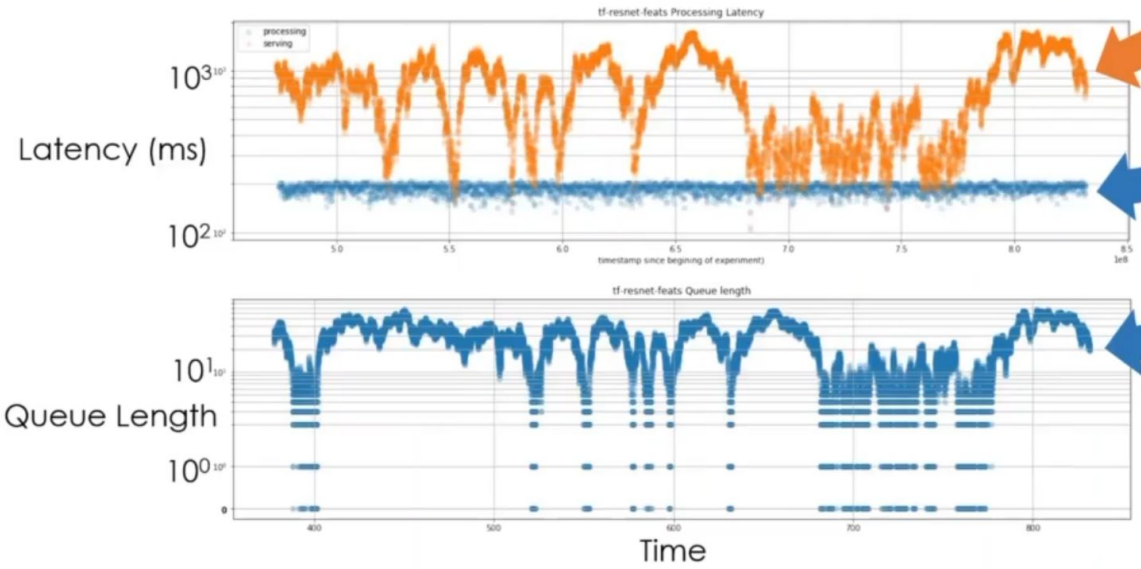
Prediction Pipelines Introduce New Challenges

Configuration problem is combinatorially more complex



Must jointly configure all models in pipeline to achieve end-to-end tail latency SLOs

Stochastic and Bursty Workloads Cause Queue Buildups



Queuing + Inference:
Variable, 5x higher than inference time

Inference Time:
Low variance

Queue length not diverging:
System throughput is properly provisioned

InferLine Inputs and Outputs

- User provides:
  - Inference Pipeline
  - Latency service level objective(SLO)
  - Sample Workload Request Trace

- InferLine controls 3 parameters per-model:
  - Batch size
  - Compute Resource Type
  - Number of Replicas (Replication Factor)