Vulcan: Automatic Query Planning for Live ML Analytics

Yiwen Zhang[†], Xumiao Zhang[†], Ganesh Ananthanarayanan[‡], Anand Iyer[§],
Yuanchao Shu[††], Victor Bahl[‡], Z. Morley Mao[†¶], Mosharaf Chowdhury[†]
[†]University of Michigan, [‡]Microsoft, [§]Georgia Institute of Technology, [††]Zhejiang University, [¶]Google

Abstract

Live ML analytics have gained increasing popularity with large-scale deployments due to recent evolution of ML technologies. To serve live ML queries, experts nowadays still need to perform manual query planning, which involves pipeline construction, query configuration, and pipeline placement across multiple edge tiers in a heterogeneous infrastructure. Finding the best query plan for a live ML query requires navigating a huge search space, calling for an efficient and systematic solution.

In this paper, we propose Vulcan, a system that automatically generates query plans for live ML queries to optimize their accuracy, latency, and resource consumption. Based on the user query and performance requirements, Vulcan determines the best pipeline, placement, and query configuration for the query with low profiling cost; it also performs fast online adaptation after query deployment. Vulcan outperforms state-of-the-art ML analytics systems by 4.1×-30.1× in terms of search cost while delivering up to 3.3× better query latency.

1 INTRODUCTION

Recent years have witnessed a growing demand for machine learning (ML) analytics. *Live ML analytics* – with applications in edge-assisted autonomous driving [62, 71, 72], live traffic analysis [1, 43], and real-time speech recognition [18, 66] – stands out due to its large-scale deployments [1,2,4]. Live ML analytics involves ML pipelines at its core, where each pipeline consists of a series of operators to perform specific ML tasks. For example, an autonomous driving perception query that detects surrounding objects of an autonomous vehicle may contain filtering operators for road surface removal [23] and 3D data compression [22,53], along with a 3D object detector to perform object detection [37,68].

Live ML analytics differentiates itself from ML on stored data in two key characteristics. First, live ML pipelines are deployed across *heterogeneous infrastructure*, spanning multiple tiers such as device edges, on-premise edges, public MEC, and cloud datacenters [1–3,6]. Second, live ML queries have *latency requirements* besides accuracy targets as the analytics is based on real-time data. Some analytics, such as object detection in autonomous vehicles, may have more stringent requirements than others depending on the priority of the task. Therefore, before deploying live ML analytics, each query describing the ML task must go through careful *query planning*. Specifically, this involves (i) constructing the pipeline by selecting a series of operators and ordering them; (ii) determining the physical placement of pipeline operators across infrastructure tiers; and (iii) selecting configurations of the pipeline operators, to optimize for query performance. A joint optimization of the three aspects is required for optimal performance and resource consumption.

Recent research on these topics have been piecemeal, focused on compute alone, and hence largely sub-optimal for live ML analytics. Although declarative query languages (e.g., SQL) have been proposed for ML queries, there exists no systematic approach to automatically construct pipelines based on query's end-to-end latency requirement. This includes selecting and ordering the filtering modules during pipeline construction, which have a great impact on the performance characteristics of ML pipelines. Furthermore, when choosing physical placement of pipeline components, one cannot afford to exhaustively search for the optimal placement through real-world deployments. As a result, deployments often rely on past experience with simple heuristics [3,67]. While recent solutions have focused largely on selecting query configurations [12,27,28,63,70], they assume the ML analytics component to be a monolithic module instead of a pipeline. As such, they are profiled for compute alone, assuming they are running in a homogeneous datacenter. In the process, networking resources are ignored, and the complexities of multi-resource planning of compute and network are overlooked altogether. Finally, these solutions rely heavily on domain-specific insights of video content, which are not applicable to general ML scenarios beyond video analytics [28,45,70].

After deploying the query in the wild, one must also adapt the query plan based on runtime dynamics such as data content and/or resource changes [12,28,52,54]. Prior solutions in providing online adaptation [28,70] focus on data content changes alone but do not adapt to compute and network resource changes, which are common in edge environments [52]. An ideal solution should change query configuration *and* placement during online adaption.

In this work, we consider how to perform automatic query planning – i.e., constructing, placing, and configuring ML pipelines, along with adapting to runtime dynamics – for live ML queries based on user-provided performance requirements. The goal is to find query plans that optimize latency and accuracy, while minimizing the network and compute resource consumption. Generating such query plans, however, is challenging as jointly optimizing pipeline construction,

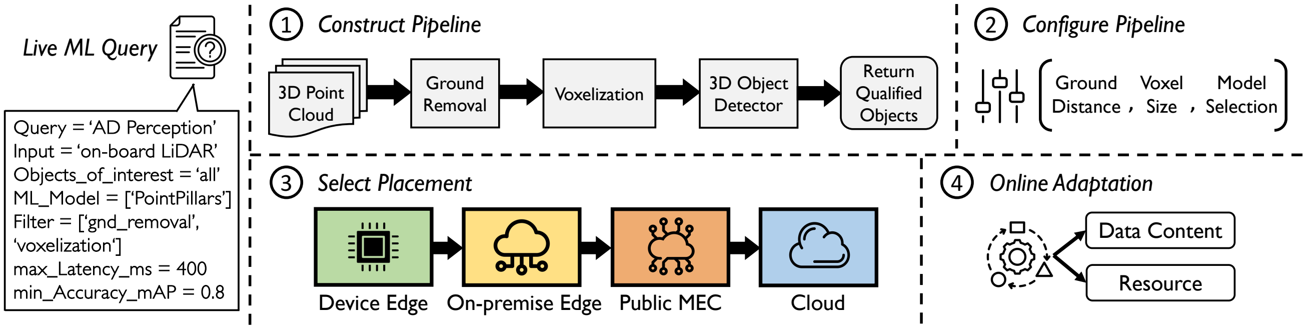


Figure 1: The existing workflow of query planning for a live ML query.

placement, and query configuration leads to a much larger search space. Moreover, as query performance is resource and data dependent, considering all possible resource and data dynamics that may or may not happen in the future can explode the search space. Clearly, a more efficient online adaptation technique is needed for faster convergence and lower cost.

We present Vulcan, an ML analytics system that performs automatic query planning for live ML queries. Its design includes the following key ideas to overcome the aforementioned challenges:

(1) Vulcan defines a novel metric to quantify each filtering operator in the pipeline by combining query precision, recall, resource usage, and latency (§4.2). This converts the complexity of filter ordering from exponential to linear.

(2) Vulcan carefully identifies components of ML pipelines that are *independent* of placement. This allows Vulcan to dramatically prune the search space of placement options.

(3) Vulcan efficiently explores the best combination of configuration knobs using Bayesian Optimization (BO) [14, 51]. It designs BO's priori assumptions and acquisition function to jointly optimize placement and configuration selection.

(4) Vulcan adapts quickly to dynamic changes in data and resources by (i) designing programming interfaces that allow for dynamic updates to live pipelines modules without disrupting them, and (ii) leveraging prior knowledge to make faster decisions on modifying configurations and placement.

We evaluate Vulcan using real-world datasets on a wide range of applications including traffic monitoring, autonomous driving perception, and automatic speech recognition. Experiments are conducted under an edge hierarchy that represents real compute and network resource setting of our production infrastructure. Vulcan generates query plans with better profiling cost by 4.1×-30.1× over state-of-the-art ML analytics systems while delivering 3.3× better query latency performance. Vulcan outperforms existing solutions for ML query configuration and placement selection with up to 2.8× better query latency and 174× lower network resource consumption. Vulcan also achieves consistently better 99^{th}-p latency by up to 2.5× by adapting to both data and resource changes during online adaptation (§6).

In summary, we make the following research contributions:

- We provide an end-to-end system design for live ML queries for a wide range of real-world ML applications.
- We propose novel solutions on search space reduction for constructing, placing, and configuring ML pipelines.
- We implement interfaces and control loop for online adaptation for fast re-profiling and dynamic re-configuration.

2 BACKGROUND AND MOTIVATION

We start with an overview of the workflow of live ML query processing, followed by motivating examples to highlight the key aspects when choosing query plans.

2.1 Processing Live ML Queries

We explain the query planning workflow by walking through an example query which detects surrounding objects in edge-assisted autonomous driving, as shown in Figure 1. The input query specifies input data, object of interests, pipeline operators, and performance requirements on accuracy and latency.

① **Constructing the pipeline.** The first step is constructing the ML pipeline by choosing a series of operators to perform the task. Specifically, it involves choosing *filtering operators*, such as voxelization and ground removal, to be deployed for substantial resource efficiency prior to the ML models, such as the 3D object detector [37, 68] in the case of Figure 1. The ordering of the filtering operators has a significant impact on performance. The best ordering is dependent on the data, resource availabilities, and performance requirements.

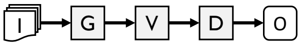
② **Configuring the pipeline.** After the operators are chosen in the pipeline, the next step is selecting *configuration knobs* to achieve the best tradeoff between query performance and resource consumption [12, 26, 28, 45, 70]. In our example, configuration knobs include ground distance, the voxel size, and the choice of a 3D object detection model.

③ **Selecting physical placement.** The next step is placing pipeline operators across the heterogeneous edge infrastructure, starting from the device edge and to the cloud (Figure 1). This heterogeneity introduces complexity in placement of the operators and influences the *end-to-end latency performance* of the ML pipeline.

④ **Performing online adaptation.** After a query is deployed, its performance is affected by *runtime dynamics* due to resource changes and data variations. Resource changes are

A_{req}	L_{req} (ms)
0.8	400

(a) Performance Requirement

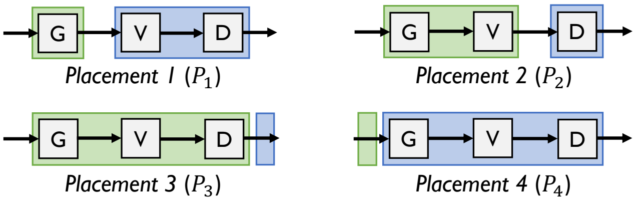


(b) Pipeline

Config	A	L (ms)	Size (MB)
C_1	0.85	260	6|4.4|3.2
C_2	0.8	270	6|3.6|2.2

(c) Query Configuration Results

Figure 2: (a) Performance requirement on accuracy (A_{req}) and latency (L_{req}) of the example query. (b) Query's pipeline ($\underline{I}nput \rightarrow \underline{G}roundRemoval \rightarrow \underline{V}oxelization \rightarrow Object\underline{D}etector \rightarrow \underline{O}utput$). (c) Offline profiling results. The last column records the size of the data at different stages of the pipeline: data at the source | after 'G' | after 'V'. Data after 'D' is not shown due to negligible size.



(a) Placement Choices

Config	Placement	A	L (ms)
C_1	P_1	0.85	480
C_1	P_2	0.85	420
C_1	P_3	0.85	520
C_1	P_4	0.85	560

	Config	Placement	A	L (ms)
	C_2	P_1	0.8	450
★	C_2	P_2	0.8	380
	C_2	P_3	0.8	540
	C_2	P_4	0.8	570

(b) End-to-end Performance Results

Figure 3: Placement choices and corresponding end-to-end performance. (a) All four feasible placement choices in a two-tier setting (Device Edge→Datacenter). (b) The baseline approach which first acquires the optimal combinations of configuration knobs during offline profiling (C_1) and then selects placement fails to meet the latency target. An ideal solution should select C_2 & P_2 by jointly optimizing configuration and placement. Note accuracy does not depend on placement and remains unchanged.

common because of other workloads on the edge infrastructure (e.g., 5G RAN containers) or network outages [39, 52]. The content of the data, such as lighting or object densities [12,28], can also change during the lifetime of a query. An ideal solution should adapt to runtime dynamics by adjusting the pipeline's filters, its configurations, and placement.

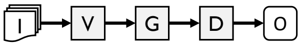
2.2 Motivating Examples

We now highlight some of the key challenges toward performing query planning for live ML queries using toy examples.

Jointly optimizing configuration and placement. Current practice considers pipeline placement and query configuration separately [12, 28, 33, 49, 67]. Designers first perform offline profiling to determine the optimal query configuration, and then select placement based on heuristics (prioritize network or compute, etc.) or greedy algorithms [67]. Such an approach fails to consider additional query latency introduced by pipeline placement, leading to sub-optimal query plans.

A_{req}	L_{req} (ms)
0.6	300

(a) Performance Requirement



(b) Pipeline

Config	A	L (ms)	Size (MB)
C_1	0.7	180	6|4|2.7
C_2	0.6	190	6|2.8|2

(c) Query Configuration Results

Config	Placement	A	L (ms)
C_2	P_2	0.6	290

(d) Optimal Query Plan

Figure 4: Given an updated performance requirement, a new pipeline is required to meet the latency target. The new pipeline swaps the order of the two filters ('V' & 'G'), delivering a different performance characteristic than the old one in Figure 2b.

To illustrate this, take an example autonomous driving query, whose performance requirements and pipeline are shown in Figure 2a and 2b. Figure 2c records the offline profiling results assuming only 2 sets of configuration knobs (C_1 and C_2) exist. In this case, the baseline approach identifies C_1 as the optimal configuration since it performs better both in terms of accuracy and latency than C_2. It then pairs C_1 with one of the placement choices in our simplified two-tier infrastructure shown in Figure 3a.

However, as shown in Figure 3b, none of placement choices would satisfy the query's end-to-end latency requirement if C_1 is selected. The end-to-end latency is composed of the compute latency (time spent in the object detector) and additional network latency occurred over the edge. For example, the additional network latency in P_1 is computed as the time it takes to transmit the output data of the ground removal module using the link bandwidth, which is set to 20MBps in this example.[1] An ideal solution should select C_2 by jointly considering both placement and configuration.

Constructing pipelines based on performance requirements. Different orderings of filtering operators lead to different performance characteristics of ML pipelines. Therefore, an ideal solution must construct the pipeline based on query-specific performance requirements. For instance, if we change the performance requirements of the original example query to a lower latency target with more tolerance on accuracy (shown in Figure 4a), then none of the query plans in Figure 3b can satisfy the new latency target as long as they use the pipeline from Figure 2b. Instead, we need to use a new pipeline which swaps the order of the two filters (Figure 4b). As filters are not independent to each other, placing 'V' before 'G' reduces more data, leading to better latency but lower accuracy (Figure 4c). Figure 4d shows the end-to-end results with the optimal query plan using the new pipeline. We omit for brevity the same process of finding the optimal placement and configuration as we did earlier.

Building an ideal solution that handles all the aforementioned aspects of live ML query planning is non-trivial, as selecting the right pipeline, placement, and configuration jointly

[1] In this example, the compute latency is assumed to be doubled when placing the detector on the device edge (i.e., in P_3).

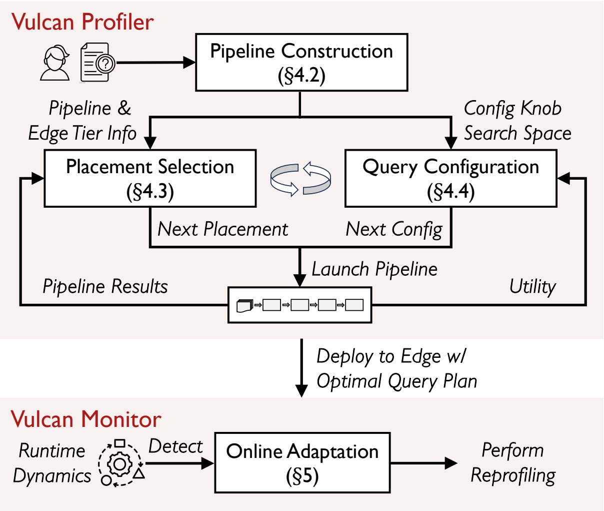


Figure 5: High-level workflow of Vulcan.

leads to a huge search space both during offline profiling and online adaptation. We next describe how Vulcan overcomes these challenges in a high-level system overview.

3 SYSTEM OVERVIEW

Vulcan is an ML analytics system that provides automatic query planning for live ML queries. It takes charge of the entire lifecycle of a ML query by constructing, configuring, and placing its ML pipeline, and performing online adaptation after the query is deployed.

Figure 5 presents a high-level system diagram of Vulcan. A user launches a live ML analytics task by submitting an input query to Vulcan, along with performance requirements. An example of Vulcan input queries can be found in Figure 1. Upon parsing the query, Vulcan Profiler generates its query plan by determining the query pipeline, placement of pipeline operators, and pipeline configuration. In Vulcan, query plans are evaluated using a utility function we define that combines the query performance and resource consumption (§4.1). To determine which pipeline to use for a query, Vulcan first constructs an initial pipeline by mapping user query specification to a general template optimized for performance and resource efficiency, and then determines the best ordering of filtering operators based on a new metric we define to capture the impact of filters on query latency and accuracy (§4.2).

Given a constructed pipeline, Vulcan jointly searches for the best placement and query configuration which, when combined, gives the highest utility. To explore placement choices with low cost, Vulcan reuses intermediate results from pipeline runs, such that a pipeline with the same configuration only needs to be offline profiled once. In the meantime, it also early prunes unpromising placement choices to further reduce the profiling cost (§4.3). For each placement, Vulcan searches for the best query configuration by leveraging Bayesian Optimization to explore a large number of query configurations with a small number of trials (§4.4).

After query deployment, Vulcan continues to monitor query performance to detect runtime dynamics. During such events, Vulcan reprofiles the pipeline in a quick and low-cost fashion by leveraging prior knowledge (§5).

4 VULCAN: PROFILER DESIGN

This section introduces the utility function we define to compare query plans, followed by how Vulcan construct, place, and configure the ML pipelines for live ML queries.

4.1 Defining Utility of Query Plans

We start by defining a utility function to evaluate the value of a query plan as we explore the search space. Existing literature has proposed various utility functions [9, 29, 50, 61] to combine query accuracy and latency. We build on top of these works and extend the utility function introduced in VideoStorm [70] to make it *resource-aware*. Resource consumption for live ML queries is as important as query performance, because each ML pipeline is deployed over edge infrastructure, where limited network and compute resources must be shared between applications. A query plan with good performance but excessive resources is undesirable.

Given a pipeline q with placement p and pipeline configurations c, we define $U_{q,p,c}$, the utility function of a query plan, as the ratio of the query performance to resource consumption:

$$U_{q,p,c} = P_{q,p,c}/R_{q,p,c} \tag{1}$$

such that the higher the utility value is, the better performance and cost for the query plan. $P_{q,p,c}$ combines query accuracy (A) and end-to-end latency (L) by calculating the reward (penalty) for achieving good (bad) performance based on a minimum accuracy target (A_m) and a maximum latency target (L_m):

$$P_{q,p,c}(A,L) = \gamma \cdot \alpha_A \cdot (A - A_m) + (1-\gamma) \cdot \alpha_L \cdot (L_m - L) \tag{2}$$

, where $\gamma \in (0,1)$. γ allows users to express their preference between accuracy and latency. $R_{q,p,c}$ combines the compute and network resource consumption of the pipeline:

$$R_{q,p,c} = \alpha_{gpu} \cdot R_{gpu} + \alpha_{net} \cdot R_{net} \tag{3}$$

The consumption of the compute (R_{gpu}) is calculated as the fraction of the GPU processing time used by the query. In Vulcan, we assume compute cost is dominated by GPU cost, as the queries we tackle rely heavily on GPU-based DNN models. The network resource consumption (R_{net}) is calculated as the sum of the fraction of the network bandwidth used by the pipeline on *each* network path between the edges. The constants α_A, α_L, α_{gpu}, and α_{net} are set by the operator to balance query performance and resource usage. Note that Vulcan's solution is orthogonal to the utility function and works for any utility function defined by the operator.

4.2 Determining the Query Pipeline

The first step in profiling is to construct the query pipelines. Vulcan performs pipeline construction by first generating an initial pipeline, and then determines the best ordering of pipeline operators to carry to the later profiling stages.

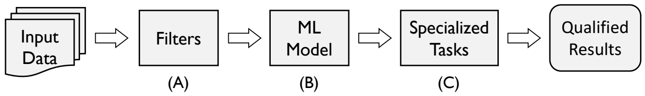


Figure 6: Template used by Vulcan to construct the initial pipeline.

4.2.1 Constructing the Initial Pipeline

Given a user query, Vulcan generates an initial pipeline using a general template with several types of building blocks, as shown in Figure 6. Starting from the data source, Vulcan constructs the initial pipeline by inserting (A) filtering modules which reduce the data size or data rate via sampling or filtering techniques, (B) the ML model to perform the actual inference task such as object detection, and (C) specialized modules for additional tasks required by the query, such as an object tracker (e.g., re-identification modules [31,38,40]), which can be performed only after the major ML inference task. Vulcan uses a pool of filter and ML modules that are readily available, provided by users, infrastructure providers, or third-party developers and organizations (e.g., public ML model zoos) to handle user queries. Filters and the ML model to use for the query is specified by the user in the input query (Figure 1). Based on the chosen operators, Vulcan generates a list of configuration knobs among which the profiler searches for an optimal set of configurations (§4.4).

The key insight behind arranging the building blocks in this way is to reduce the amount of data transfer across the edge *earlier* in the pipeline and leave operators with higher computation cost in *later* pipeline stages. This maximizes the savings in both network and compute resource as less data is transmitted and processed across the edge tiers. The design can improve end-to-end query latency by reducing the network latency as well as the GPU processing delay with potentially smaller data size for ML inference.

This initial pipeline leaves us with a follow-up given the impact of filter ordering on query performance (§2.2): *In what order should we place the filters?*

4.2.2 Selecting the Ordering of Filters

A naïve solution for selecting the filter ordering is to explore pipeline placement and configuration for all possible orderings; this, however, does not scale as the number of filters increases. In Vulcan, we propose a new solution that compares the impact of different filter orderings on query's accuracy, latency, and resource consumption by evaluating ***recall*** and ***precision*** of filters. We first explain how it works for a single filter before moving on to the multi-filter scenario. We define recall of a filter as the fraction of samples in the original data that contains the objects of interest (i.e., relevant data) that passes through the filter. On the other hand, precision of a filter is the fraction of data samples in its output that contains relevant data. For a given filter, we would expect its recall to be high such that it still captures most of the desired data, and query *accuracy* is preserved. A filter with low recall drops true positive samples which cannot be recovered later in the pipeline. Among filters with the same recall, we prefer the ones with higher precision because these filters provide higher data reduction rate by picking up fewer irrelevant samples, leading to better *latency and resource savings*.

We can now define the metric to evaluate how a given filter affects a query plan's utility ($U_{q,p,c}$). The metric should also handle a query's preference between latency and accuracy, based on the parameter γ defined in Eq (2) in our utility function. To this end, we leverage a variation of the F-measure in information retrieval theory [58] to encode this preference. Denote F_γ as the score for a given filter with its precision and recall measurements:

$$F_\gamma = (1+\beta^2)\cdot\frac{\text{precision}\cdot\text{recall}}{(\beta^2\cdot\text{precision})+\text{recall}} \tag{4}$$

where $\beta = \gamma/(1-\gamma)$. In the F-measure definition, the value of β captures how many times recall is considered more important than precision. As we explained earlier, recall and precision corresponds to query accuracy and latency respectively, and thus the value of $\gamma/(1-\gamma)$ (see Eq (2)) is used to capture our accuracy-latency preference.

Measuring F_γ tells us how well a single filter fits into a query's optimal query plan. However, two challenges remain. First, directly applying F_γ to sort a sequence of filters does not work well as the recall of a filter can change based on its preceding filter. Second, filters may have their own configuration knob that leads to different precision or recall measurements. Applying one set of configuration for all filters oversimplifies the problem with inaccurate estimation, whereas evaluating too many configuration sets increases profiling cost.

We address multiple filters by treating a sequence of filters as a *bulk filter* with input being the data source and the output being the one from the last filter, and measure the overall F_γ using representative data for each permutation of the available filters. To deal with filters with various configurations, we choose a few representative configuration settings to capture the effect of configuration knobs on filters, where in each setting all the filters are configured at the x-th percentile in the range of their configuration knob. Filters with no configuration parameter remain unchanged. We empirically find x = 20-th, 50-th and 80-th good enough at picking promising pipelines, which results in {3 × total number of filter ordering} F_γ values to collect. Vulcan then picks the ordering which achieves the highest F_γ to complete the pipeline construction.

4.3 Determining Placement Choices

After constructing the pipeline, the next step is placing each of the pipeline operators across the edge infrastructure (see Figure 5). On the one hand, rule-based solutions are good at reducing search cost but may fail to explore all promising placement choices. On the other hand, exhaustively searching through all placement choices requires high search cost during profiling as we deploy the query across the edge. Vulcan combines the benefits of the two approaches by (i) reducing search cost by reusing intermediate results from pipeline runs, and (ii) early pruning unpromising placement choices.

Algorithm 1: Placement Selection

```
1  Notation:
     $q$: constructed pipeline (from §4.2.1),
     $\mathcal{P}$: all feasible placement choices given q,
     $A_m$: accuracy target, $L_m$: latency target, $U$: utility,
     $res$: pipeline results used to calculate utility
2  Function $SelectBestPlacement(q,\ A_m,\ L_m)$:
3    $\mathcal{P} \leftarrow GenerateAllPlacementChoices(q)$
4    foreach placement $p \in \mathcal{P}$ do
5      $c \leftarrow NextConfig()$    ▷ from §4.4
6      $count = 0$    ▷ reset early pruning counter
7      if $c.hasExplored()$ then
8        $res = LoadFromCache(q,\ c)$
9      else
10       $res \leftarrow LaunchPipeline(q,\ c)$
11       $CachePipelineResults(res)$
12     $U_{p,c} = CalculateUtility(q,\ p,\ res)$
13     if $U_{p,c} \leq U_{p,c}(A_m, L_m)$ then
14       $count \leftarrow count + 1$
15       if $count \geq EarlyPruneCount$ then
16         continue
17   return $\arg\max_{p \in \mathcal{P}, c} U(p,c)$
```

4.3.1 Reusing Pipeline Results

The idea of reusing pipeline results is based on two key observations we make in live ML pipelines. First, query accuracy does *not* depend on the placement choices of a pipeline with the same query configuration. Second, the amount of data generated after each pipeline operator is independent of placement choices. These observations allow us to deploy the pipeline ***offline** in the datacenter **only once** per selected query configuration* during the profiling stage to collect pipeline operator results, and *reuse* those results to evaluate a new placement choice by calculating additional *latency and resource consumption components* introduced by the placement, while reusing the same query accuracy result. Given a total of M placement choices and N combinations of pipeline configurations, our solution improves the search complexity from $O(MN)$ to $O(N)$ for a given pipeline.

Algorithm 1 describes how Vulcan evaluates placement choices. Given a pipeline, we begin with generating a collection of all feasible placement choices. Obviously infeasible choices where operators are placed in a different order than they appear in the pipeline (e.g., placing the DNN model in front of the filters) are excluded from the collection. For each placement choice, Vulcan explores promising query configurations to evaluate the utility of the query plan (details of how Vulcan picks query configurations are in Section 4.4). For every new set of pipeline configurations, Vulcan launches the pipeline inside the datacenter. In this case, Vulcan not only collects query performance and resource consumption for utility calculation but also caches intermediate results from each pipeline operator, including the operator's output size, output bandwidth, and data processing time (i.e., time spent in a filtering module or GPU inference time)[2] (Algorithm 1 lines 10-11). If a chosen pipeline configuration c has been explored by a previous placement choice, Vulcan calculates the utility for the new placement p by estimating the new query end-to-end latency $L_{p,c}$ and resource consumption $R_{p,c}$ without launching the pipeline (lines 7-8,12). $L_{p,c}$ is estimated by summing up the total processing time of each operator measured offline excluding the DNN module, the additional network latency introduced by placement, and the updated GPU inference latency as shown below:

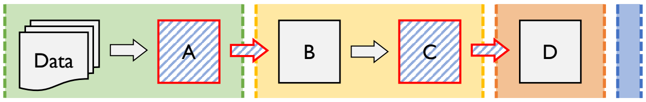


Figure 7: An example of how additional network latency is calculated for a pipeline with 4 operators placed across the edge infrastructure (i.e., device edge → on-premise edge → public MEC → cloud). Output sizes of shaded operators are used to calculate the additional latency introduced by the placement.

$$L_{p,c} = L_{offline,total} - L_{offline,gpu} + \sum L_{p,net} + L_{p,gpu} \qquad (5)$$

The network component of the new latency, $\sum L_{p,net}$, is calculated by summing up the network latency going across two adjacent tiers. Figure 7 illustrates how this process works. The latency is calculated by taking the ratio of a component's output size (cached per configuration) to the assigned link bandwidth capacity the query data traverse through. Note that only the components sending data to the next tier in the infrastructure are considered. The GPU inference latency, L_{gpu}, is updated by multiplying with a coefficient based on the GPU type to reflect the performance difference, which we determined by profiling all GPUs available in our cluster. $R_{p,c}$ is estimated in a similar way by including additional network bandwidth and GPU processing time introduced by the placement. After all placement choices are explored, Vulcan selects the placement (together with the optimal configuration) that achieves the highest utility (line 17).

4.3.2 Pruning Unpromising Placement Choices

Although caching intermediate results allows Vulcan to never re-launch a pipeline with the same configuration, exhaustively exploring all feasible placement choices may still incur large search cost as the profiler strives to find a good configuration for an unpromising placement. For example, a query preferring latency performance will not favor the placement that places the filtering module too far away from the data source.

To apply early pruning, we set a utility threshold which is equal to the utility value when both minimum performance metrics is used for the given query configuration and placement, namely $U(Q_m, L_m)$, which is evaluated to zero in our utility definition (§4.1). When the profiler obtains N utility values below the threshold, we early prune the current placement

[2] Vulcan also collects the size and bandwidth of the data source to account for the case where only the data source is placed on the first tier.

choice (Algorithm 1 lines 13-16). Compared to other alternative pruning solutions, such as building a statistical model for pruning decisions, this simple scheme works well because Vulcan profiler is designed to always pick more promising configuration than its last attempt (§4.4), and consecutive bad utility values indicate a high probability of unpromising placement. This also allows us to set N to a small value ($N = 3$ in our implementation) as otherwise the profiler would have already terminated itself after a small number of attempts.

Placing split ML models. Recent research [21] has proposed the idea of splitting the layers of a large DNN model and placing them at different edge tiers. Vulcan can handle such design as long as the split layers are properly specified in the input queries as individual modules.

4.4 Determining Query Configuration

For each placement choice, Vulcan leverage Bayesian Optimization (BO) [14, 51] to efficiently explore pipeline configurations that achieve the best performance with minimized resource consumption (see Figure 5).

4.4.1 Why Do We Choose BO?

BO is a methodology for optimizing expensive objective functions and is widely applied to many computer systems [8, 13, 35, 42, 48, 64]. At a high level, it learns the shape of the objective function by picking inputs that has the highest probability of reaching the global maximum of the function. As BO accumulates more observations, it becomes more confident on the actual shape of the objective function. This is a good fit for our solution as we need to quickly find the optimal configuration that maximizes the utility function without exploring too many choices.

More importantly, we observe BO has many advantages over other popular optimization schemes in the context of live ML queries. Greedy Hill Climbing has been applied in query configuration for video analytics queries [70], but it is purely exploitative and cannot perform as efficient global exploration as BO does (§6.3 provides detailed evaluation). Multi-Armed Bandit (MAB) [25] is another popular optimization scheme for sequential decision making, but it is designed for optimizing cumulative rewards, contrasting our goal of finding the one configuration with the highest utility. In addition, BO tunes the entire set of input configurations all together for each iteration no matter how large the input vector is, whereas MAB can only adjust one knob at a time. We also prefer BO over population-based optimization scheme such as Particle Swarm Optimization (PSO), as applying PSO in query configuration requires launching many ML pipelines in parallel, leading to a much high computation cost (e.g., GPU resourse).

4.4.2 Applying BO to Query Configuration

We define the objective function of BO as $f(\vec{x})$, which models how good a given query plan is based on a given pipeline and a physical placement choice. The input $\vec{x}$ is the set of query configuration knobs, and the output of f is the utility value, $U_{q,p,c}$ for a given pipeline q with placement p and a set of configurations c. For each iteration, Vulcan launches the pipeline with the configurations suggested by BO (i.e., $\vec{x}$), and collects the measurements to compute $U_{q,p,c}$, which is then fed back to BO as the new observation.

Choice of prior and acquisition functions. Internally, BO learns an objective function by leveraging a *prior function* and an *acquisition function*. The former represents the belief about the space of possible objective functions, whereas the latter guides BO to choose the next promising input where the value of acquisition function is maximized. We choose Gaussian Process as the prior function and use *Matern 5/2* as its covariance function to describe the smoothness of the prior distribution, which is known to perform well among systems applying BO [8, 64]. Among three major choices of the acquisition function, namely probability of improvement [36] (PI), expected improvement [30] (EI), and upper confidence bound [65] (UCB), we select UCB as it works the best for our workloads. We defer a dynamic approach of selecting acquisition functions [15] to future work.

Starting and stopping BO. We start with N random sets of input query configurations as initial observations for BO to learn the rough shape of the objective function. We set $N = 3$ in all of our experiments and found it works well for various workload settings. Vulcan stops BO when the improvement of the utility value is less than a threshold for a few consecutive runs (i.e., 10% for 5 consecutive runs, which empirically works well). We include a sensitivity analysis on how the parameters we use in the starting and stopping conditions affect BO's performance in Section 6.7.

One alternative design which seems promising but we do not consider is to directly apply BO for placement selection and pipeline construction by treating available placement choices and pipelines as another two knobs in the total configuration space. In BO, input parameters are specified in a continuous range. For example, the voxel size in the voxelization module of an AD perception pipelines is chosen between 0.1 and 0.5. As BO moves within this range, the behavior of the module changes with the size of the voxel, leading to a smoother shape of the objective function with more predictable outputs. In contrast, the behavior of different placement choices or filter orderings is hard to predict, and going through the configuration space of those (i.e., placement or pipeline choices indexed by numbers) leads to very rough shape of the objective function for BO to grasp; thus it becomes much more difficult to find the global optimum.

5 ONLINE ADAPTATION

This section describes how Vulcan performs online adaptation to handle runtime dynamics after query deployment. Vulcan currently does not support concurrent execution of different ML queries, and we defer the support of multi-resource sharing among concurrent live ML pipelines to future work.

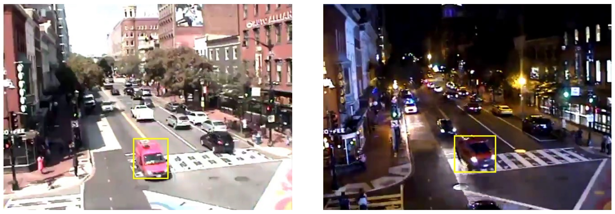


Figure 8: Scenes taken during different time of day from a video query detecting red vehicles.

5.1 Detecting and Handling Runtime Dynamics

Vulcan leverages two design ideas to quickly converge back to the best query plan during online adaption: (i) monitor utility change to detect runtime dynamics, and (ii) leverage prior knowledge during reprofiling.

Detecting runtime dynamics. Vulcan detects runtime dynamics by monitoring the change in a query's utility values, as any runtime dynamics (content, network, or compute based) leads to a change in utility. This includes all the measurements required to calculate up-to-date utility (i.e., latency, accuracy, module output size and bandwidth), and they are periodically reported to Vulcan monitor via HTTP requests. To obtain real-time ground truth on query accuracy given unlabeled live data, Vulcan launches a duplicated pipeline with the most expensive configuration inside the cloud, which only receives live data periodically to minimize network cost. A substantial change in the utility triggers reprofiling which deploys the query in the cloud similar to the case of offline profiling. We set the threshold of utility change empirically (10% in our implementation) via profiling.

Leveraging prior knowledge in reprofiling. We observe that most of the query, such as the object of interest and where the query takes place, remains the same after deployment; meaning, we can take the advantage of prior knowledge from offline profiling. Figure 8 shows two example scenes taken by the same camera during different time of the day from a query that detects red vehicles. We observe a high level of similarity between the two scenes except for environment illumination. Let us define the distance between two configurations, C_A and C_B, to be the total number of steps needed for each configuration knob in C_A to change to each knob value in C_B. In this example, applying the same configuration from daytime to nighttime scenes leads to an average 26.2% utility drop among all placement choices, but it requires only an average distance of 2.47 steps to converge back to the query plan with the highest utility (figures omitted in the interest of space).

To apply prior knowledge, Vulcan makes the following changes to the normal profiling process. Vulcan keeps track of the most recent top-K and worst-K configuration per placement choices ($K = 3$), and applies them as initial data points in BO such that BO can quickly grasp the shape of the objective function. We also tune up the exploitation factor in the acquisition function (κ in UCB [65]) to $1/10$ of its original value to focus more on exploitation than exploration. Note that Vulcan does not just stick to the best placement choice but choose to re-perform placement selection. As runtime dynamics can involve network and compute resource changes, this allows Vulcan to change the placement of the pipeline when merely adjusting query configuration makes little impact on recovering query performance. We evaluate how Vulcan performs under different types of runtime dynamics in Section 6.6.

```
HttpProcessor _httpProcessor = new HttpProcessor(
     Config.ENDPOINT_URL_SAMPLING);
Int sampling_rate = 3; // selecting 1 out of 3 frames
var form = new MultipartFormDataContent();
form.Add(new StringContent(sampling_rate.ToString()),
     "frameSamplingRate");
var result = await _httpProcessor.PostAsync(form);
```

(a) Sending Configuration Updates

```
// POST: api/Config
[Produces("application/json")]
[HttpPost]
public async Task<IActionResult> PostConfig([FromQuery
     ] string frameSamplingRate = null)
{
  if (frameSamplingRate != null)
  {
    Config.FRAME_SAMPLING_RATE = Int32.Parse(
     frameSamplingRate);
    /* sampling rate updated for subsequent frames */
  }
  return Ok();
}
```

(b) Posting Configuration Updates

Figure 9: Code snippets of Vulcan APIs on dynamically updating query configuration. (a) Vulcan Profiler sending the configuration updates to deal with runtime dynamics. (b) Vulcan Controller at the container updates the configuration to use the updated value.

5.2 Enabling Online Adaptation

After identifying the right query plan upon runtime dynamics, we must also enforce this at production scale. Unfortunately, existing frameworks for large-scale deployment, such as Kubernetes [7], do not support container modification during execution. Therefore, Vulcan adds its own implementation.

Dynamically updating query configuration. Figure 9 shows the example interfaces of Vulcan sending configuration updates and posting them inside the container upon a change in the frame sampling rate of a traffic monitoring query. In Vulcan, each pipeline module is installed and launched by a container. Query configurations are updated in real time without stopping the containers. If any configuration knobs need to be updated after reprofiling, Vulcan sends out updated configuration using HTTP requests to the containers that launch the corresponding modules (Figure 9a). The exact location (ENDPOINT_URL_SAMPLING) of the containers is acquired when Vulcan first deploys the pipeline. The containers receiving the request update the configuration right away, without stopping the current ML task (Figure 9b).

Handling placement changes. If the new query plan involves a placement change (e.g., during network or compute resource change), Vulcan migrates the container of the corresponding module to the updated tier. To perform the migration, Vulcan

Table 1: ML model variations used in evaluation.

Model	# Parameters (Million)				
YOLO [56]	**v5n6**	**v5s6**	**v5m6**	**v5l6**	**v5x6**
	3.2	12.6	35.7	76.8	140.7
PointPillars [37]	**SECFPN**	**SECFPN (FP16)**			
	4.9	4.9			
SSN [73]	**SECOND**	**RegNet**			
	6.2	7.1			
CenterPoint [68]	**DCN**	**Circular NMS**			
	6.0	6.1			
wav2vec2 [10]	**base**	**large_10m**	**large_960h**		
	94.4	315.5	315.5		
HuBERT [69]	**large**	**xlarge**			
	315.5	962.5			

first launches the module with the same query configuration on the target tier. It then updates the location of the new container to the upstream module via the same API for configuration updates described in §5.1, before removing the old container on the current tier. The new location of the container is also updated in Vulcan monitor for future adaptation.

6 EVALUATION

We evaluate the effectiveness of Vulcan in performing automatic query planing in terms of profiling time, query performance, and query resource consumption. Our key findings:

(1) Vulcan generates query plan with better profiling cost by 4.1×-30.1× over state-of-the-art ML analytics systems while delivering up to 2.0×-3.3× better latency (§6.2).

(2) Vulcan performs better query configuraiton, placement selection, and pipeline construction by outperforming existing solutions with up to 2.8× better query latency and 174× lower network resource consumption (§6.3-§6.5).

(3) Vulcan achieves consistently better 99^{th}-p latency performance (by up to 2.5×) during online adaptation over the-state-of-the-art (§6.6).

6.1 Experiment Setup

Live ML Queries. We illustrate Vulcan's performance in performing query planning for three example live ML queries:

- *Video Monitoring*: monitors the traffic volume by examining live video frames and counting the vehicles of a specific color (white color in our examples).
- *Autonomous Driving Perception*: takes 3D point cloud as input and generates a real-time perception (represented as 3D bounding boxes) surrounding a vehicle.
- *Automatic Speech Recognition*: converts live human speech (with background noises) into written text.

Datasets. We adopt several real-world datasets for each query example to perform a comprehensive evaluation. The video monitoring queries use videos captured by traffic cameras among different metropolitan areas in Bellevue and Washington D.C. The autonomous driving queries use LiDAR sensor data from nuScenes [16]. Automatic speech recognition queries use the VOiCES dataset [57] with background noise enabled. Appendix A.1 has more details.

Pipelines. Unless otherwise specified, video monitoring queries in our evaluation experiments use the same pipeline which consists of the following components in order: a background subtractor to detect moving vehicles, a color filter, and a variations of YOLOv5 [5] object detector. The autonomous driving pipeline feeds 3D point clouds into a ground removal module, followed by a voxelization module and a variations of PointPillars [37], SSN [73], or CenterPoints [68] 3D object detector. The speech recognition queries use a pipeline that consists of an audio sampler, a noise reduction module, a variation of wav2vec2 [10] or HuBERT [69] model, and a decoder. Table 1 records all ML model variations we use in the evaluation. Appendix A.1 describes the details of the query configuration knobs for all queries.

Baselines. We consider the following baselines for Vulcan.

• *Exhaustive search.* To determine the optimal placement and configuration for a ML pipeline, we use exhaustive search to explore all possible placement choices and configurations.

• *ML analytics systems.* We implement four state-of-the-art ML analytics systems: VideoStorm [70], Chameleon [28], JellyBean [67], and LLAMA [60]. Both VideoStorm and Chameleon adopt a variant of greedy-hill climbing during query profiling, and Chameleon applies spatial and temporal correlations to reduce the search cost during online adaptation. JellyBean selects models with target accuracy and lowest cost, and applies beam search to determine query placement in a greedy fashion. LLAMA dynamically explores query configuration by computing per-invocation pipeline latency.

• *Pipeline placement strategies.* Besides JellyBean's greedy placement, we implement the following commonly-adopted placement strategies: (1) *prioritizing network (PN)* which places operators closer to the device edge, (2) *prioritizing compute (PC)* which places operators closer to the cloud, and (3) *balancing network and compute (NC)* which places filtering modules closer to the device edge and the remaining operators to the cloud.

Emulation Setup. We emulate a setup of 4 edge tiers consisting of the device edge, the on-prem edge, the public MEC, and the cloud, which is consistent with our production edge infrastructure by adding additional network latency and compute overhead. The network bandwidth and compute cost are emulated based on the real network and hardware settings in our production infrastructure.

We run all experiments 20 times with different random seeds, and collect 10th, 50th, and 90th percentiles of data to include the effect of randomness in Vulcan and other baselines. The 10th and 90th percentiles are plotted via error bars unless otherwise specified. We set L_m for video monitoring, autonomous driving, and speech recognition queries as 2000ms, 400ms, and 500ms, respectively. Q_m is set at 0.8 × the accuracy achieved by the most expensive configuration for each type of queries. We set γ (preference of query accuracy over latency) to be 0.5 for all queries unless otherwise specified.

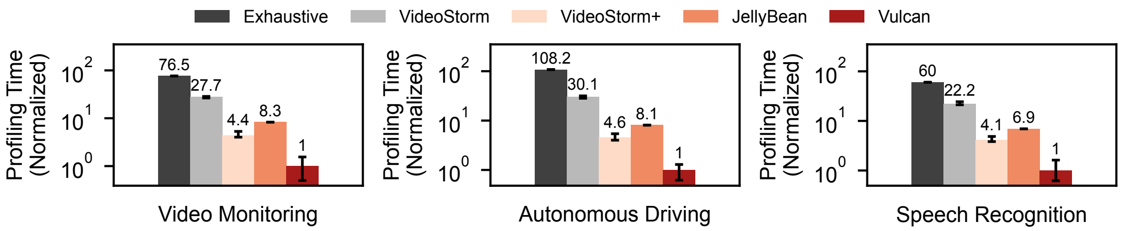


Figure 10: Comparing the profiling cost of video monitoring, autonomous driving, and speech recognition queries.

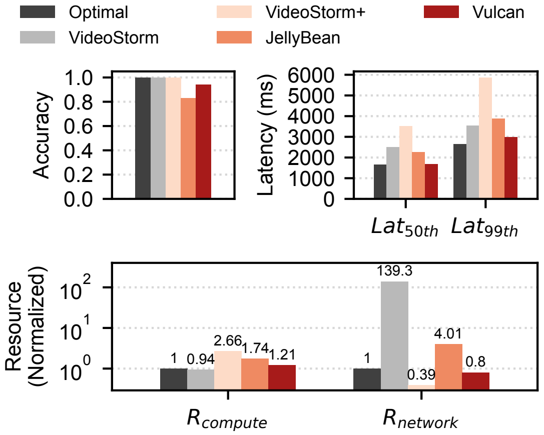


Figure 11: Comparing end-to-end performance and resource consumption for video monitoring queries.

6.2 End-to-End Improvement

We start with showing the end-to-end improvement of Vulcan over other baselines in generating query plans for all three types of queries. We compare exhaustive search, original VideoStorm that explores all placement choices, VideoStorm+ that explores a combination of all three baseline placement strategies (i.e., PN, PC and NC) on top of VideoStorm, and JellyBean. We compare Vulcan with Chameleon and LLAMA during online adaptation in §6.6 as their query configuration by design happens in the online phase.

We record profiling time normalized by the time spent by Vulcan, which achieves the smallest value in both types of queries. We also record performance (accuracy, median latency, and 99^{th}-p latency) and resource consumption (network and compute) achieved by the query plan and normalize the results based on the optimal query plan generated by exhaustive search. For VideoStorm, VideoStorm+, and JellyBean, we record the query plan achieved with highest utility given the same profiling time as Vulcan.

Figure 10 records the profiling cost for all types of queries, and Figure 11 summarizes the performance and resource consumption for video monitoring queries. Results for autonomous driving and speech recognition queries are similar and moved to Appendix A.2 in the interest of space. Vulcan achieves significantly lower profiling cost than exhaustive search, VideoStorm, VideoStorm+, and JellyBean by at least 60×, 22.2×, 4.1×, and 6.9× respectively across all three types of queries, and its query performance and resource consumption are very close to the optimal one achieved through exhaustive search, achieving up to 2.0×-3.3× better tail latency than other baseline approaches. Vulcan's performance improvement comes from its joint optimization of pipeline placement and configuration with low cost. Vulcan avoids exhaustively searching for optimal placement by reusing pipeline results. VideoStorm+ improves profiling time by only exploring a few placement choices, at the cost of worse latency and network resource consumption, and it is still outperformed by Vulcan by at least 4.4×. JellyBean's placement algorithm greedily finds promising placement choices but separately optimizes query configuration and placement, leading to sub-optimal latency, accuracy, and resource usage.

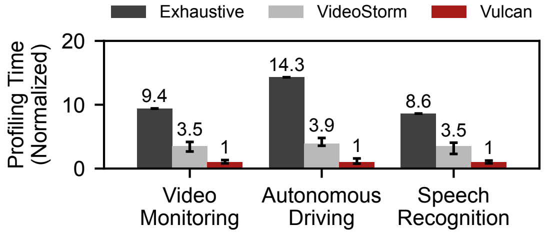


Figure 12: Profiling cost of query configuration given the same pipeline and placement.

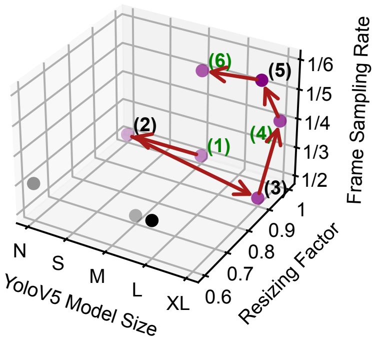


Figure 13: BO's search path in selecting configurations. The new maximum in utility is marked in green. The initial random configurations are shown in black.

6.3 Selecting Better Query Configurations

We next delve into the performance of each profiling components in Vulcan, starting with query configuration. Figure 12 shows the profiling cost among Vulcan and other baselines with the same pipeline and the best placement choice. Even without the benefits achieved in efficient placement selection, Vulcan can still outperform VideoStorm, which leverages greedy hill climbing, by 3.5× in profiling time. To illustrate how Vulcan achieves this, we plot one of Vulcan's example

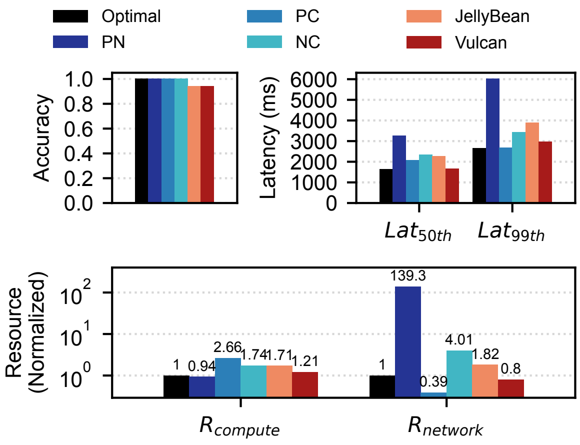


Figure 14: Comparing Vulcan with different placement strategies on serving video monitoring queries.

BO search path in the video monitoring query in Figure 13. BO focuses its exploration on larger resizing factor after verifying that smaller values lead to worse performance, and finds the optimal query configuration at the 6^{th} step. BO takes 5 more steps to confirm we are not likely to encounter better results before stopping, which we do not plot for legibility.

6.4 Selecting Better Placement

We evaluate Vulcan's placement decisions by comparing with common placement strategies (§6.1) and JellyBean. Each baseline placement strategy uses exhaustive search to find the optimal pipeline configuration, and all comparisons use the same pipeline. We ignore the profiling time improvement of Vulcan over other baselines and only focus on comparing the achieved performance and resource consumption. Figure 14 illustrates the results of video monitoring queries. Results for autonomous driving and speech recognition queries are similar and shown in Appendix A.3. We observe that PN prioritizes network latency but fails to consider the large ML inference latency on slower compute nodes, resulting in worse end-to-end query latency. PC optimizes ML inference latency, which does not always lead to better latency. Moreover, it consumes significantly more network resources. NC tries to strike a balance between PN and PC but still achieves worse latency. JellyBean's placement algorithm is based on a fixed query configuration that is determined in a prior stage, leading to a sub-optimal placement choice. In comparison, Vulcan always achieves the same placement choices as the exhaustive search and delivers up to 2.8× better query latency and 174× network resource consumption than other baselines, thanks to its joint optimization between placement and configuration. The performance gap between Vulcan's query plan and the optimal query plan is caused by Vulcan picking a different pipeline configuration, in which case Vulcan takes much less profiling time with similar performance and resource consumption.

6.5 Selecting Better Pipelines

We now evaluate how well Vulcan selects the filter ordering during pipeline construction based on performance requirements of the queries. We compare the performance of fixed pipelines with Vulcan's choices, which is dynamically determined based on F_γ in Eq 4 (§4.2). To focus on whether Vulcan makes the correct choice of filter ordering, we remove the potential noise of BO by using the best pipeline configuration determined by exhaustive search. We sweep γ and record the utility value achieved by picking the corresponding pipeline. Figure 15 shows that Vulcan is able to select the best pipeline in 8 out of 9 cases. On the other hand, sticking to the other two fixed pipeline settings leads to only 6 out of 9 cases and 3 out of 9 cases for selecting the correct filter ordering.

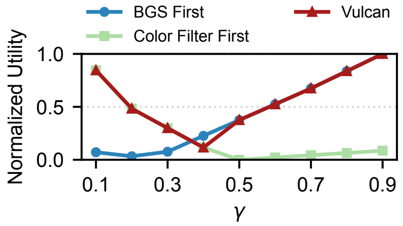


Figure 15: Compare Vulcan's selection of filter ordering with fixed pipeline settings in video monitoring queries.

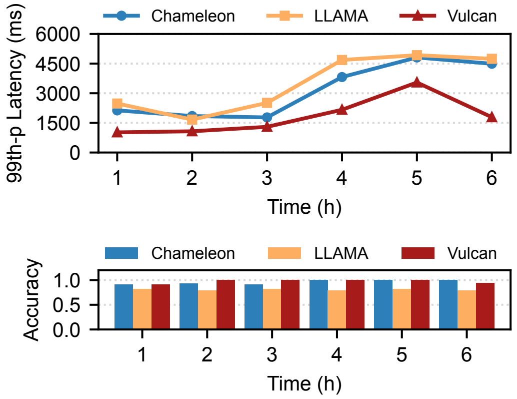
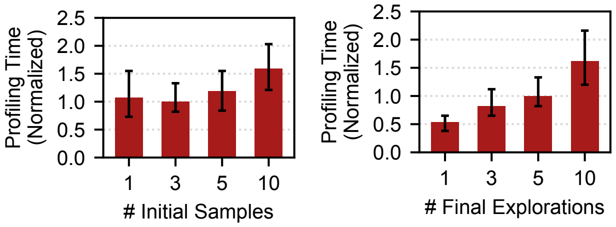


Figure 16: Comparing Vulcan with Chameleon during online adaptation for video monitoring queries.

6.6 Handling Runtime Dynamics

To evaluate Vulcan's online adaptation, we compare it with Chameleon and LLAMA, the state-of-the-art solution in adapting runtime dynamics in ML queries. We took a continuously running 6-hour long video from our Washington D.C. video dataset to monitor red vehicles. Performance (99^{th}-p latency and accuracy) was collected and reported at the end of every hour. All three systems started with the same video monitoring pipeline and placement, where the background subtractor and the color filter are placed on the device edge, and the object detector is placed on the public MEC. At the end of the 3rd hour, we reduced the link capacity from the on-premise edge to the Public MEC by 20× to simulate a network outage. We see in Figure 16 that Vulcan achieves consistently better latency than Chameleon and LLAMA by up to 2.5×. Neither Chameleon nor LLAMA updates pipeline placement upon resource changes, causing worse latency after the outage. On the other hand, Vulcan adjusted the pipeline



(a) Starting Condition **(b)** Stopping Condition

Figure 17: Sensitivity analysis of BO's parameters.

placement by placing the object detector onto the on-premise edge after the outage, thus mitigating the latency hikes.

6.7 Sensitivity Analysis

We analyze the sensitivity of two parameters used in BO during pipeline configuration in Figure 17. We use the same experiment setting in Figure 12 with video monitoring queries. Figure 17a plots the profiling time normalized by the value picked in Vulcan implementation when sweeping the number of initial random configurations passed to BO. The number of random configurations does affect the profiling time but has a negligible impact on the quality of the query plan achieved. Figure 17b sweeps the stopping threshold of BO, which is the number of consecutive rounds without significant improvement. BO needs at least 3 rounds to obtain a good chance of finding optimal configurations, but adding more rounds won't further improve the quality of profiling.

7 RELATED WORK

Configuration management for ML analytics. Finding better configurations for ML analytics is a well studied research topic [28, 70]. VideoStorm [70] investigates query's accuracy-latency profile and applies a greedy hill-climbing approach to search for query configurations. Chameleon [28] applies spatial and temporal correlations of video frames to reduce the search cost of query configurations during online adaptation. Although VideoStorm and Chameleon works well for ML quries with fixed pipeline placement, Vulcan jointly explores placement and configurations for live ML queries deployed across a heterogeneous infrastructure.

Constructing ML pipelines with declarative queries. Researchers have proposed declarative query languages, especially for video analytics, to construct ML pipelines [11, 19, 44, 59]. MIRIS [11] provides a declarative interface to select video object tracks and selects corresponding modules based on user specification. Viva [59]'s declarative interface allows users to specify relational hints to express the relationship among different modules, which helps Viva to construct and optimize the pipeline. However, those works only consider accuracy requirement when optimizing pipelines and ignore the resource demands especially network usage, providing no guarantees to end-to-end latency performance.

Domain-specific optimization for ML analytics Applying domain-specific knowledge to perform optimization for ML analytics is an active research area spanning many use cases including video analytics [11, 19, 28], autonomous driving perception [55, 71, 72], and automatic speech recognition [20, 41]. BlazeIt [19] leverages spatiotemporal information of objects in video to optimize aggregation and limit queries. VI-Eye [72] achieves better accuracy and latency performance by exploiting domian knowledge in autonomous driving scenarios to recognize key semantic objects that can be used to align vehicle-infrastructure point cloud pairs. Vulcan's design is orthogonal to domain-specific techniques and can be applied to different use cases without additional changes.

Continuous learning for ML analytics. Another line of work that gets increasingly more attention nowadays is continuous learning in ML analytics [12, 34, 46]. Ekya [12] jointly schedules and allocate resources to ML retraining and inference to handle data drift. RECL [34] integrates model reusing with model retraining to quickly adapt to a lightweight expert DNN model for each specific video scenes. Techniques leveraging continuous learning is complementary to Vulcan and can be applied to Vulcan's online adapation phase. On the other hand, Vulcan's design can be applied to those works as well, including dynamically updating query configurations and fast detection of data changes by monitoring utility changes.

Reducing cost of ML analytics via filtering. There have been recent studies on applying different types of filtering techniques to reduce the resource consumption of ML analytics without compromising accuracy [17, 24, 26, 32, 47]. NoScope [32] searches for and trains a cascade of models that preserves the accuracy of the ML inference but with far less computation cost. Focus [26] uses cheap convolutional network classifiers (CNNs) to construct an approximate index of all possible object classes in the video frame, which reduces the use of expensive CNNs during query time. Probabilistic Predicates [47] constructs and applies binary classifiers to filter out data blob that will not pass query predicate and thus accelerate queries with expensive user-defined functions. Vulcan builds up on the idea of applying filtering and propose a novel technique to order the filters for ML pipelines.

8 CONCLUSION

Serving live ML analytics involves constructing, placing, and configuring ML pipelines as well as their online adaptation. However, existing query planning solutions for live ML queries remain elusive with piecemeal and sub-optimal. We present Vulcan, an ML analytics system that performs automatic query planning for live ML analytics. Vulcan automatically construct the pipeline and determine the best ordering of filtering operators for query performance. It efficiently explores placement choices by reusing of intermediate pipeline profiling results, and leverage Bayesian optimization with prior knowledge to handle query configuration and online adaptation. Vulcan outperforms state-of-the-art solutions on profiling time, query latency, and resource consumption when serving queries with real-world datasets.

ACKNOWLEDGEMENTS

We would like to thank Jae-Won Chung, the anonymous NSDI reviewers, and our shepherd, Danyang Zhuo, for providing valuable feedback. This work was supported in part by NSF grants CNS-1845853, CNS-2104243, and CNS-2106184.

REFERENCES

[1] Microsoft rocket for live video analytics. https://www.microsoft.com/en-us/research/project/live-video-analytics/, 2020.

[2] Build modern connected applications at the edge with 5g. https://azure.microsoft.com/en-us/blog/how-developers-can-benefit-from-the-new-5g-paradigm/, 2022.

[3] Edge video service (evs). https://azure.microsoft.com/en-us/blog/microsoft-and-att-demonstrate-5gpowered-video-analytics/, 2022.

[4] Microsoft rocket for live video analytics. https://azure.microsoft.com/en-us/blog/microsoft-and-att-are-accelerating-the-enterprise-customer-s-journey-to-the-edge-with-5g/, 2022.

[5] Yolov5. https://github.com/ultralytics/yolov5, 2022.

[6] Azure public multi-access edge compute (mec). https://azure.microsoft.com/en-us/solutions/public-multi-access-edge-compute-mec/, 2023.

[7] Kubernetes. https://github.com/kubernetes/kubernetes, 2023.

[8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[9] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS*, 2006.

[10] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations, 2020.

[11] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. Miris: Fast object track queries in video. In *SIGMOD*, 2020.

[12] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *NSDI*, 2022.

[13] Eric Brochu, Tyson Brochu, and Nando de Freitas. A bayesian interactive optimization approach to procedural animation design. In *SCA*, 2010.

[14] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning, 2010.

[15] Eric Brochu, Matthew W. Hoffman, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *UAI*, 2011.

[16] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[17] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya Dulloor. Scaling video analytics on constrained edge nodes. In *MLSys*, 2019.

[18] Google Cloud. Speech-to-text: Automatic speech recognition. https://cloud.google.com/speech-to-text, 2022.

[19] Matei Zaharia Daniel Kang, Peter Bailis. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. In *VLDB*, 2020.

[20] Nilaksh Das, Monica Sunkara, Dhanush Bekal, Duen Horng Chau, Sravan Bodapati, and Katrin Kirchhoff. Listen, know and spell: Knowledge-infused subword modeling for improving asr performance of oov named entities. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7887–7891, 2022.

[21] John Emmons, Sadjad Fouladi, Ganesh Ananthanarayanan, Shivaram Venkataraman, Silvio Savarese, and Keith Winstein. Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary. In *HotEdgeVideo*, 2019.

[22] Alireza Ghasemieh and Rasha Kashef. 3d object detection for autonomous driving: Methods, models, sensors, data, and challenges. *Transportation Engineering*, 8:100115, 2022.

[23] Tiago Gomes, Diogo Matias, André Campos, Luís Cunha, and Ricardo Roriz. A survey on ground segmentation methods for automotive lidar sensors. *Sensors*, 23(2), 2023.

[24] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, 2016.

[25] Daniel N Hill, Houssam Nassif, Yi Liu, Anand Iyer, and S V N Vishwanathan. An efficient bandit algorithm for realtime multivariate optimization. In *KDD*, 2017.

[26] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[27] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. Spatula: Efficient Cross-camera Video Analytics on Large Camera Networks. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2020.

[28] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *SIGCOMM*, 2018.

[29] Ramesh Johari and John N. Tsitsiklis. Efficiency loss in a network resource allocation game. *Mathematics of Operations Research*, pages 29(3):407–435, 2004.

[30] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, page 13(4):455–492, 1998.

[31] Kai Kai Jüngling and Michael Arens. Local feature based person reidentification in infrared image sequences. In *IEEE AVSS*, 2010.

[32] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing neural network queries over video at scale. In *PVLDB*, 2017.

[33] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ASPLOS*, 2017.

[34] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. Recl: Responsive resource-efficient continuous learning for video analytics. In *NSDI*, 2023.

[35] Yuki Koyama, Issei Sato, Daisuke Sakamoto, and Takeo Igarashi. Sequential line search for efficient visual design optimization by crowds. In *ACM Transactions on Graphics*, 2017.

[36] Harold J. Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, page 86:97–106, 1964.

[37] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12697–12705, 2019.

[38] Wei Li, Rui Zhao, Tong Xiao, and Xiaogang Wang. Deepreid: Deep filter pairing neural network for person re-identification. In *CVPR*, 2014.

[39] Zhuqi Li, Yuanchao Shu, Ganesh Ananthanarayanan, Longfei Shangguan, Kyle Jamieson, and Paramvir Bahl. Spider: A Multi-Hop Millimeter-Wave Network for Live Video Analytics. In *ACM/IEEE Symposium on Edge Computing (SEC)*, 2021.

[40] Giuseppe Lisanti, Iacopo Masi, Andrew D. Bagdanov, and Alberto Del Bimbo. Person re-identification by iterative re-weighted sparse ranking. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[41] Linda Liu, Yi Gu, Aditya Gourav, Ankur Gandhe, Shashank Kalmane, Denis Filimonov, Ariya Rastrow, and Ivan Bulyko. Domain-aware neural language models for speech recognition. In *ICASSP 2021*, 2021.

[42] Daniel Lizotte, Tao Wang, Michael Bowling, and Dale Schuurmans. Automatic gait optimization with gaussian process regression. In *IJCAI*, 2007.

[43] Franz Loewenherz. Video analytics towards vision zero. `https://bellevuewa.gov/sites/default/files/media/pdf_document/video-analytics-presentation-ITE-conference-021317.pdf`, 2017.

[44] Chenglang Lu, Mingyong Liu, and Zongda Wu. Svql: A sql extended query language for video databases. In *IJDTA*, 2015.

[45] Yan Lu, Shiqi Jiang, Ting Cao, and Yuanchao Shu. Turbo: Opportunistic Enhancement for Edge Video Analytics. In *ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2022.

[46] Yan Lu, Zhun Zhong, and Yuanchao Shu. Multi-View Domain Adaptive Object Detection in Surveillance Cameras. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2023.

[47] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *SIGMOD*, 2018.

[48] Ruben Martinez-Cantin, Nando de Freitas, Eric Brochu, Jose Castellanos, and Arnaud Doucet. A bayesian exploration-exploitation approach for optimal online sensing and planning with a visually guided mobile robot. *Autonomous Robots*, pages 27(2):93–103, 2009.

[49] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9), 2020.

[50] Robert C. Merton. *Continuous-Time Finance*. Blackwell, 1990.

[51] Jonas Mockus. *Bayesian Approach to Global Optimization.* Kluwer Academic, 1989.

[52] Shadi Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge-computing. In *ACM SIGMOBILE GetMobile*, 2020.

[53] Pirouz Nourian, Romulo Gonçalves, Sisi Zlatanova, Ken Arroyo Ohori, and Anh Vu Vo. Voxelization algorithms for geospatial applications: Computational methods for voxelating spatial datasets of 3d city models containing 3d surface, curve and point data models. *MethodsX*, 3:69–86, 2016.

[54] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. GEMEL: Model Merging for Memory-Efficient, Real-Time Video Analytics at the Edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[55] Hang Qiu, Pohan Huang, Namo Asavisanu, Xiaochen Liu, Konstantinos Psounis, and Ramesh Govindan. Autocast: Scalable infrastructure-less cooperative perception for distributed collaborative driving. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '22, 2022.

[56] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *CVPR*, 2017.

[57] Colleen Richey, Maria A. Barrios, Zeb Armstrong, Chris Bartels, Horacio Franco, Martin Graciarena, Aaron Lawson, Mahesh Kumar Nandwana, Allen Stauffer, Julien van Hout, Paul Gamble, Jeff Hetherly, Cory Stephenson, and Karl Ni. Voices obscured in complex environmental settings (voices) corpus, 2018.

[58] Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 1979.

[59] Francisco Romero, Johann Hauswald, Aditi Partap, Daniel Kang, Matei Zaharia, and Christos Kozyrakis. Optimizing video analytics with declarative model relationships. *Proc. VLDB Endow.*, 16(3):447–460, nov 2022.

[60] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *SoCC*, 2021.

[61] sBrian T. Ratchford. Cost-benefit models for explaining consumer choice and information seeking behavior. *Management Science*, 28, 1982.

[62] Shuyao Shi, Jiahe Cui, Zhehao Jiang, Zhenyu Yan, Guoliang Xing, Jianwei Niu, and Zhenchao Ouyang. Vips: Real-time perception fusion for infrastructure-assisted autonomous driving. In *MobiCom*, 2021.

[63] Jiang Shiqi, Lin Zhiqi, Li Yuanchun, Shu Yuanchao, and Liu Yunxin. Flexible High-resolution Object Detection on Edge Devices with Tunable Latency. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2021.

[64] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[65] Niranjan Srinivas, Andreas Krause, Sham M. Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *ICML*, 2010.

[66] Voci. Voci: Real-time speech recognition. https://www.vocitec.com/ads/real-time-speech-to-text, 2022.

[67] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures. In *VLDB*, 2023.

[68] Tianwei Yin, Xingyi Zhou, and Philipp Krahenbuhl. Center-based 3d object detection and tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11784–11793, 2021.

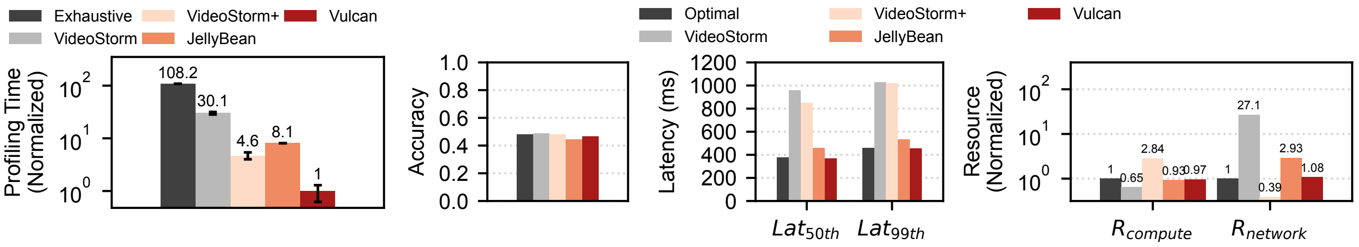
[69] Ji Won Yoon, Beom Jun Woo, and Nam Soo Kim. Hubert-ee: Early exiting hubert for efficient speech recognition, 2022.

[70] 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 *NSDI*, 2017.

[71] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. Emp: Edge-assisted multi-vehicle perception. In *MobiCom*, 2021.

[72] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. Emp: Edge-assisted multi-vehicle perception. In *MobiCom*, 2021.

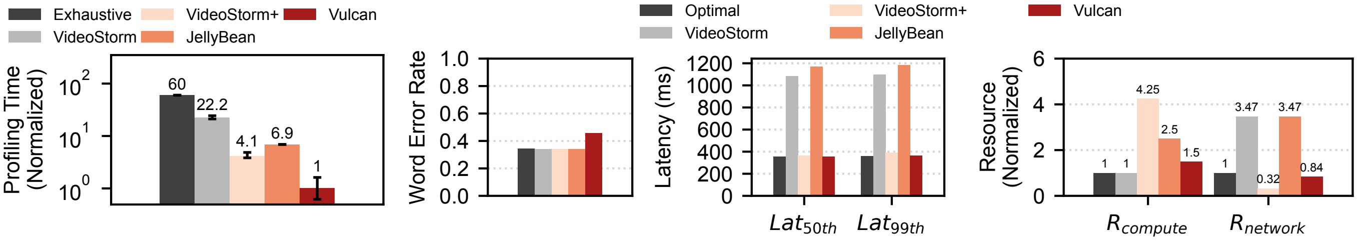
[73] Xinge Zhu, Yuexin Ma, Tai Wang, Yan Xu, Jianping Shi, and Dahua Lin. Ssn: Shape signature networks for multi-class object detection from point clouds. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXV 16*, pages 581–597. Springer, 2020.



(a) Profiling Cost

(b) Performance and Resource Consumption

Figure 18: End-to-end performance when exploring the best placement and query configuration for autonomous driving queries.



(a) Profiling Cost

(b) Performance and Resource Consumption

Figure 19: End-to-end performance when exploring the best placement and query configuration for speech recognition queries.

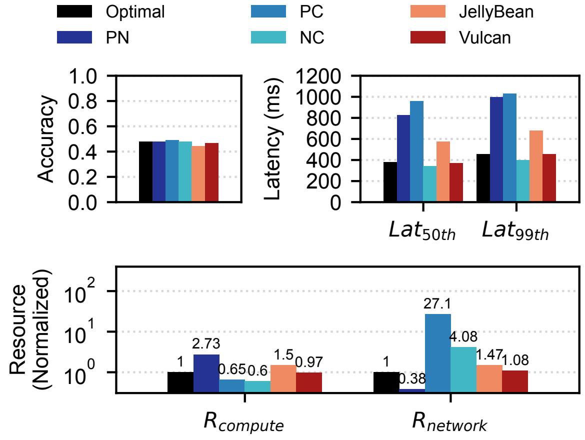


Figure 20: Comparing Vulcan with different placement strategies on serving AD perception queries.

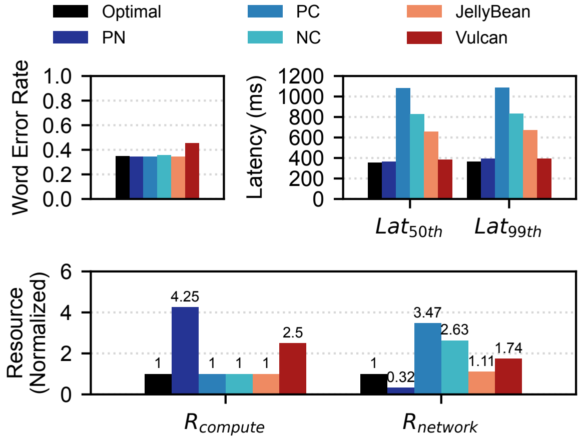


Figure 21: Comparing Vulcan with different placement strategies on serving ASR queries.

A ADDITIONAL EVALUATION RESULTS

A.1 Datasets and Query Configuration

We describe below the details of the datasets and query configuration we used in the paper.

Datasets:

The traffic monitoring queries use videos captured by traffic cameras among different metropolitan areas in Bellevue and Washington D.C. The videos are encoded in MP4 format (1280x720p, 30fps, and 120 seconds long) and randomly sampled from two-hour long videos among 24 days.

For autonomous driving perception queries, we use LiDAR sensor data from nuScenes [16], a large-scale autonomous driving dataset. The dataset features 1000 20-second driving scenes collected over months, in Boston and Singapore encompassing a diverse range of challenging driving situations.

Automatic speech recognition queries use the VOiCES dataset [57], an English speech audio dataset by male and female speakers. The dataset contains 3903 audio files (total 15 hours long) containing different room settings, simulated head movement, and various background noise patterns.

Configuration Knobs. Besides the variation of ML models described in §6.1, our queries in Evaluation configure the following additional configuration knobs.

- Video monitoring queries configure input frame sampling rate (1/2, 1/3, 1/4, 1/5, 1/6) and frame resizing factor (0.6, 0.7, 0.8, 0.9, 1) for frame resolution.
- Autonomous driving queris configure the ground removal factor[3] and voxel size, each with 5 configuration values (0.1, 0.2, 0.3, 0.4, 0.5).
- Speech recognition queires configure audio sampling rate (8k, 10k, 12k, 14k, 16k) and frequency mask width (500,

[3] corresponds to the maximum distance between a ground point and the estimated 2D ground plane.

1000, 2000, 3000, 4000) of the noise reduction module.

A.2 End-to-End Improvement

Figure 18 and Figure 19 shows the comparison between Vulcan and other baselines in the end-to-end performance of autonomous driving and speech recognition queries. The same conclusion can drawn as mentioned in §6.2.

A.3 Selecting Better Placement

Figure 20 and Figure 21 records the performance and resource consumption of autonomous driving and speech recognition queries, where PN, PC, and NC use the optimal query plan achieved by exhaustive search. Similar to §6.4, Vulcan always achieves the best placement choice, outperforming other baselines in query performance and resource consumption.