CATO: End-to-End Optimization of ML-Based Traffic Analysis Pipelines

Gerry Wan[1] Shinan Liu[2] Francesco Bronzino[3] Nick Feamster[2] Zakir Durumeric[1]
[1]*Stanford University* [2]*University of Chicago* [3]*École Normale Supérieure de Lyon*

Abstract

Machine learning has shown tremendous potential for improving the capabilities of network traffic analysis applications, often outperforming simpler rule-based heuristics. However, ML-based solutions remain difficult to deploy in practice. Many existing approaches only optimize the predictive performance of their models, overlooking the practical challenges of running them against network traffic in real time. This is especially problematic in the domain of traffic analysis, where the efficiency of the serving pipeline is a critical factor in determining the usability of a model. In this work, we introduce CATO, a framework that addresses this problem by jointly optimizing the predictive performance and the associated systems costs of the serving pipeline. CATO leverages recent advances in multi-objective Bayesian optimization to efficiently identify Pareto-optimal configurations, and automatically compiles end-to-end optimized serving pipelines that can be deployed in real networks. Our evaluations show that compared to popular feature optimization techniques, CATO can provide up to 3600× lower inference latency and 3.7× higher zero-loss throughput while simultaneously achieving better model performance.

1 Introduction

Machine learning (ML) models have grown to outperform traditional rule-based heuristics for a variety of traffic analysis applications, such as traffic classification [36, 44], intrusion detection [73], and QoE inference [15, 47]. Over the past few years, researchers have explored various approaches to developing more accurate models, ranging from better feature selection to employing sophisticated model types and traffic representations [4, 12, 25, 29, 37, 46, 51, 55, 57, 62, 76, 81]. However, the predictive performance of ML-based solutions often overshadows an equally critical aspect—the end-to-end efficiency of the serving pipeline that processes network traffic and executes the model.

For traffic analysis, a significant challenge lies not just in developing accurate models, but in meeting the performance demands of the network. Many network applications must operate in real time with sub-second reaction times and/or process hundreds of gigabits per second of traffic without packet loss [72]. Unfortunately, models developed without consideration of the associated systems costs of serving them in real networks often turn out to be unusable in practice [14]. Current approaches to this problem typically rely on lightweight models [43], programmable hardware [36, 74], or early inference techniques [11, 54], but many of these unnecessarily compromise on predictive performance [16, 64, 74].

Recent studies have stressed the need to balance both the systems costs and predictive performance of ML-based traffic analysis solutions [14, 64]. However, achieving this balance is difficult. The end-to-end latency and throughput of a serving pipeline, which includes packet capture, feature extraction, *and* model inference, are difficult to approximate without in-network measurements. Furthermore, the search space over optimal feature representations is exponential in the number of candidate traffic features, and also depends on how far into a flow to wait before making a prediction. The added complexity of not just considering one objective, but two, makes end-to-end optimization of such systems an open challenge.

In this work, we present CATO, a generalizable framework that systematically optimizes the systems costs and model performance of ML-based traffic analysis pipelines. We start by formalizing the development of ML models for traffic analysis as a *multi-objective* optimization problem. We then combine multi-objective Bayesian optimization, tailored specifically for traffic analysis, with a realistic pipeline profiler to efficiently construct end-to-end optimized traffic analysis pipelines. CATO simultaneously searches over the selected features and the amount of captured traffic needed to compute those features, factors which have been shown to significantly impact both efficiency and predictive performance [14, 34]. During this search, CATO performs direct end-to-end measurements on the resulting serving pipelines to both *optimize* and *validate* their in-network performance.

We evaluate CATO on live network traffic and offline traces across a range of classification and regression traffic analy-

sis tasks. Our experimental results show that compared to popular feature optimization methods, CATO can reduce the end-to-end latency of the serving pipeline by up to 3600×, from several minutes to under 0.1 seconds, while simultaneously improving model performance. Additionally, CATO can increase zero-loss classification throughput by up to 3.7×.

We hope that our work helps to realize the potential impact of using machine learning to manage and improve networks. Code is available at: https://github.com/stanford-esrg/cato.

2 Background and Motivation

The networking community has long attempted to use machine learning (ML) to perform traffic analysis tasks like QoE inference [15, 26, 39, 47, 48], traffic classification [36, 44, 65], intrusion detection [73], and load balancing [18]. As traffic increasingly becomes encrypted, ML has also been shown to be a promising technique for understanding otherwise opaque network traffic, replacing traditional deep packet inspection and other rule-based heuristics [12, 53, 63].

While significant progress has been made in improving the predictive capabilities of models used for traffic analysis [12], the real-world deployability of a model is based not just on conventional notions of ML performance (e.g., accuracy, F1 score), but also on the associated systems-level performance (e.g., latency, throughput) of the entire serving pipeline [14]. Given the real-time demands of network operations, any applications that rely on ML must nonetheless operate within tight performance budgets. Even small delays can cause substantial packet loss and render a model ineffective [15, 31], making systems performance even more crucial for traffic analysis. As a result, ML-based traffic analysis cannot solely target high predictive performance—the end-to-end efficiency of the entire serving pipeline must be jointly optimized as well.

2.1 ML-Based Traffic Analysis

ML-based traffic analysis typically begins with the ingestion of raw traffic and ends with a prediction of a traffic property, such as a service quality metric. While traffic analysis applications are diverse, we focus on the class of problems that involves per-flow or per-connection inference, such as traffic/device classification, QoE inference, or intrusion detection. These applications typically make a prediction about an entire flow or connection, then initiate an action such as triggering an alert, blocking or rerouting the flow, or performing further analysis downstream (Figure 1).

Traffic inference extends beyond merely executing the model; it also involves packet capture, connection tracking, flow reassembly, and feature extraction. Raw traffic undergoes multiple operations, including header parsing, computation, and encoding before arriving at the representation that is used as input to the model. The final model inference step makes the prediction, with its predictive performance determined

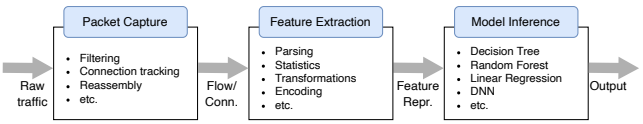


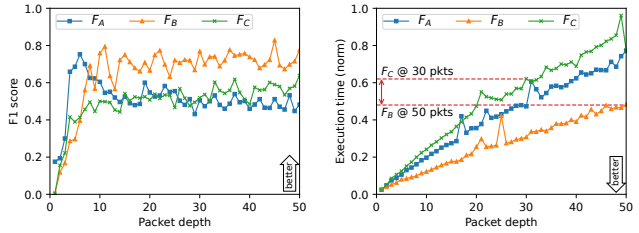
Figure 1: A typical serving pipeline for ML-based traffic analysis. Usability of a model hinges on both its predictive accuracy and the systems performance of the entire pipeline.

by the model type (e.g., random forest, neural network), the computed features, and the amount of data captured from the flow. The end-to-end systems performance depends on *all* of these aspects together.

For traffic analysis in particular, the choice of features computed from the network traffic is often as important, if not more so, than the model itself [29, 34, 55]. While many previous works have focused on the model inference stage [43, 46, 57, 64, 74, 81], design decisions made in the earlier stages of the serving pipeline are crucial to its performance and practicality, and warrant careful consideration.

Optimizing Predictive Performance. Many techniques have been proposed to accurately make predictions about network traffic. These approaches range from popular feature selection methods that choose highly predictive features based on summary statistics [51], packet lengths [11], timing [37], and/or frequency domain characteristics [25], to new techniques like GGFAST [55], which generates specialized "snippets" to classify encrypted flows. Following advancements in domains like computer vision and natural language processing, network researchers have also proposed sophisticated deep learning models [46, 57, 81] and traffic representations [29, 62] designed to further optimize the predictive performance of traffic analysis applications. However, most of these machine learning techniques are evaluated using offline packet traces on metrics like accuracy, precision, recall, or F1 score, overlooking the need to both *optimize* and *validate* their in-network systems performance. As a result, ML-based traffic analysis solutions that have been demonstrated to have high accuracy in controlled laboratory experiments often turn out to be unusable in real-time deployments because of the systems costs associated with running them [14].

Optimizing Serving Efficiency. To address the systems requirements of traffic analysis on modern networks, some works propose using lightweight models [43, 74] or choose features that reduce model inference time [69]. However, these techniques can over-compromise on predictive performance for speed, and often overlook the efficiency of other pieces of the serving pipeline like packet capture and feature extraction. Other approaches aim to optimize serving efficiency by making predictions as early as possible [10, 11, 21, 60]. While many traffic analysis solutions implicitly rely on the entire network flow or connection [7, 9, 15, 40, 62], these "early inference" techniques make predictions after observing a predefined number of packets.



(a) Packet Depth vs. F1 Score. The best feature sets differ at varying packet depths.

(b) Packet Depth vs. Exec. Time. It can be cheaper to extract low-cost features at greater depths.

Figure 2: Effects of different (feature set, packet depth) configurations on F1 score and execution time. We highlight the size and complexity of the search space.

However, there is no ideal *packet depth* (i.e., the number of packets to use from any given flow) that is universally effective across applications. Choosing an appropriate value typically requires prior domain-specific knowledge or resorting to manual trial-and-error [54]. Consequently, existing works that do explicitly choose a packet depth often opt for values such as 10, 50, or 100 packets with little justification [5, 29, 30, 45, 50, 55, 56, 64]. As we will show in Section 5.2, this approach can miss significant opportunities for gains in both efficiency and model performance.

2.2 Challenge of End-to-End Optimization

Despite progress towards individually optimizing the predictive performance or serving efficiency of ML-based traffic analysis, improvements to either area in isolation often result in solutions that fail to achieve optimal combined model and systems performance objectives. Systematically designing traffic analysis pipelines that jointly optimize both of these objectives remains an open challenge.

We illustrate this challenge by attempting to design an IoT device classifier using the dataset published by Sivanathan et al. [65]. Our goal is to construct a serving pipeline that is *Pareto-optimal* across both its execution time and F1 score. In other words, it should not be possible to further reduce the execution time without also reducing the F1 score, or vice versa. We combine techniques from prior work by experimenting with different features and early inference, both of which have been shown (and we confirm) to significantly impact multiple aspects of the pipeline, including predictive performance, model inference time, and feature extraction time [14, 34, 51, 54]. For this example, we choose from six candidate flow features (Appendix A, Table 4) and vary the packet depth collected in each flow from 1 to 50, which is consistent with values used in prior work [11, 45, 54, 55]. We train the model, compile the complete serving pipeline, and exhaustively measure the F1 score and execution time for all $2^6 \times 50 = 3{,}200$ (feature set, packet depth) combinations.

As seen in Figure 2, F1 score and execution time vary with the chosen features and packet depth. For readability, we plot only three (labeled $F_A/F_B/F_C$) out of the 64 possible feature sets, but find qualitatively similar results across those not shown. We can see in Figure 2a that the best feature sets by F1 score differ dramatically at different packet depths. While F_A has the highest F1 score within the first 10 packets, the ranking flips at higher packet counts. Interestingly, the predictive performance of F_B and F_C increase with packet depth, whereas the opposite is true for F_A. In Figure 2b, we observe that for the *same* feature set, execution time generally increases with packet depth. However, the overall cost of waiting 50 packets to extract F_B is lower than the cost of waiting 30 packets for F_C. This reveals that having the flexibility to optimize the timing of feature extraction can significantly enhance serving efficiency, but is not always as straightforward as simply minimizing analyzed packets. If we look across both figures, we see that over-optimizing on execution time can also adversely affect F1 score and vice versa. The non-linear trade-offs between objectives further highlight the challenge in identifying Pareto-optimal solutions without exhaustive measurement.

The trade-offs between predictive performance and systems costs form a multi-dimensional and multi-objective search space that extends beyond merely identifying which features result in more accurate models. It also includes considerations for features that are efficient to extract, as well as how much data must be captured to compute and represent the features. While exhaustive measurement of end-to-end systems costs and model performance is feasible for just six candidate features, it quickly becomes impractical when scaling up to the dozens to hundreds of flow features typically considered by developers. In our example, it took 5 days to train, compile, and measure all 3,200 serving pipelines, but it would take over 7,000 years with 25 candidate features. The size and complexity of the search space, coupled with the need to consider and validate both model performance and systems cost objectives, makes end-to-end optimization challenging. Addressing this challenge is the central contribution of our work.

3 Cost-Aware Traffic Analysis Optimization

We introduce CATO (Cost-Aware Traffic Analysis Optimization), our solution for cost-aware ML-based traffic analysis optimization. The goal of CATO is to automatically construct traffic analysis pipelines that jointly minimize the end-to-end systems costs of model serving while maximizing predictive performance. At its core, CATO combines a multi-objective Bayesian optimization-guided search with a novel pipeline generator and feature representation profiler to produce serving pipelines suitable for deployment in real networks.

Symbol	Description
$\mathcal{F}$	Set of candidate network flow features
N	Maximum connection depth
$\mathcal{P}(\mathcal{F})$	Power set of $\mathcal{F}$
$\mathbb{X}$	Search space defined as $\mathbb{X} = \mathcal{P}(\mathcal{F}) \times N$
F	Set of features in a feature representation
n	Connection depth from which F is extracted
x	Feature representation $x = (F, n)$
$\texttt{cost}(x)$	Systems cost objective function
$\texttt{perf}(x)$	Predictive performance objective function
Γ	Set of Pareto-optimal solutions

Table 1: Summary of Variables

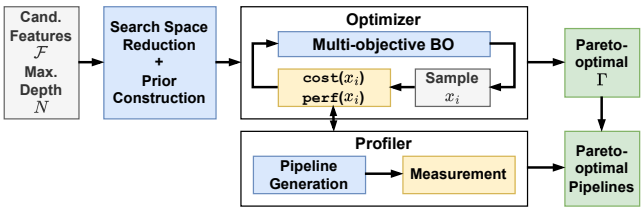


Figure 3: CATO combines a multi-objective BO-based Optimizer and a realistic pipeline Profiler to construct and validate efficient ML-based traffic analysis serving pipelines.

3.1 Problem Definition

CATO takes as input a set of candidate network flow features, denoted by $\mathcal{F}$. In line with conventional machine learning practice, these are typically derived from domain expertise or determined by the capabilities of the traffic collection tool. Common examples for traffic analysis include mean packet size, bytes transferred, connection duration, etc. [51], but can also include more complex features like frequency domain [25,77] and application-layer characteristics [15]. CATO also takes as input a maximum connection depth $N \in \mathbb{R}$, which serves as an *upper bound* on the amount of data in the connection that is considered for inference. Concretely, this can be the number of initial packets (i.e., packet depth), bytes, or time into the connection prior to feature extraction. These inputs define the *search space* $\mathbb{X} = \mathcal{P}(\mathcal{F}) \times N$, where $\mathcal{P}(\mathcal{F})$ is the power set of $\mathcal{F}$, i.e., the set of all possible subsets of $\mathcal{F}$, from which CATO selects feature representations.

A *feature representation* $x = (F, n)$ consists of a set of features $F \subseteq \mathcal{F}$ and a value $n \leq N$ that indicates the connection depth from which F is extracted. Each feature representation gives rise to a serving pipeline with an associated end-to-end systems cost and predictive performance, denoted by the functions $\texttt{cost}(x) : \mathbb{X} \to \mathbb{R}$ and $\texttt{perf}(x) : \mathbb{X} \to \mathbb{R}$. We note that these functions are general and can be user-defined according to the specific objectives of the traffic analysis problem. For instance, $\texttt{cost}(x)$ can refer to the end-to-end inference latency, execution time, (negative) throughput, etc., while $\texttt{perf}(x)$ can be defined as F1 score, accuracy, (negative) mean-squared-error, etc. We list a summary of variables in Table 1.

Multi-objective Optimization. We formalize the development of ML models for traffic analysis as a multi-objective optimization over the search space $\mathbb{X}$. The aim is to identify the Pareto front $\Gamma \subseteq \mathbb{X}$, which consists of all non-dominated points in $\mathbb{X}$. In other words, Γ contains the maximally desirable feature set / connection depth configurations, where no further improvement in systems cost or model performance can be achieved without compromising the other objective.

We deliberately choose a multi-objective optimization over a single-objective approach. Unlike a single-objective problem that aims to maximize model performance while satisfying system constraints, or vice versa, a multi-objective solution offers several advantages. The first is that the exact system and model performance requirements may not be known a priori (e.g., due to variable traffic rates or shared system resources), making it difficult to precisely define constraints. Second, if requirements change, a single-objective approach would necessitate redefining and rerunning the optimization with new objectives and constraints [68]. By expressing the problem as a multi-objective optimization, CATO identifies multiple Pareto-optimal solutions that each achieve a different balance between systems cost and model performance, providing the flexibility to accommodate changing application needs (e.g., adjusting accuracy thresholds or imposing new latency constraints) without re-optimization.

3.2 CATO Overview

CATO constructs end-to-end optimized traffic analysis pipelines according to the systems cost and model performance objective functions. It does so by efficiently identifying Pareto-optimal feature representations, and generating ready-to-deploy serving pipelines for a given model from those feature representations. Figure 3 depicts the high-level design, which consists of the Optimizer and the Profiler:

- The Optimizer takes the set of candidate features and maximum connection depth, and performs a multi-objective Bayesian optimization-guided search over the feature representation space. It periodically queries the Profiler for the systems costs and model performance of its sampled feature representations, which it uses to further refine the search for the Pareto front.
- The Profiler accepts queries from the Optimizer, compiles binaries for the end-to-end serving pipeline, and runs them to accurately measure $\texttt{cost}(x)$ and $\texttt{perf}(x)$. These measurements serve the dual purpose of guiding the Optimizer towards Pareto-optimal solutions and validating the in-network performance of the generated pipelines.

3.3 The CATO Optimizer

In general, measuring $\texttt{cost}(x)$ and $\texttt{perf}(x)$ for an arbitrary feature representation is computationally expensive. It involves generating the serving pipeline, training and evaluating the ML model, and measuring performance costs either through simulation or in physical testbeds. The massive size of the search space and the computational cost of evaluating

the objective functions precludes the possibility of exhaustively searching all possible configurations. To handle this intractability, CATO leverages Bayesian Optimization (BO), building on recent developments in multi-objective design space exploration [52] and sample-efficient BO [32] to efficiently estimate the Pareto front in $\mathbb{X}$.

Why Bayesian Optimization? Bayesian optimization is a technique designed for global optimization of black-box objective functions [35], and has seen success in domains like hyperparameter tuning [66, 70], compiler optimization [28, 52], and robotics [17]. It is particularly useful for expensive-to-evaluate, non-linear objectives, as in our case with $\texttt{cost}(x)$ and $\texttt{perf}(x)$. Moreover, the discontinuous nature of these objective functions make the use of traditional gradient-based or linear optimizers a poor fit for our problem.

BO works by building a probabilistic surrogate model for the objective function(s), and uses it to make decisions about which points (e.g., feature representations) in the search space to evaluate next. Typically, BO begins by sampling an initial number of points at random to build the surrogate model. Each subsequent iteration involves generating a set of candidate points, using the surrogate model to predict the objective function's output on the candidate set, and choosing the next "best" (e.g., maximizing Expected Improvement) point to evaluate using the real objective function. The surrogate model is then updated to include the newly evaluated point, and the process repeats until some stopping criteria is met, such as a maximum number of iterations or a satisfactory level of convergence [24, 35]. While BO is not guaranteed to work well in high-dimensional, multi-objective spaces [61], we describe how we augment it for our context later in this section. We compare the performance of BO against other search techniques in Section 5.3 and show that our approach is efficient at approximating the Pareto front.

BO Formulation. We formulate CATO's search process as a multi-objective Bayesian optimization problem with $|\mathcal{F}|+1$ dimensions: one dimension per feature in $\mathcal{F}$ and one for the connection depth n. Each feature parameter is represented by a binary indicator variable, which denotes whether or not the feature is included. The connection depth parameter is separately encoded as an integer or real-valued variable upper bounded by the maximum connection depth N. This setup lets CATO *concurrently* search over both features and connection depth while optimizing for systems cost and model performance. We define this search as a minimization of the two functions $\texttt{cost}(F,n)$ and $-\texttt{perf}(F,n)$. These objective functions are managed by the Profiler (Section 3.4), which generates complete serving pipelines according to the feature representations sampled by the Optimizer, and returns the end-to-end systems cost and model performance metrics.

Tailoring BO for Traffic Analysis. In its basic form, BO has several limitations. Conventional applications of BO typically involve single-objective, low-dimensional (fewer than 20) search spaces [24, 61]. However, our traffic analysis problem is inherently multi-objective, high-dimensional, and involves a complex search space with mixed categorical (features) and numerical (connection depth) variables. To address this, we augment the CATO Optimizer with two preprocessing techniques to improve its sample efficiency (Figure 3). The first is a dimensionality reduction step that strategically discards candidate features that are unlikely to improve the model's predictive performance regardless of its impact on the end-to-end systems costs. By default, we exclude features with a mutual information [71] score of zero, which indicates no direct informational relationship with the target variable.

The second technique incorporates prior probabilities into the BO formulation, accelerating the search by providing the Optimizer with "hints" about the approximate locations of Pareto-optimal feature representations. To account for both objectives, CATO constructs two sets of priors: one over the feature space that targets $\texttt{perf}(x)$, and one over the connection depth that targets $\texttt{cost}(x)$. The set of priors over the feature space encodes each feature's relative contribution to the model's performance, and are derived from the mutual information scores computed in the dimensionality reduction step. Formally, we define the prior probability of whether a feature f is part of a Pareto-optimal feature representation $x=(F,n)$ as $\mathbb{P}(f \in F | x \in \Gamma) = (1-\delta)\frac{I(f)}{I_{max}} + \frac{\delta}{2}$, where $I(f)$ represents the mutual information of f with respect to the target variable, I_{max} is the maximum mutual information among all candidate features, and δ is a damping coefficient. The damping coefficient is used to adjust the priors to prevent the feature with the highest mutual information from always being included. $\delta = 0$ signifies no damping, while $\delta = 1$ results in uniform priors for all features. These probabilities encourage CATO to more frequently explore regions of the search space that include features with higher predictive power.

The prior over the connection depth is represented by a probability mass function that decays linearly as the connection depth increases. The rationale is that for the same feature set, waiting longer to capture more packets or bytes before feature extraction correlates with worse systems performance. This prior encourages CATO to more frequently explore representations that require fewer packets or less data, despite not requiring domain-specific knowledge about the optimal connection depth at which to collect features. As we will show in Section 5.5, CATO is robust to reasonably large connection depth ranges.

We emphasize that these preprocessing steps can be performed efficiently without needing to evaluate the objective functions. Despite the term "prior," no prior knowledge about the optimal features or connection depth needs to be supplied by the user. CATO automatically derives the priors used to accelerate its search, thereby streamlining its usability.

3.4 The CATO Profiler

The CATO Profiler evaluates the feature representations sampled by the Optimizer based on the concrete definitions of $\texttt{cost}(x)$ and $\texttt{perf}(x)$. To accomplish this, it generates code for the packet capture and feature extraction stages of each sampled point, trains the model, and runs the full serving pipeline to *directly measure* its end-to-end systems costs and model performance. This measurement serves two purposes: (1) guiding the search process of the Optimizer, and (2) validating the in-network performance of identified solutions.

Why Measure? Using heuristics to estimate the end-to-end systems cost of a traffic analysis pipeline is difficult. Much like how existing heuristics that approximate model performance often fail to capture interdependencies and correlations among features [6, 34, 71], systems cost heuristics can similarly fail to capture the complexities of packet capture and feature extraction. Traffic analysis is particularly sensitive to this, since the processing steps during feature extraction often overlap in non-trivial ways. For instance, computing mean TCP window size and the number of ACKs sent in a connection require parsing each packet down to its TCP header, a shared task that must be factored into the end-to-end cost. Likewise, computing the mean window size also involves calculating its sum, the latter of which can then essentially be used for free. Other factors like resource contention and characteristics of the network traffic (e.g., bursty vs. non-bursty) can also unpredictably affect the end-to-end systems cost [58].

We argue that rather than trying to model or predict these complex systems-level interactions, it is both more accurate and useful to perform direct measurement. With direct measurement, CATO captures the actual end-to-end cost of the serving pipeline, encompassing all critical components including packet capture, feature extraction, and the model inference itself. Accurate measurement not only helps the Optimizer make well-informed decisions, but also helps users build confidence in validating whether identified solutions are operationally viable. Although this approach can be computationally expensive, the cost of training the model, generating the full serving pipeline, and measuring its performance is balanced by the sample efficiency of the Optimizer.

Pipeline Generation. To evaluate different feature representations during the search process, we require an *automated* way to measure $\texttt{cost}(x)$ and $\texttt{perf}(x)$ for any $x \in \mathbb{X}$. With a search space size of $O(2^{|\mathcal{F}|} \times N)$, manually implementing packet capture, feature extraction, and model inference for each evaluated point is impractical. One approach that enables flexible evaluation is "runtime branching," which uses branching logic at runtime to determine which paths in the code should be executed to extract a given feature representation. However, runtime branching introduces additional overhead that can contaminate the cost measurements of performance-sensitive traffic analysis pipelines. Instead, CATO employs *conditional compilation* to build and run customized end-to-end serving pipelines tailored to each configuration. The resulting binary matches the performance of a manually implemented pipeline, containing only the set of operations needed to collect traffic data up to the specified connection depth, extract the corresponding features, and execute the model inference. This technique not only constructs fully operational traffic analysis pipelines, but also provides the flexibility to accurately measure any point in the search space.

Pipeline Measurement. CATO presents a testbed interface that replicates a real-world deployment scenario of the pipeline. For model performance measurements, the Profiler trains a fresh model for each representation sampled by the Optimizer and directly measures its predictive performance to account for any interaction effects between features. The final performance metric is derived from a hold-out test set. We note that CATO operates on pre-labeled datasets, meaning it does not focus on automatic labeling or ground truth generation. The framework is designed to optimize model accuracy and system performance based on this labeled input. For systems cost measurements, CATO either simulates traffic inputs from the training data, or, when feasible, deploys the full serving pipeline in its target network environment (e.g., a passive monitoring or bump-in-the-wire deployment model) for end-to-end measurements. While each measurement can be expensive, the Optimizer is intentionally designed to minimize the number of measurements needed to approximate the Pareto front. We report wall-clock times for several of our evaluated use cases (Section 5.1) in Appendix E.

4 Implementation of CATO

We detail our implementation of CATO, covering the Optimizer, Profiler, model training, and objective functions.

Bayesian Optimization. We implement the CATO Optimizer using HyperMapper [52], a Bayesian optimization framework for design space exploration. HyperMapper supports multi-objective optimization with mixed-variable search spaces, but is not tailored specifically for high-dimensional BO. We use πBO [32] for prior injection, but adapt its implementation to incorporate CATO-generated priors in multi-objective use cases. We use a random forest as the surrogate model, which has been shown to perform well compared to more traditional Gaussian processes for discontinuous and non-linear objective functions [52]. The prior over the packet depth is constructed using the Beta distribution with $\alpha = 1$ and $\beta = 2$. We initialize the Optimizer with three iterations of random search space exploration and choose $\delta = 0.4$ based on empirically tuned values (Section 5.5).

Pipeline Generation. The CATO Profiler generates serving pipelines using a modified version of Retina [72], a Rust framework that compiles traffic *subscriptions* into efficient packet processing pipelines. A subscription defines the rules for how incoming traffic should transformed into a specific

```
1  fn on_packet(&mut self, packet: Packet) {
2      #[cfg(any(feature="iat_sum"))]
3      {
4          let pkt_timestamp = packet.timestamp();
5          self.iat_sum += pkt_timestamp - last_timestamp;
6          let last_timestamp = pkt_timestamp;
7      }
8      #[cfg(any(feature="ttl_min",feature="winsize_max"))]
9      let eth = packet.parse_eth();
10     #[cfg(any(feature="ttl_min",feature="winsize_max"))]
11     let ipv4 = eth.parse_ipv4();
12     #[cfg(any(feature="ttl_min"))]
13     self.ttl_min = self.ttl_min.min(ipv4.ttl());
14     #[cfg(any(feature="winsize_max"))]
15     {
16         let tcp = ipv4.parse_tcp();
17         self.winsize_max = self.winsize_max.max(tcp.winsize());
18     }
19 }
20
21 fn extract(&mut self) -> Vec<f64> {
22     vec![
23         #[cfg(feature="iat_sum")]
24         self.iat_sum,
25         #[cfg(feature="ttl_min")]
26         self.ttl_min,
27         #[cfg(feature="winsize_max")]
28         self.winsize_max,
29     ]
30 }
```

Figure 4: An example portion of the CATO Profiler's template subscription module. Each operation is predicated on its associated features and conditionally compiled with the `cfg` macro. For instance, if the evaluated feature set consists of `ttl_min` and `winsize_max`, then only lines 9, 11, 13, 16, and 17 will execute on each new packet, and only those two features will be extracted in the final feature representation. This enables dynamic cost profiling that matches the characteristics of a manually implemented feature extraction stage.

representation, and invokes a callback on the returned data. We implement the model inference stage in the callback, and subscribe to a template feature representation that can be modified at compile-time to the specific representation being evaluated. To dynamically generate the custom packet capture and feature extraction stages, we create a Retina subscription module that implements the processing steps needed to extract *all* candidate features. Each operation (e.g., parse an IPv4 header, add to a cumulative sum of packet inter-arrival times, etc.) is then annotated with a configuration predicate that specifies the subset of features necessitating its execution. If the feature representation being evaluated contains at least one of the predicated features, the predicate evaluates to true and the operation is conditionally compiled into the binary. This technique avoids redundant computation in shared steps, such as parsing headers, and ignores operations associated with features that are not included. For packet capture, we annotate the subscription with an early termination flag that stops data collection once the connection depth is reached.

Figure 4 shows pseudo-code for an example portion of the template subscription module. We implement 67 candidate features (Appendix A, Table 4) in 1,600 lines of Rust code. We note that the chosen candidate features are not specific to any use case: they are widely used in traffic analysis applications and are common features exposed by open source tools [3, 15, 16, 23, 51, 64, 72, 77]. We use number of packets into the connection to measure connection depth.

We note that our Profiler implementation uses Retina [72] to target commodity servers. However, CATO's core design principles remain applicable to optimizing and validating hardware-based traffic analysis pipelines, which we discuss further in Section 6.

Model Training. As an optimization framework, CATO is general to the specific type of model used in the traffic analysis pipeline. We implement support for three model types: decision trees (DT), random forests (RF), and deep neural networks (DNN). For DT and RF, we use scikit-learn's DecisionTreeClassifier and RandomForestClassifier, with 5-fold nested cross validation and grid search for hyperparameter tuning. We tune the maximum tree depth from 3–20 and set the number of estimators to 100 for RF. To match the speed of the Rust-based feature extraction stage, we retrain the best-performing DT and RF models in Rust using the SmartCore [1] library and evaluate the final Rust model on a hold-out test set containing 20% of the data.

For DNN, we implement a fully connected feedforward neural network in TensorFlow, consisting of three hidden layers with ReLU activation and L2 regularization. We apply dropout to prevent overfitting and use the Adam optimizer for training. Since Rust lacks mature DNN libraries, we train and evaluate the DNN models entirely in Python/TensorFlow. Additional details are provided in Appendix C.

Objective Functions. We use end-to-end inference latency, zero-loss classification throughput, and pipeline execution time as three different metrics for systems cost. End-to-end inference latency measures the duration from the arrival of the first packet in the connection to the model's final prediction. This includes the time spent extracting features from raw traffic, the model inference time, and time spent waiting for packets to arrive. Zero-loss throughput is the highest ingress traffic rate that can be sustained by the serving pipeline with no packet drops, which we negate to match the sign of $\texttt{cost}(x)$ minimization. The execution time measures the total CPU time spent in the serving pipeline, excluding time between packets. This metric is less dependent on the specific characteristics of the input traffic, and is an indirect measure of both latency and throughput. Although these can be combined into a single cost metric, we evaluate them separately to show CATO's flexibility. Depending on the traffic analysis use case, which we detail in the next section, we use either the F1 score or root-mean-squared error, calculated from the predictions on the hold-out test set as the model performance metric.

5 Evaluation

We evaluate CATO over a variety of configurations and use cases. Section 5.1 details our datasets and testbeds. In Section 5.2, we show that CATO can help traffic analysis applications achieve substantially lower inference latency and higher

Use Case	Type	Traffic	Model
app-class	Classification	Live	Decision Tree
iot-class	Classification	Dataset	Random Forest
vid-start	Regression	Dataset	Deep Neural Network

Table 2: Evaluation Use Cases

throughput without compromising model performance, and in many cases improve upon both metrics. We also compare it with Traffic Refinery [14], a recent system for cost-aware ML on network traffic. Section 5.3 compares the efficiency of the CATO Optimizer to alternative Pareto-finding approaches, and Section 5.4 performs an ablation study of the Profiler. In Section 5.5, we run micro-benchmarks on CATO's sensitivity to various search space sizes and hyperparameters.

5.1 Datasets and Testbeds

We consider three use cases in our evaluations: web application classification (app-class), IoT device recognition (iot-class), and video startup delay inference (vid-start). These are typical analysis tasks of varying complexity that are representative of the type of ML-based inference performed on network traffic. Table 2 summarizes them, with more details provided in Appendix B.

Web Application Classification. While open-source traffic classification datasets exist, replaying them at modern line rates is challenging without duplicating flows. To evaluate serving pipelines against real traffic at high speeds, we develop a use case that identifies one of six common web applications from live traffic on a large university network. This type of classification is typically used by web application firewalls or in the early stages of network QoE inference pipelines [2, 15]. For ground truth, we label connections using the server name in the TLS handshake. We train and evaluate decision tree models using flow statistics captured from the network, then deploy them to the same network for real-time serving using Retina [72].

IoT Device Recognition. To help make our results reproducible, we also consider an IoT device recognition use case based on the dataset published by Sivanathan et al. [65]. We use a random forest to classify connections as belonging to one of 28 IoT device types. Although real-time throughput experiments are not feasible without duplicating flows, we use this dataset to report micro-benchmarks and evaluate CATO's ability to approximate the true Pareto front.

Video Startup Delay Inference. We further demonstrate CATO's generalizability to different traffic analysis tasks and model types through a regression use case that predicts the startup delay of video streams. Startup delay inference is widely used in analysis of encrypted video traffic as a measure of QoE [8, 15, 49]. We choose startup delay (rather than other QoE metrics) to provide a regression task that complements the previous two classification use cases. We also adopt a more complex DNN instead of a tree-based model, using the YouTube dataset published by Bronzino et al. [15].

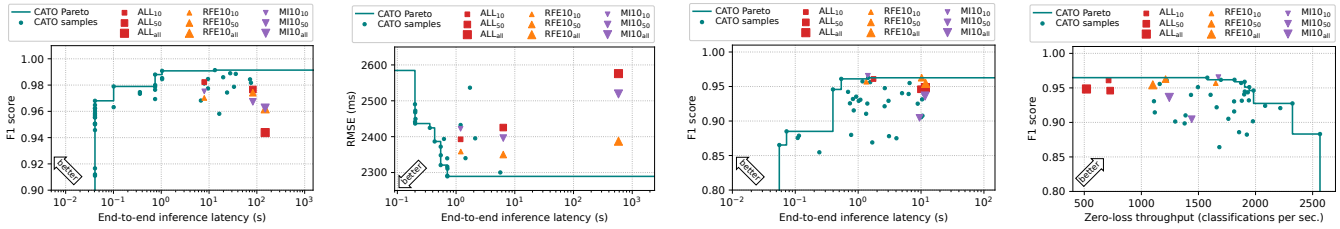
5.2 Model Serving Performance

We first examine the end-to-end inference latency, zero-loss throughput, and predictive performance of CATO-optimized serving pipelines. Note that CATO itself is not a classifier, but a general framework for optimizing ML-based serving pipelines for real-time traffic analysis. Therefore, instead of directly comparing CATO with existing classifiers or models, we evaluate it against optimization strategies commonly used in prior work to build those models. We use the following feature optimization methods and combine them with early inference techniques as our baselines:

- ALL: Use all available features.
- RFE10: Select the top ten features by recursive feature elimination [27]. RFE trains a model using all available features, then iteratively removes the least important feature and retrains until the desired number remains.
- MI10: Select the top ten features based on mutual information [71]. This is a model agnostic algorithm that measures how much information each feature contributes to the target variable and picks the most relevant ones.

Prior traffic analysis solutions typically wait until the end of the connection before making a prediction [7, 9, 15, 40, 62] or use a fixed packet depth for early inference [5, 10, 11, 29, 45, 50, 56]. For example, Peng et al. [54] collects up to the first 10 (including TCP handshake) packets, while recent work like GGFAST [55] use the first 50. For a thorough analysis, we compare against these strategies by running each baseline at packet depths of 10, 50, and all packets. CATO does not assume a predefined optimal packet depth, but searches over the entire feature representation space as part of its optimization process. We choose a maximum packet depth of 50 and run for 50 iterations, which is consistent with common machine learning practices [13, 32]. We show how CATO reacts to different packet depth ranges in Section 5.5.

End-to-End Inference Latency. Figures 5a, 5b, and 5c show the end-to-end inference latency and predictive performance (F1 score or RMSE) for iot-class, vid-start, and app-class, respectively. Each CATO sample represents a candidate point explored during the optimization process, with CATO's Pareto front constructed from the set of non-dominated points. For iot-class and vid-start, all points on CATO's Pareto front dominate the baseline solutions, achieving equal or better predictive performance with lower end-to-end latency. For iot-class, CATO can reduce the inference latency by 11–79× compared to solutions that use the first 10 packets in the connection, 817–2000× compared to those that use the first 50, and over 3600× (from several minutes to under 0.1 seconds) compared to those that wait until the end of the connection. Likewise for vid-start, CATO generates solutions that can infer video startup delays in less than one second (a 2.2–2900× speedup, depending on



(a) `iot-class` latency (b) `vid-start` latency (c) `app-class` latency (d) `app-class` throughput

Figure 5: Comparison of F1 score / RMSE vs. end-to-end inference latency / zero-loss throughput (single-core) for `iot-class`, `vid-start`, and `app-class` serving pipelines. CATO identifies multiple solutions on its Pareto front that dominate those found by traditional optimization techniques, and can achieve significantly better systems and predictive performance.

baseline) while also reducing the mean squared error of its predictions.

Since end-to-end inference latency is largely dominated by packet inter-arrival times, this improvement can be attributed to CATO's ability to find alternative sets of features using the fewest packets necessary without compromising the predictive performance of these popular feature selection methods. For example, RFE10 using the first 10 packets (RFE10_{10}) in `iot-class` achieves an F1 score of 0.970 with an inference latency of 7.9 seconds. However, CATO identifies a different set of features using just the first 3 packets for a better F1 score of 0.979 and an inference latency of 0.1 seconds.

We find a similar pattern for `app-class`, where CATO-optimized pipelines outperform most baseline methods across both objectives. While MI10_{10} and RFE10_{50} achieve slightly higher F1 scores (0.963 and 0.962), CATO produces a solution with a nearly identical F1 score (0.960) and a latency of 0.54 seconds—2.6× and 19× faster than MI10_{10} and RFE10_{50}, respectively. These results reinforce that for traditional feature optimization methods, it is not always clear a priori which feature set at which packet depth results in the best model performance or serving efficiency. Through end-to-end optimization of both objectives over the entire feature representation space, CATO is able to automatically derive and validate the performance of faster and more accurate traffic analysis pipelines.

Zero-Loss Classification Throughput. We compare the predictive performance and classification throughput of solutions found by CATO with those found by the baseline methods for `app-class`. We exclude `iot-class` and `vid-start` due to limitations in replaying the traces at high speeds without repeating flows. For a realistic assessment, we use live traffic from our campus network, but restrict all experiments to a single core to avoid saturating our network's maximum ingress throughput. In an actual deployment scenario, the throughput can be easily scaled up by adding more cores, owing to the per-core scalability of Retina [72]. More details about our throughput experiments can be found in Appendix D.

Figure 5d shows that CATO's solutions outperform the baselines in both throughput and F1 score, with the exception

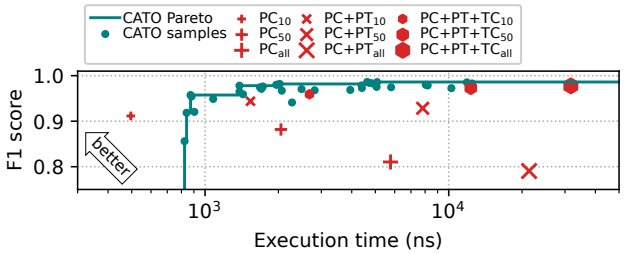


Figure 6: F1 score vs. pipeline execution time for `iot-class` using CATO and Traffic Refinery (in red). While PC_{10} achieves a strong trade-off due to its low cost, CATO more consistently finds solutions closer to the Pareto front.

of MI10_{10}. Despite this, CATO successfully identifies the feature representation with the highest overall F1 score and the one with the highest zero-loss throughput. For a decrease in F1 score from 0.96 to 0.93, CATO can increase throughput by 37%. Compared to solutions that wait until the end of the connection, CATO can improve the zero-loss throughput by a factor of 1.6–3.7×, and 1.3–2.7× for those that require the first 50 packets while also achieving higher model performance. Notably, CATO achieves these results after exploring just 50 feature representations out of $2^{67} \times 50 = 7 \times 10^{21}$ (67 candidate features, up to a maximum packet depth of 50).

Comparison with Traffic Refinery. We further compare CATO with Traffic Refinery [14], a recent traffic analysis framework that also facilitates joint evaluation of model performance and system costs. Unlike CATO, Traffic Refinery requires manual exploration of flow features and connection depth. We simulate Traffic Refinery's cost profiler using CATO's execution time cost metric and replicate its built-in packet counter (PC), packet timing (PT), and TCP counter (TC) feature classes. While Traffic Refinery defaults to making an inference after ten seconds into the flow, we evaluate it at packet depths of 10, 50, and all packets for consistency with the above baselines (see Appendix F for more details).

Figure 6 plots F1 score vs. execution time for `iot-class` using CATO and Traffic Refinery. The results show that Traffic Refinery's macro-aggregation of standard feature classes

is less efficient than CATO at finding optimal trade-offs. CATO achieves better accuracy at lower cost across all sampled points except PC_{10}. Even in this case, it is still able to identify an alternative representation with a similar F1 score and a modest 344 ns increase in pipeline execution time. While packet counters alone at a packet depth of 10 happen to perform well for `iot-class` at minimal cost, adding packet timing and TCP state information further improves accuracy but incurs a much higher execution time. Identifying such configurations with Traffic Refinery requires trial-and-error, as it is not clear which combinations perform best at which packet depths. In contrast, CATO more efficiently explores the feature representation space, clustering solutions closer to the Pareto front and more effectively balancing predictive performance and systems cost across a broader range of configurations.

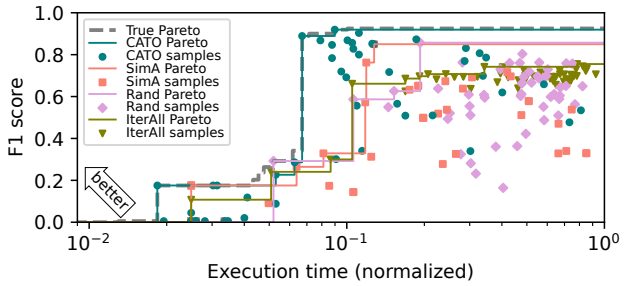


Figure 7: Estimated Pareto fronts after 50 iterations. CATO outperforms other Pareto-finding approaches, especially in regions with high F1 scores and low execution times. The sampled points illustrate the candidate solutions explored by each algorithm during the optimization process, while only the Pareto-optimal points are included in the final output.

5.3 Optimizer Efficiency

In this section, we evaluate the efficiency of the CATO Optimizer by measuring the quality of its computed Pareto front and the speed at which it converges to the true Pareto front. Since it is infeasible to exhaustively measure all points in $\mathbb{X}$ to obtain the true Pareto front for the default candidate feature set size of 67, we evaluate on a smaller candidate set of six features (Appendix A, Table 4) for the `iot-class` use case in order to obtain the ground truth. We compare the Optimizer with three alternative Pareto-finding algorithms:

- SIMA: Use simulated annealing [38], a metaheuristic for solving optimization problems with a complex search space. We provide details of our multi-objective implementation in Appendix G.
- RAND: Sample a random subset of features at a random packet depth without replacement.
- ITERALL: Use all available features but increment the packet depth on each iteration, starting from one.

We remark that these algorithms differ from the baseline methods described in the previous section in that they attempt to estimate the Pareto front rather than a single point solution. In N search iterations, each of these alternative approaches makes exactly N calls to $\texttt{cost}(x)$ and $\texttt{perf}(x)$.

Pareto Front Quality. We compare the quality of the Pareto front estimated by each algorithm using Hypervolume Indicator (HVI). HVI is a common metric used to compare multi-objective algorithm performance, and measures the area between the estimated Pareto front and the true Pareto front bounded by a reference point [41]. We use pipeline execution time as our chosen systems cost metric and F1 score for model performance. Since F1 score and execution time have different raw value ranges, we normalize the data before computing HVI to assign similar importance to both objectives.

We run each search algorithm for 50 iterations at a maximum packet depth of 50. In Figure 7, we plot the feature representations sampled at each iteration alongside the corresponding Pareto front constructed from the non-dominated points. These sampled points represent candidate solutions explored during optimization and do not directly impact the end-to-end traffic analysis pipeline. While many intermediate samples overlap between algorithms due to inherent randomness, they primarily serve to illustrate the regions explored by each algorithm and are not included in the final solution. For reference, we also show the true Pareto front computed from exhaustively measuring all $2^6 \times 50 = 3{,}200$ feature representations in the search space.

We observe that CATO's Pareto front closely approximates the true Pareto front while sampling less than 1.6% of the search space. Using a worst-case reference point (F1 score of 0 and normalized execution time of 1), CATO achieves an HVI of 0.98, compared to 0.88, 0.86, and 0.77 achieved by SIMA, RAND, ITERALL, respectively. We note that CATO does not entirely dominate all alternative search algorithms, especially around F1 scores of 0.3. However, it performs better at higher and lower extremes. This behavior can be attributed to CATO's tendency to explore representations that require very few packets (due to the decay-shaped prior placed over connection depth), while also injecting priors based on each feature's relative importance. If we only consider solutions with an F1 score of at least 0.8, the HVI is 0.95 for CATO, 0.39 for SIMA and RAND, and 0 (no solutions found) for ITERALL.

Convergence Speed. Figure 8 compares the sample efficiency of CATO with alternative search algorithms. We extend the 50 sample explorations commonly used by ML practitioners [13, 32] to 1,500 to examine the convergence rate towards the true Pareto front as most of the search space is explored. We also plot the performance of CATO's baseline BO formulation without dimensionality reduction and prior injection (CATOBASE). We exclude ITERALL from this analysis since more than 50 iterations would exceed the maximum packet depth covered by the ground truth Pareto front. Moreover, we find that the HVI for ITERALL does not

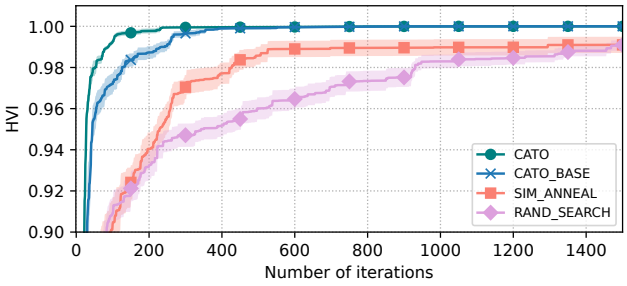


Figure 8: CATO efficiently converges to the true Pareto front. We show the mean and standard error of the HVI with a worst-case reference point across 20 runs.

show significant improvement beyond this point.

We can see that CATO converges to the true Pareto front fastest, demonstrating a more sample-efficient approach to optimizing ML-based traffic analysis pipelines. CATO surpasses 0.99 HVI (using a worst-case reference point) within 87 iterations on average compared to CATOBASE's 240 iterations, demonstrating a speedup of 2.76×. This speedup can be attributed to the incorporation of priors on the optimization parameters as described in Section 3.3, which helps CATO emphasize more promising regions in the search space. SIMA and RAND are less sample efficient, surpassing 0.99 HVI at 1,295 iterations and 1,469 iterations for a CATO speedup of 14.9× and 16.9×, respectively.

5.4 Ablation Study of the Profiler

We assess the impact of the Profiler on the estimated Pareto front found by CATO. We retain the Optimizer, including dimensionality reduction and prior injection, and perform an ablation study by replacing $\texttt{cost}(x)$ and $\texttt{perf}(x)$ measurements with heuristic metrics. We devise four variants of CATO. The first is CATO W/ NAÏVE COST, which replaces the original cost metric with the sum of the costs of each feature in isolation. This design captures the end-to-end systems costs of individual features, but fails to account for shared processing steps during packet capture and feature extraction. The second is CATO W/ MODEL INF COST, which measures the model inference speed but ignores the cost of packet capture and feature extraction. CATO W/ PKT DEPTH COST directly uses packet depth as the cost. CATO W/ NAÏVE PERF retains the original $\texttt{cost}(x)$ metric but replaces $\texttt{perf}(x)$ with the sum of each feature's mutual information with respect to the target variable. This version does not account for the effects of feature interactions.

We run each variant for 50 Optimizer iterations using the smaller candidate feature set, then measure (using the Profiler) the true $\texttt{perf}(x)$ and $\texttt{cost}(x)$ of each sampled point in a post-processing step to compare HVI. Figure 9 reveals that CATO comes closest to the true Pareto front, demonstrating that there is value to incorporating real model performance

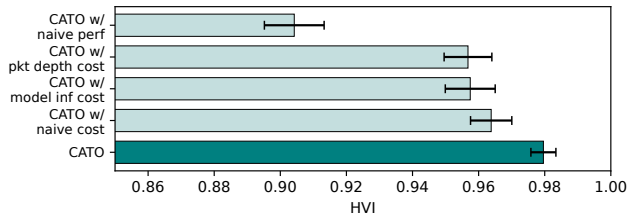


Figure 9: CATO with alternative Profiler metrics. End-to-end measurements prove useful for estimating the Pareto front and validating the performance of identified solutions.

and systems costs measurements as feedback for the Optimizer. Furthermore, we note that none of the variants provide a means to *validate* the expected in-network performance of their identified solutions. In particular, CATO W/ NAÏVE PERF and CATO W/ PKT DEPTH COST do not yield meaningful performance metrics. CATO W/ NAÏVE COST and CATO W/ MODEL INF COST are better, but may overestimate or underestimate the true systems cost, respectively. Depending on the concrete definition of $\texttt{cost}(x)$, this could lead to the deployment of an unrealizable model (e.g., by overestimating the throughput or underestimating the latency of the pipeline).

5.5 Microbenchmarks

In this section, we evaluate CATO's robustness to varying search space sizes and its sensitivity with respect to its BO initialization and damping coefficient hyperparameters.

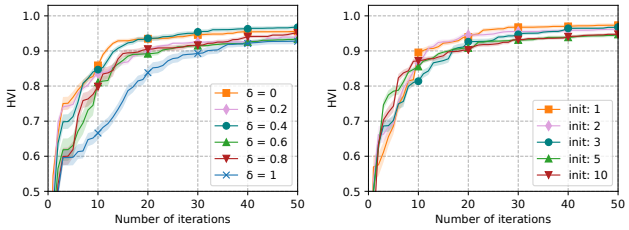
Varying Maximum Connection Depth. Previously, we showed that CATO is highly effective across both large (67) and small (6) candidate feature sets, but limited the search space to a maximum connection depth of 50 packets. Here, we explore the impact of maximum connection depth and the size of the search space on CATO's ability to identify Pareto-optimal feature representations.

In general, limiting the search space inherently restricts the number of possible feature representations, potentially compromising the quality of the estimated Pareto front. Conversely, expanding the search space offers a richer set of feature representations, but makes locating the true Pareto front more challenging. Table 3 shows CATO's performance across different maximum packet depths for `iot-class` using the full 67 candidate features. Since it is difficult to compare HVI across configurations without a ground truth, we report metrics for the estimated Pareto-optimal representations with the highest F1 score and lowest overall execution time.

We can see that restricting the packet depth to very small values (e.g., less than 5) limits CATO's ability to find a solution that yields an F1 score above 0.99, likely because not many feature sets can achieve such high model performance using only the first few packets in a flow. However, if we expand the search space to include features extracted from *up to* the first 10–100 packets, CATO is still able to identify fea-

Max Depth N	Highest F1 Score n	F1 score	Time (μs)	Lowest Execution Time n	F1 score	Time (μs)
3	3	0.959	1.37	1	0.310	0.2
5	4	0.983	1.30	1	0.520	0.26
10	7	0.994	2.04	1	0.520	0.26
25	7	0.989	2.61	1	0.520	0.26
50	7	0.993	2.10	1	0.461	0.27
100	10	0.990	3.01	1	0.005	0.24
∞	42k	0.984	28.2	52k	0.944	16.5

Table 3: Estimated Pareto-optimal solutions with the highest F1 score and lowest execution time for different maximum packet depths. CATO is able to identify high quality solutions, even when expanding the maximum connection depth.



(a) Damping coefficient δ. (b) BO initialization samples.

Figure 10: Effects of varying the damping coefficient and the number of samples used to initialize the BO surrogate model.

ture sets that only need the first 7–10 packets to achieve good model performance despite the larger maximum packet depth. However, if the search space over packet depth is unbounded, CATO struggles converge to a feature representation with low cost since the Optimizer has too much flexibility to explore any value between 1 and the maximum number of packets across *all* flows in the training set. These results reinforce that concurrently searching over different features and when to collect those features, rather than predefining a fixed connection depth, can lead to more optimal traffic analysis pipelines. CATO can identify highly efficient and predictive feature representations within a wide range of connection depths, even without prior knowledge of its optimum.

Sensitivity Analysis. We analyze the sensitivity of CATO with respect to its hyperparameters. We again use the smaller candidate feature set to measure HVI against the true Pareto front with a worst-case reference point. Figure 10a shows the impact of varying the damping coefficient δ between 0 and 1. Recall that $\delta = 0$ represents a prior probability equivalent to the normalized mutual information, while $\delta = 1$ represents uniform priors for each feature. Using uniform priors performs the worst with an HVI of 0.93 after 50 iterations. With less damping, CATO converges faster in earlier iterations. We find that a $\delta = 0.4$ results in the highest performance at 50 iterations, while $\delta = 0$ also performs well overall.

In Figure 10b, we vary the number of samples used to initialize the BO surrogate model. Initialization samples are chosen at random but weighted according to the priors. We observe little difference in CATO's performance for small initialization values, but find that initializing with just 1 point empirically results in the highest HVI after 50 iterations. We choose to go with a more conventional value for BO [35], and choose 3 initialization samples by default.

6 Related Work

Efficient Inference. Several systems have been proposed for efficient traffic analysis inference. However, many of these efforts focus only on increasing the speed of the final model inference stage, rather than that of the end-to-end serving pipeline [22, 43, 69]. Such approaches overlook the effects of the packet capture and feature extraction stages, both of which CATO considers in its end-to-end optimization. AC-DC [34] and pForest [16] are similar in that they explicitly consider the preprocessing costs of extracting features. AC-DC performs inference under dynamic memory constraints while pForest targets programmable hardware. Both of these differ from CATO in that they generate pools of models and dynamically switch between them based on inference requirements.

There is a growing body of research that proposes the use of programmable hardware for traffic analysis [9, 16, 33, 36, 59, 64, 67, 68, 74, 75, 80, 82]. For example, N3IC [64] uses binary neural networks to implement traffic analysis models on FPGAs and SmartNICs, while BoS [75] enables RNN inference on programmable switches. These approaches focus on the trade-off between accuracy and efficiency of the *model* under the constraints of dataplane hardware, such as limited memory and lack of support for floating-point operations, multiplications, and loops. Our work, by contrast, focuses on the choice of *traffic* representation—spanning both feature selection and connection depth—to co-optimize systems performance and predictive accuracy. CATO can complement hardware-focused techniques: by applying the CATO Optimizer and extending our Profiler implementation to target programmable hardware, we can identify traffic representations that are cheaper to collect while also validating the end-to-end performance of the hardware pipeline. This combined strategy can further boost the efficiency and predictive performance of ML-based traffic analysis, which we leave for future work.

Balancing Systems and Model Performance. There exist general-purpose serving frameworks that aim to balance efficiency with model performance through techniques like caching [20], adaptive model selection [20, 58], autoscaling [19, 58, 78], and scheduling for resource-quality tradeoffs [42, 79]. Unlike CATO, these frameworks primarily focus on optimizing resource allocation and are less suited for real-time traffic analysis, where even small inefficiencies in end-to-end systems performance can cause not just delayed results, but also potentially invalidate the model due to packet loss. Additionally, CATO explicitly considers the optimal point in the flow (i.e., connection depth) for making

predictions, which previous frameworks do not address. Traffic Refinery [14] considers joint optimization of model and system performance for ML-based network traffic analysis. However, it relies on manual exploration of data representations and focuses only on individual feature costs, whereas CATO performs automated end-to-end optimization of the entire analysis pipeline.

Bayesian Optimization for Traffic Analysis. Bayesian Optimization is a popular technique for compiler optimization [28], FPGA design [52], and hyperparameter tuning [66, 70], but has seen limited use in optimizing ML-based traffic analysis pipelines. Most similar is Homunculus [68], which uses Bayesian optimization to generate ML models for datacenter network applications under the resource constraints of data-plane hardware. Homunculus optimizes over different model architectures and their hyperparameters, but is single-objective and does not consider the effects of different feature sets, connection depths, and their associated systems costs. CATO, on the other hand, uses multi-objective BO over the entire feature representation space to simultaneously optimize serving efficiency and predictive performance.

7 Conclusion

In this work, we introduced CATO, a framework for end-to-end optimization of ML-based traffic analysis pipelines. By leveraging multi-objective Bayesian optimization coupled with a realistic pipeline generator and profiler, CATO efficiently builds and validates serving pipelines that balance both model accuracy and systems performance. Our evaluations on live traffic and offline traces showed that CATO can improve end-to-end inference latency, throughput, and pipeline execution time across diverse traffic analysis tasks, while also maintaining or enhancing model performance. Future work includes broader model selection strategies and extending performance profiling across heterogeneous serving hardware.

Acknowledgements

We thank our shepherd Zhizhen Zhong as well as Tina Wu, Thea Rossman, Qizheng Zhang, Carl Hvarfner, Luigi Nardi, and the anonymous reviewers for their helpful feedback. We thank the Stanford networking team, including Andrej Krevl, Johan van Reijendam, and Will Johnson. This work was supported in part by a Sloan Research Fellowship, the National Science Foundation under Grant Numbers #2319080 and #2124424, the ANR Project No ANR-21-CE94-0001-01 (MINT), and gifts from Google, Inc., Cisco Systems, Inc., and Comcast Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or other funding organizations.

References

[1] Smartcore. https://smartcorelib.org/, 2023.

[2] Web application firewall documentation. https://learn.microsoft.com/en-us/azure/web-application-firewall/, 2023.

[3] Zeek. https://zeek.org/, 2023.

[4] Mahmoud Abbasi, Amin Shahraki, and Amir Taherkordi. Deep learning for network traffic monitoring and analysis (NTMA): A survey. In *Computer Communications*, 2021.

[5] Hasan Faik Alan and Jasleen Kaur. Can android applications be identified using only TCP/IP headers of their launch time traffic? In *ACM Conference on Security and Privacy in Wireless Networks*, 2016.

[6] André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. Permutation importance: A corrected feature importance measure. In *Bioinformatics*, 2010.

[7] Zied Aouini and Adrian Pekar. NFStream: A flexible network data analysis framework. In *Computer Networks*, 2022.

[8] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for internet video. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[9] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando M. V. Ramos, and André Madeira. Flowlens: Enabling efficient flow classification for ml-based network security applications. In *Network and Distributed Systems Security Symposium (NDSS)*, 2021.

[10] Laurent Bernaille, Renata Teixeira, Ismael Akodjenou, Augustin Soule, and Kavé Salamatian. Traffic classification on the fly. In *ACM SIGCOMM Computer Communication Review*, 2006.

[11] Laurent Bernaille, Renata Teixeira, and Kavé Salamatian. Early application identification. In *International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2006.

[12] Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M. Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. In *Journal of Internet Services and Applications*, 2018.

[13] Xavier Bouthillier and Gaël Varoquaux. Survey of machine-learning experimental methods at NeurIPS2019 and ICLR2020. *Research Report, Inria Saclay Ile de France*, 2020.

[14] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Hyojoon Kim, Renata Teixeira, and Nick Feamster. Traffic refinery: Cost-aware data representation for machine learning on network traffic. In *ACM Measurement and Analysis of Computing Systems*, 2021.

[15] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. In *ACM Measurement and Analysis of Computing Systems*, 2019.

[16] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680v2*, 2022.

[17] Roberto Calandra, Nakul Gopalan, André Seyfarth, Jan Peters, and Marc Peter Deisenroth. Bayesian gait optimization for bipedal locomotion. In *International Conference on Learning and Intelligent Optimization (LION)*, 2014.

[18] Briang Chang, Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Learned load balancing. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2023.

[19] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *ACM Symposium on Cloud Computing*, 2020.

[20] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[21] Alberto Dainotti, Antonio Pescapé, and Carlo Sansone. Early classification of network traffic through multi-classification. In *International Workshop on Traffic Monitoring and Analysis (TMA)*, 2011.

[22] Kayathri Devi Devprasad, Sukumar Ramanujam, and Suresh Babu Rajendran. Context adaptive ensemble classification mechanism with multi-criteria decision making for network intrusion detection. *Concurrency and Computation: Practice and Experience*, 2022.

[23] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of encrypted and VPN traffic using time-related features. In *International Conference on Information Systems Security and Privacy (ICISSP)*, 2016.

[24] Peter I. Frazier. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[25] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. Realtime robust malicious traffic detection via frequency domain analysis. In *ACM SIGSAC Conference on Computer and Communication Security (CCS)*, 2021.

[26] Craig Gutterman, Katherine Guo, Sarthak Arora, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, and Gil Zussman. Requet: Real-time QoE detection for encrypted youtube traffic. In *ACM Transactions on Multimedia Computing, Communications*, 2020.

[27] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. In *Machine Learning*, 2002.

[28] Erik Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. BaCO: A fast and portable Bayesian compiler optimization framework. In *ACM Architectural Support for Programming Languages and Operating Systems*, 2023.

[29] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. New directions in automated traffic analysis. In *ACM Conference on Computer and Communication Security (CCS)*, 2021.

[30] Nen-Fu Huang, Gin-Yuan Jai, Han-Chieh Chao, Yih-Jou Tzang, and Hong-Yi Chang. Application traffic classification at the early stage by characterizing application rounds. In *Information Sciences*, 2013.

[31] Johann Hugon, Gaetan Nodet, Anthony Busson, and Francesco Bronzino. Towards adaptive ml traffic processing systems. In *Proceedings of the on CoNEXT Student Workshop 2023*, 2023.

[32] Carl Hvarfner, Danny Stoll, Artur Souza, Marius Lindauer, Frank Hutter, and Luigi Nardi. πBO: Augmenting acquisition functions with user beliefs for Bayesian optimization. In *International Conference on Learning Representations (ICLR)*, 2022.

[33] Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani. Leo: Online ml-based traffic classification at multi-terabit line rate. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[34] Xi Jiang, Shinan Liu, Saloua Naama, Francesco Bronzino, Paul Schmitt, and Nick Feamster. AC-DC: Adaptive ensemble classification for network traffic identification. *arXiv preprint arXiv:2302.11718*, 2023.

[35] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. In *Journal of Global Optimization*, 1998.

[36] Radhakrishna Kamath and Krishna M. Sivalingam. Machine learning based flow classification in DCNs using P4 switches. In *International Conference on Computer Communications and Networks*, 2015.

[37] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. BLINC: Multilevel traffic classification in the dark. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2005.

[38] Scott Kirkpatrick, C. Daniel Gelatt Jr., and Mario P. Vecchi. Optimization by simulated annealing. In *Science*, 1983.

[39] Vengatanathan Krishnamoorthi, Niklas Carlsson, Emir Halepovic, and Eric Petajan. BUFFEST: Predicting buffer conditions and real-time requirements of HTTP(S) adaptive streaming clients. In *ACM Multimedia Systems Conference*, 2017.

[40] Jong-Hyouk Lee and Kamal Singh. SwitchTree: In-network computing and traffic analyses with random forests. In *Neural Computing and Applications*, 2020.

[41] Miqing Li and Xin Yao. Quality evaluation of solution sets in multiobjective optimisation: A survey. In *ACM Computing Surveys*, 2019.

[42] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. *USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[43] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *SIGCOMM*, 2019.

[44] Yingqiu Liu, Wei Li, and Yunchun Li. Network traffic classification using K-means clustering. In *International Multi-Symposiums on Computer and Computational Sciences*, 2007.

[45] Manuel Lopez-Martin, Belen Carro, Antonio Sanchez-Esguevillas, and Jaime Lloret. Network traffic classifier with convolutional and recurrent neural networks for internet of things. In *IEEE Access*, 2017.

[46] Mohammad Lotfollahi, Mahdi Jafari Siavoshani, Ramin Shirali Hossein Zade, and Mohammdsadegh Saberian. Deep packet: A novel approach for encrypted traffic classification using deep learning. *Soft Computing*, 2020.

[47] Tarun Mangla, Emir Halepovic, Mostafa Ammar, and Ellen Zegura. Using session modeling to estimate HTTP-based video QoE metrics from encrypted network traffic. In *IEEE Transactions on Network and Service Management*, 2019.

[48] M. Hammad Mazhar and Zubair Shafiq. Real-time video quality of experience monitoring for HTTPS and QUIC. In *IEEE International Conference on Computer Communications*, 2018.

[49] M. Hammad Mazhar and Zubair Shafiq. Real-time video quality of experience monitoring for HTTPS and QUIC. In *IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[50] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, Ahmad-Reza Sadeghi, N. Asokan, and Sasu Tarkoma. IoT sentinel: Automated device-type identification for security enforcement in IoT. In *International Conference on Distributed Computing Systems*, 2017.

[51] Andrew Moore, Denis Zuev, and Michael Crogan. Discriminators for use in flow-based classification. Technical report, 2005.

[52] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. HyperMapper: a practical design space exploration framework. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2019.

[53] Thuy T.T. Nguyen and Grenville Armitage. A comprehensive survey on machine learning for networking: Evolution, applications and research opportunities. In *IEEE Communications Surveys & Tutorials*, 2008.

[54] Lizhi Peng, Bo Yang, and Yuehui Chen. Effective packet number for early stage internet traffic identification. In *Neurocomputing*, 2015.

[55] Julien Piet, Dubem Nwoji, and Vern Paxson. GGFAST: Automating generation of flexible network traffic classifiers. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2023.

[56] Shahbaz Rezaei, Bryce Kroencke, and Xin Liu. Large-scale mobile app identification using deep learning. In *IEEE Access*, 2019.

[57] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. *arXiv preprint arXiv:1708.06376*, 2017.

[58] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[59] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the ai accelerator? In *Morning Workshop on In-Network Computing*, 2018.

[60] Gabriel Gómez Sena and Pablo Belzarena. Early traffic classification using support vector machines. In *International Latin American Networking Conference (LANC)*, 2009.

[61] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. In *Proceedings of IEEE, vol. 104, no. 1*, 2016.

[62] Tal Shapira and Yuval Shavitt. FlowPic: A generic representation for encrypted traffic classification and applications identification. In *IEEE Transactions on Network and Service Management*, 2021.

[63] Jayveer Singh and Manisha Nene. A survey on machine learning techniques for intrusion detection systems. In *Intl. Journal of Advanced Research in Computer and Communication Engineering*, 2013.

[64] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting traffic analysis with neural network interface cards. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[65] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT devices in smart environments using network traffic characteristics. In *IEEE Transactions on Mobile Computing*, 2019.

[66] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian optimization of machine learning algorithms. In *International Conference on Advances in Neural Information Processing Systems*, 2012.

[67] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A data plane architecture for per-packet ML. In *ACM Architectural Support for Programming Languages and Operating Systems*, 2022.

[68] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. Homunculus: Auto-generating efficient data-plane ML pipelines for datacenter networks. In *ACM Architectural Support for Programming Languages and Operating Systems*, 2023.

[69] Da Tong, Yun R Qu, and Viktor K Prasanna. High-throughput traffic classification on multi-core processors. In *IEEE International Conference on High Performance Switching and Routing*, 2014.

[70] Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *Proceedings of the NeurIPS 2020 Competition and Demonstration Track,*, 2021.

[71] Jorge R. Vergara and Pablo A. Estévez. A review of feature selection methods based on mutual information. In *Neural Computing and Applications*, 2014.

[72] Gerry Wan, Fengchen Gong, Tom Barbette, and Zakir Durumeric. Retina: Analyzing 100 GbE traffic on commodity hardware. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[73] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. Malware traffic classification using convolutional neural network for representation learning. In *International Conference on Information Networking*, 2017.

[74] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *ACM Workshop on Hot Topics in Networks*, 2019.

[75] Jinzhu Yan, Haotian Xu, Zhuotao Liu, Qi Li, Ke Xu, Mingwei Xu, and Jianping Wu. Brain-on-switch: Towards advanced intelligent network data plane via nn-driven traffic analysis at line-speed. In *USENIX Symposium on Networked Systems Design and Implementation*, 2024.

[76] Hao Yang, Qin He, Zhenyan Liu, and Qian Zhang. Malicious encryption traffic detection based on NLP. In *Security and Communication Networks*, 2021.

[77] Kun Yang, Nick Feamster, and Samory Kpotufe. Feature extraction for novelty detection in network traffic. *arXiv preprint arXiv:2006.16993v2*, 2021.

[78] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *USENIX Annual Technical Conference*, 2019.

[79] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *USENIX Symposium on Networked Systems Design and Implementation*, 2017.

[80] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Automating in-network machine learning. *arXiv preprint arXiv:2205.08824v1*, 2022.

[81] Weiping Zheng, Jianhao Zhong, Qizhi Zhang, and Gansen Zhao. MTT: An efficient model for encrypted network traffic classification using multi-task transformer. *Applied Intelligence*, 2022.

[82] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. An efficient design of intelligent network data plane. In *USENIX Security Symposium*, 2023.

A Candidate Features

Table 4 lists the set of candidate features used in our evaluations. We aggregate common network flow features used throughout networking research for ML-based traffic analysis [3, 15, 16, 51, 64, 68, 72]. These are also commonly supported by open source tools, and are not specific to any particular use case. However, for privacy reasons related to our live traffic experiments, we restrict these to various types of summary statistics to avoid saving packet payloads to disk.

B Dataset Collection

All experiments with live traffic are performed on a server running Ubuntu 20.04, with a dual Xeon Gold 6248R 3GHz CPU, 384 GB of memory, and a 100GbE Mellanox ConnectX-5 NIC. All experiments using offline datasets are performed on a server with a dual Xeon Gold 6154 3GHz CPU and 384 GB of memory, running Ubuntu 20.04.

Web Application Classification. For `app-class`, we classify the following applications: Netflix, Twitch, Zoom, Microsoft Teams, Facebook, Twitter, or "other." Connections are labeled using the SNI from the TLS handshake, and only statistical features listed in Table 4 are collected (see Appendix H for ethical considerations involving live traffic). We use Retina [72] for data collection, which we run for 30 seconds with 12.5% flow sampling, and an additional 10 minutes with 50% flow sampling filtered on the target applications to help collect a more balanced dataset. Flow sampling reduces the effective ingress network throughput while maintaining per-connection consistency, and is done entirely in the NIC

Feature	Description	In mini cand. set
dur	total duration	yes
proto	transport layer protocol	no
s_port	src port	no
d_port	dst port	no
s_load	src → dst bps	yes
d_load	dst → src bps	no
s_pkt_cnt	src → dst packet count	yes
d_pkt_cnt	dst → src packet count	no
tcp_rtt	time between SYN and ACK	no
syn_ack	time between SYN and SYN/ACK	no
ack_dat	time between SYN/ACK and ACK	no
s_bytes_sum	src → dst total bytes	yes
d_bytes_sum	dst → src total bytes	no
s_bytes_mean	src → dst mean packet size	yes
d_bytes_mean	dst → src mean packet size	no
s_bytes_min	src → dst min packet size	no
d_bytes_min	dst → src min packet size	no
s_bytes_max	src → dst max packet size	no
d_bytes_max	dst → src max packet size	no
s_bytes_med	src → dst median packet size	no
d_bytes_med	dst → src median packet size	no
s_bytes_std	src → dst std dev packet size	no
d_bytes_std	dst → src std dev packet size	no
s_iat_sum	src → dst total packet inter-arrival time	no
d_iat_sum	dst → src total packet inter-arrival time	no
s_iat_mean	src → dst mean packet inter-arrival time	yes
d_iat_mean	dst → src mean packet inter-arrival time	no
s_iat_min	src → dst min packet inter-arrival time	no
d_iat_min	dst → src min packet inter-arrival time	no
s_iat_max	src → dst max packet inter-arrival time	no
d_iat_max	dst → src max packet inter-arrival time	no
s_iat_med	src → dst median packet inter-arrival time	no
d_iat_med	dst → src median packet inter-arrival time	no
s_iat_std	src → dst std dev packet inter-arrival time	no
d_iat_std	dst → src std dev packet inter-arrival time	no
s_winsize_sum	src → dst sum of TCP window sizes	no
d_winsize_sum	dst → src sum of TCP window sizes	no
s_winsize_mean	src → dst mean TCP window size	no
d_winsize_mean	dst → src mean TCP window size	no
s_winsize_min	src → dst min TCP window size	no
d_winsize_min	dst → src min TCP window size	no
s_winsize_max	src → dst max TCP window size	no
d_winsize_max	dst → src max TCP window size	no
s_winsize_med	src → dst med TCP window size	no
d_winsize_med	dst → src med TCP window size	no
s_winsize_std	src → dst std dev TCP window size	no
d_winsize_std	dst → src std dev TCP window size	no
s_ttl_sum	src → dst sum of IP TTL values	no
d_ttl_sum	dst → dst sum of IP TTL values	no
s_ttl_mean	src → dst mean TTL	no
d_ttl_mean	dst → src mean TTL	no
s_ttl_min	src → dst min TTL	no
d_ttl_min	dst → src min TTL	no
s_ttl_max	src → dst max TTL	no
d_ttl_max	dst → src max TTL	no
s_ttl_med	src → dst median TTL	no
d_ttl_med	dst → src median TTL	no
s_ttl_std	src → dst std dev TTL	no
d_ttl_std	dst → src std dev TTL	no
cwr_cnt	number of packets with CWR flag set	no
ece_cnt	number of packets with ECE flag set	no
urg_cnt	number of packets with URG flag set	no
ack_cnt	number of packets with ACK flag set	no
psh_cnt	number of packets with PSH flag set	no
rst_cnt	number of packets with RST flag set	no
syn_cnt	number of packets with SYN flag set	no
fin_cnt	number of packets with FIN flag set	no

Table 4: Candidate feature set $\mathcal{F}$ containing 67 commonly used flow features. We indicate the six that are used in the smaller candidate set for ground truth analyses.

using hardware filters [72]. We avoid collecting at full network throughput to ensure that no packets are dropped in the data collection phase. In total, we collected 2M samples of connection data over 50 different packet depths.

IoT Device Recognition. For `iot-class`, we classify one of 28 IoT device types using the UNSW IoT dataset [65]. We use the September 2016 traces in our evaluations, which include approximately 134K connections. Models are trained and evaluated using data from eight days of packet traces.

Video Startup Delay Inference. We use the dataset collected by Bronzino et al. [15] for `vid-start`, focusing exclusively on YouTube traffic, where each video session consists of a single TCP connection. The final dataset comprises 4,287 connections, capturing a wide range of startup delay times. Delay times range from 315 ms to 54 seconds at P99, with the maximum observed delay being 14 minutes.

C Model Training

For DT and RF model training, we use scikit-learn's DecisionTreeClassifier and RandomForestClassifier with default parameters and tune the maximum tree depth from the set {3, 5, 10, 15, 20}. The RandomForestClassifier is configured with 100 estimators. To integrate with the Rust-based packet capture and feature extraction stages, the tuned DT and RF models are retrained in Rust using the SmartCore [1] library.

DNN hyperparameters are tuned over the following values: batch size {16, 32, 64}, learning rate {0.001, 0.01}, dropout rate {0.2, 0.4, 0.6, 0.8}, L2 regularization {0.1, 0.5}, and number of neurons in each hidden layer {4, 8, 16}.

D Measurement Details

End-to-End Inference Latency. We implement two versions of inference latency measurement, depending on the use case. For `app-class`, we record the arrival timestamp of the SYN packet, and subtract it from the timestamp of the final prediction output by the model. Since we do not have access to live traffic from the `iot-class` dataset, we compute the end-to-end inference latency by taking the sum of the pipeline execution time, model inference time, and packet inter-arrival times up to the specified connection depth. The inter-arrival times are calculated from packet timestamps in the traces.

Zero-Loss Throughput. Since we are unable to control the input traffic rate on our live network, we choose to restrict our serving pipelines to a single core to differentiate the performance of each optimization method. This ensures that we can find an upper bound on the throughput since no solution will be able to saturate the input traffic rate. To measure the zero-loss throughput, we leverage Retina's flow sampling capabilities by starting at the full traffic rate and slowly decreasing the percentage of flows randomly dropped by the NIC until we observe zero packet loss for 30 seconds. We

repeat this for multiple trials and take the average zero-loss throughput sustained by the traffic analysis pipeline.

Execution Time. Pipeline execution time is a measure of total CPU time spent in the serving pipeline, excluding time spent waiting for packets to arrive. We measure this by inserting calls to query the Read Time-Stamp Counter register at the start and end of each packet processing step and the model inference stage.

E Optimization Wall-Clock Time

Wall-clock time depends heavily on the application use-case, model type, number of samples explored, and the concrete definitions of $\texttt{cost}(x)$ and $\texttt{perf}(x)$. For reference, Table 5 reports the breakdown in time elapsed for CATO to compute the Pareto fronts depicted in Figure 5d and Figure 7, which target different use cases and system cost metrics. As expected, execution time is mostly consumed by the Profiler, which, on each iteration, generates a fresh serving pipeline, trains and evaluates the model, and measures the end-to-end systems costs. We deem this a worthwhile trade-off because the time spent by the Profiler to validate the systems cost of each sampled solution ensures that the resulting Pareto-optimal pipeline can meet real-time performance requirements. The BO-guided sampling that determines the next feature representation for the Profiler to evaluate adds between 1.4 and 55 seconds per iteration depending on the search space.

F Reproducing Traffic Refinery

We reproduce key components of the Traffic Refinery [14] framework for evaluation. Traffic Refinery defines several feature classes that contain features commonly used in ML-based traffic analysis. Specifically, we replicate the PacketCounter (PC), PacketTiming (PT), and TCPCounter (TC) feature classes using subsets of our candidate feature set. PC includes all packet and byte counters, PT includes all packet inter-arrival statistics, and TC includes all flag counters, window size statistics, and RTT. We simulate using Traffic Refinery by manually aggregating feature classes, varying the packet depths, and measuring the predictive performance and pipeline execution time using CATO's Profiler. While Traffic Refinery also has the ability to profile state and storage costs, we focus on execution time in our evaluation since it is a shared cost metric in both frameworks.

G Simulated Annealing Details

We describe our simulated annealing [38] implementation SIMA from Section 5.3. SIMA starts with a random feature representation x as the current "best" point. On each iteration i, it samples a neighbor point x_i and measures $\texttt{cost}(x_i)$ and $\texttt{perf}(x_i)$. Neighbors are sampled by randomly perturbing either the feature set or the packet depth with equal probability:

- Feature set perturbation: Add, remove, or replace a feature at random.
- Packet depth perturbation: Move up to some maximum step size away from the current packet depth, with the maximum step size decreasing linearly from the maximum packet depth as more samples are explored. This allows for more exploration earlier in the search.

Since our optimization is multi-objective, we adjust the standard simulated annealing neighbor acceptance criterion as follows: If the neighboring point dominates the current point across both objectives, it is accepted as the new current point. Otherwise, it is still accepted with probability $\mathbb{P}(x, x_i, T_i) = \exp(f(x) - f(x_i))/T_i$, where $f(x)$ is an equal weighted combination of $\texttt{perf}(x)$ and $\texttt{cost}(x)$, and T_i is the temperature at iteration i. Consistent with simulated annealing algorithms, the temperature gradually decreases as the search space is explored. This mechanism allows for non-dominating solutions to still be accepted with higher probability at the beginning of the search process, preventing SIMA from getting trapped in local minima. We empirically tune SIMA with different initial temperatures and cooling schedules, and choose $T_0 = 1$ and $T_{i+1} = 0.99T_i$.

Figure 8 (Section 5.3) reveals that while SIMA is less sample efficient than CATO, it is generally more efficient than RAND. However, RAND catches up after approximately 1500 sample explorations, likely due to SIMA's reduced ability to explore new feature representations as the temperature decreases.

Use case / # cand. features **Systems cost metric:**	`app-class` / 67 zero-loss throughput	`iot-class` / 6 processing time
Preprocessing	22.4 s	4.1 s
Opt. Iteration (50×)		
BO sample	55.5 s	1.4 s
Pipeline generation	53.1 s	46.5 s
Measure $\texttt{perf}(x)$	29.6 s	26.4 s
Measure $\texttt{cost}(x)$	546.7 s	70.3 s
Total elapsed	9.5 h	2 h

Table 5: CATO optimization wall-clock times. BO is well-suited for expensive-to-evaluate objective functions, such as $\texttt{perf}(x)$ and $\texttt{cost}(x)$.

H Ethical Considerations

As part of our experiments with high-speed network traffic, we evaluated the performance of models against live campus network traffic. The candidate flow features we use included only aggregate flow statistics and an application name derived from the SNI field in TLS handshakes. We never captured or analyzed client IP addresses, viewed any individual flows or connection records, stored any packets to disk, or investigated human behavior; our IRB has ruled that this type of analysis

does not constitute human subjects research. Nonetheless, we took steps to ensure the security and privacy of campus users. All live traffic experiments were isolated to a single hardened server that was deployed in partnership with our campus networking and security teams in order to not increase the attack surface for users.