OptiServe: Cost-Aware, Performance-Driven, and Accuracy-Tuned Serverless Applications with ML Workloads

Arian Boukani

A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree of Master of Science

GRADUATE PROGRAM IN COMPUTER SCIENCE

York University
TORONTO, ONTARIO

January, 2026

Abstract

Serverless computing has emerged as a popular cloud paradigm due to its seamless scalability and cost-efficient, pay-as-you-go pricing model. Its potential to support machine learning (ML) inference workloads,including generative AI tasks, has led to growing adoption of ML functions within serverless applications. A key challenge, however, is selecting suitable ML models that balance execution time, deployment cost, and inference accuracy in latency- and cost-sensitive environments.

In this study, we present a framework for optimizing serverless applications that incorporate ML components through tri-objective optimization. We develop high-fidelity analytical models, augmented with lightweight profiling, to capture the trade-offs among cost, performance, and accuracy across different model choices. These models serve as the foundation for guiding ML model selection and deployment strategies to meet application-specific service-level objectives. We validate our framework through real-world experiments on AWS using real serverless applications. Furthermore, we demonstrate its practicality by performing extensive what-if analyses, exploring a wide range of application scenarios and configurations, in under a minute. Our extensive experiments on real-world applications show that OptiServe recommends memory and ML model configurations that achieve over 95% of the accuracy of ideal configurations in 89.64% of cases, enabling efficient, low-cost deployments while maintaining model accuracy and meeting performance targets.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Dr. Hamzeh Khazaei and Dr. Manar Jammal, for their constant support and guidance throughout my graduate studies. Their advice, encouragement, and insight played a key role in shaping this work and helping me grow as a researcher.

I am also grateful to my friends and colleagues in the Performant and Available Computing Systems (PACS) Lab. Their support, discussions, and collaboration made this journey both productive and enjoyable, and greatly contributed to my academic and personal development.

Finally, I would like to thank my parents, who believed in me even when I doubted myself. Their unconditional support, patience, and love have been the foundation of everything I have achieved, and I am deeply grateful to them.

Table of Contents

List of Tables

List of Figures

Declaration of Authorship

I, Arian Boukani, hereby declare that this thesis titled, "OptiServe: Cost-Aware, Performance-Driven, and Accuracy-Tuned Serverless Applications with ML Workloads", and the work presented herein are solely my own efforts. Parts of this thesis have been submitted as a manuscript to the 46th IEEE International Conference on Distributed Computing Systems (ICDCS), which is currently under review.

In addition to the work presented in this thesis, I have contributed to several research publications during my graduate studies that are related to systems for machine learning training. While these works are not directly included in this thesis, they are closely aligned with its broader theme of performance modelling and optimization for machine learning systems.

1. A. Pourali, A. Boukani, and H. Khazaei, "CAPE: Generalized convergence prediction across architectures without full training," *Transactions on Machine Learning Research*, 2025.

2. A. Pourali, A. Boukani, and H. Khazaei, "PreNeT: Leveraging computational features to predict deep neural network training time," in *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2025.

3. A. Pourali, A. Boukani, and H. Khazaei, "Chronos: A unified framework for predicting training time and convergence in deep learning," in *Proceedings of the Machine Learning and Systems Conference (MLSys)*, vol. 8, 2026 (submitted).

4. M. Naderi, A. Boukani, A. Pourali, and H. Khazaei, "AdapTrain: Adaptive model partitioning for efficient independent subnet training on heterogeneous and dynamic cloud infrastructures," *IEEE Transactions on Cloud Computing* (submitted).

Chapter 1

Introduction

1.1 Overview

Cloud computing has gained significant interest in recent years from a variety of users due to its high scalability, availability, low infrastructure management requirements, and cost-effectiveness. Major cloud service providers, such as Amazon Web Services (AWS) [1], Google Cloud Platform (GCP) [2], and Microsoft Azure [3], offer a wide range of services. Serverless computing, a recent paradigm, has attracted considerable attention from cloud users because it eliminates the need for provisioning and deploying infrastructure, offering a pay-as-you-go cost model. Most cloud providers now offer serverless computing in the form of various services, including Function-as-a-Service (e.g., AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions), serverless containers (e.g., AWS Fargate and Google Cloud Run), database services, and other specialized offerings. Recent workload analyses [4] indicate that 50% to 72% of organizations using major cloud providers have adopted serverless computing.

As serverless computing has gained traction in recent years, researchers have been working to overcome its key limitations. One of the biggest challenges serverless developers face is rightsizing, optimizing resource allocation (such as memory and CPU) to strike the right balance between performance and cost of the function. This issue becomes more significant when serverless functions are grouped and deployed as serverless applications. However, this problem contradicts the core philosophy of serverless computing, which aims to abstract away infrastructure management for users. To tackle this challenge, developers often resort to trial and error, manually testing different configurations and inputs to find the right balance. Some rely on automated tuning tools, but these solutions can be expensive and do not always scale well [5]. In addition, they often lack dynamic optimization for different latency constraints [6]. These constraints are defined by the Service Level Objectives (SLOs); failing to meet them can lead to degraded service quality or even financial losses, as users may not be charged for queries that miss their deadlines [7].

With the rise of AI (Artificial Intelligence), particularly ML (Machine Learning), an increasing number of applications now incorporate ML tasks into their workflows. Serverless computing has become a popular choice for ML training [8] and inference [9, 10, 11, 12, 13, 14, 15]. Its high scalability, ease of deployment, and potential cost savings are key drivers for adopting serverless computing in ML inference [16]. Additionally, the stateless nature of ML inference makes serverless computing an excellent fit for this purpose. While ML training happens offline and can take hours or even days, inference needs to run in real-time, responding to dynamic queries under strict SLOs. Because of this, ML model serving applications

must balance meeting SLO targets with minimizing cloud costs, ensuring efficient resource allocation while maintaining high performance. When using ML models for inference tasks, several factors beyond memory and CPU size can impact response time and cost. These include the choice of ML model [6, 17, 18, 19, 20], batching and scheduling strategies [9, 11, 18, 21], and model partitioning [12, 22].

A group of interrelated serverless functions form a serverless application. Nearly 46% of applications running on Microsoft Azure consist of more than one function [23], reflecting a growing trend toward composing applications from multiple serverless functions. To support such workflows, cloud providers offer orchestration services such as Amazon Step Functions, Google Cloud Workflows, and Azure Durable Functions.

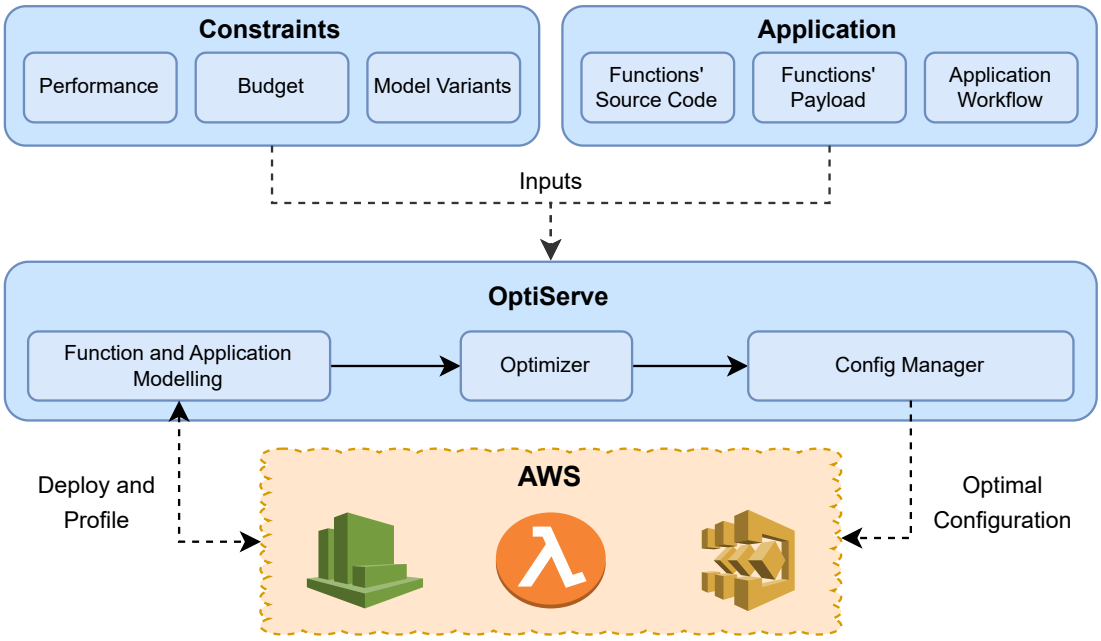

FIGURE 1.1: A high-level overview of OptiServe

In this paper, we introduce **OptiServe**, a framework designed to model and optimize serverless applications in terms of performance, cost, and the proper ML

model selection (i.e., models with different parameter sizes and accuracy). Such applications tend to have complex structures and ML inference tasks. OptiServe aims to minimize both cost and execution time while ensuring the required performance and model accuracy. It dynamically adjusts the application configurations in real time to meet service-level objectives (SLOs) and maintain the desired accuracy for ML models. Figure 1.1 depicts the high-level overview of OptiServe.

This framework takes two main inputs: the optimization constraints and the application's logic, including its source code and payloads. OptiServe first deploys the application on serverless cloud (e.g., AWS Step Functions and Lambda), then profiles its execution for the modelling phase. It collects logs that are used to build performance and cost models, which are then fed into the optimizer. Using heuristics, the optimizer identifies the optimal configuration that satisfies the given constraints. Finally, the Config Manager applies the optimal settings to reconfigure the application.

1.2 Motivation

Despite the growing adoption of serverless platforms for deploying machine learning applications, achieving cost-effective and performance-aware inference in this setting remains challenging. The difficulty stems not from a single limitation, but from the interaction between cloud billing models, opaque resource management, workflow-level dependencies, and accuracy–performance trade-offs inherent to modern ML models. The following points summarize the key factors that motivate the need for a workflow-level, accuracy-aware optimization framework.

- **Inference is now a recurring cost center.** ML inference has become a dominant contributor to operational spending for AI-driven services. As models grow in size and are deployed continuously over long periods, the cumulative cost of serving requests can exceed the one-time training cost by a significant margin. As a result, optimizing inference cost is no longer optional, but a practical necessity for sustainable deployment.

- **Rightsizing is hard in black-box FaaS environments.** Rightsizing a serverless function involves selecting a resource configuration that meets performance targets at minimal cost. In FaaS platforms, this task is complicated by limited visibility into execution placement, resource contention, and scaling behavior. Configuration changes often require redeployment, and naive trial-and-error tuning is both time-consuming and costly.

- **Workflows make local decisions globally suboptimal.** Serverless applications are increasingly structured as workflows rather than isolated functions. In such settings, end-to-end latency, cost, and quality depend on the interaction between stages. Critical-path functions dominate user-perceived latency, branching affects the expected number of invocations, and suboptimal choices at one stage can propagate downstream. Consequently, optimizing functions in isolation frequently fails to satisfy application-level constraints.

- **Accuracy is an explicit objective, not a side effect.** Many existing tuning approaches implicitly assume fixed inference accuracy and focus solely on cost or latency. In practice, ML workflows often provide multiple

model variants with different accuracy–latency–cost trade-offs. This introduces a three-way optimization problem in which accuracy must be treated as a first-class objective, especially for applications with explicit service-level objectives (SLOs) and minimum quality requirements.

1.3 Thesis Contributions

This thesis makes the following contributions:

- **Tri-objective formulation for serverless ML workflows.** We formalize the problem of configuring a serverless workflow with ML stages as a joint optimization over *end-to-end latency*, *end-to-end cost*, and *end-to-end accuracy*, under application-level constraints.

- **Analytical modelling for function and workflow behavior.** We develop high-fidelity analytical models (augmented with lightweight profiling) that capture the latency and cost behavior of serverless function variants and propagate these effects through workflow structure.

- **Accuracy-aware workflow interface.** We introduce an end-to-end accuracy abstraction for workflows with multiple ML stages by supporting a user-defined aggregation function that reflects application semantics while remaining compatible with optimization.

- **Lightweight optimization algorithms for practical configuration search.** We design deterministic heuristics that enable efficient search across memory configurations and ML model variants, making the approach practical

for real deployments and enabling fast what-if analysis.

- **Experimental validation on real serverless workflows.** We evaluate OptiServe on AWS using real applications and demonstrate that it can recommend configurations that satisfy constraints while preserving a high fraction of the accuracy of ideal (oracle) configurations.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 provides the necessary background and foundational concepts that motivate our work. Chapter 3 reviews related research and positions our contributions within the existing literature. Chapters 4 and 5 present the design and implementation of **OptiServe** in detail. Chapter 6 evaluates the proposed system and discusses the experimental results. Finally, Chapter 7 concludes the thesis by summarizing the main contributions and outlining directions for future research.

Chapter 2

Background and Related Work

Serverless platforms offer pay-as-you-go billing, automatic scaling, and a fully managed execution environment, making them particularly well-suited for latency-sensitive and burstable machine learning (ML) inference workloads. By elastically provisioning CPU and memory resources on demand, serverless platforms eliminate the need for idle capacity headroom, enabling applications to absorb rapid traffic spikes without manual intervention. Recent studies demonstrate that, when combined with locality-aware optimizations, serverless architectures can achieve millisecond-scale cold start latencies for inference tasks, further underscoring their suitability for interactive AI services [13].

2.1 Limitations of Existing Works

ML inference has become the dominant contributor to the operational budgets of AI-driven businesses. As models scale to billions of parameters, the cost of serving even modest query volumes can exceed the one-time training expense by 10 to 20× over the model's lifetime [24]. With inference accounting for up to 80% of total AI

spend in large deployments, cost-effective serving has emerged as a critical priority for both cloud providers and enterprise SRE teams. In the absence of automated tuning, inference pipelines tend to either overprovision, wasting budget on idle capacity, or underprovision, leading to latency violations during peak load, both unacceptable in production.

Rightsizing serverless functions, selecting the appropriate memory allocation, CPU share, and ML model variant to meet both latency and accuracy SLOs, is especially challenging in black box FaaS environments. Users have limited visibility into VM placement, resource contention, and provider scaling policies, and modifying function configurations often requires full redeployment. Profiling-based tuners, such as StepConf, have shown that dynamic configuration can reduce costs by up to 40.9% compared to static settings [25]. While strategies like on-demand container loading can mitigate cold start delays, they do not eliminate the need for automated rightsizing to meet both performance and cost objectives [26].

Many modern frameworks treat function performance as a black box, relying on trained runtime predictors or on device profiling. Iyer and others [27] embed a lightweight profiler that measures latency and memory usage for each operation, constructs Pareto frontier profiles, and dynamically reconfigures inference backends per model without explicitly learning performance curves. INFaaS [6], in contrast, trains statistical models on historical executions to support model-less inference serving. OptiServe takes a different approach by encoding exact cost and performance formulas for each function variant offline and applying lightweight, deterministic heuristics at runtime, thereby eliminating the need for profiling altogether.

Most existing work addresses only two objectives, such as latency versus cost or accuracy versus latency, but not the full three-way trade-off involved in selecting model variants, configuring resources, and scheduling execution [28]. Parrotfish [5], for example, applies parametric regression to tune memory allocations in serverless functions, optimizing for cost and latency. However, it does not explicitly incorporate accuracy considerations, unlike some SLA-driven ML frameworks.

2.2 Related Work

This section reviews prior research relevant to the modelling, analysis, and optimization of serverless systems and machine learning workloads. We organize the discussion along three complementary dimensions: function-level modelling techniques for predicting performance and cost in serverless environments, application-level modelling approaches that capture workflow structure and dependencies, and optimization frameworks that tune system configurations under latency, cost, and resource constraints. This organization highlights both the progress made in each area and the limitations that motivate the workflow-level, accuracy-aware optimization framework proposed in this thesis.

2.2.1 Function Modelling in Serverless Systems

Accurate modelling is essential for optimizing resource provisioning and execution in serverless environments. Prior work explores analytical models [25, 28, 29], regression techniques, and ML-based predictors to estimate latency, memory usage, and cold start delays. These approaches capture performance-cost tradeoffs under varying configurations, often incorporating SLO-awareness, as in ChunkFunc

[30] and Cho and others [31], which use lightweight profiling to guide resource selection.

Machine learning based predictors expand the design space by learning mappings from inputs, configurations, and environment states to runtime behaviour. SimNet [32] uses deep networks to predict instruction-level latency for architecture simulation, while Driple [33] applies GNNs to forecast resource usage in distributed training. INFaaS [6] uses online profiling to guide variant selection, FuncPipe [34] optimizes model partitioning and resource configuration for serverless distributed training, and Iyer and others [27] combine runtime metadata with graph-based learning to dynamically select inference backends.

Profiling is widely used to optimize training and inference in serverless ML. Optimus [15] mitigates cold starts by enabling inter-function model transformation across warm functions. USHER [35] builds interference- and startup-aware configurations via fine-grained runtime measurements, while Fireworks [36] reduces cold start latency using post-JIT VM snapshots, avoiding complex queuing or RL techniques.

For large-scale inference, recent work emphasizes throughput-aware modelling. PetS [37] proposes unified scheduling for parameter-efficient transformers; *Prompt Cache* [38] reduces redundant LLM invocations via retrieval-based caching; Atom [39] boosts throughput through fused low-bit quantization strategies. Beyond profiling, systems like Faasm [40] use FaaS-specific abstractions to enable fast scheduling and in-memory state sharing without profiling. Featherlight [41] reduces latency and memory in serverless WASM environments using thread-level

execution, zero-copy memory sharing, and adaptive scheduling.

Batch and scheduling-aware modelling has also gained traction. BATCH [9] and Orca [42] model how batching affects latency and throughput for optimized inference. TorchSparse [43] improves sparse convolution efficiency through compute and memory optimizations, without modelling batching. PipeCo [29] addresses cold-start latency via pipelined model loading. TOP [44] uses operator-level task models to support asynchronous execution under memory constraints.

2.2.2 Application Modelling

In serverless environments, applications are often represented as directed acyclic graphs (DAGs), which make explicit the invocation order, data dependencies, and potential resource contention between functions. ORION [45] sizes, bundles, and prewarms DAG nodes to meet latency objectives, while WiseFuse [46] applies node fusion and edge reordering to reduce critical path latency. FaaSFlow [47] enhances serverless DAG execution by supporting complex control logic and enabling fine-grained scheduling through DAG parsing, workflow partitioning, and worker-side execution strategies.

Some frameworks introduce pipeline-style data chaining within DAGs. FuncPipe [34] applies pipeline parallelism via model partitioning and scatter-reduce pipelining for cost-efficient deep learning training on serverless platforms. PipeCo [29] mitigates cold start latency in inference by vertically partitioning DL models and prewarming container slices in overlapping pipelines. SplitRPC [48] decomposes RPC requests into control (metadata) and data (payload) paths, allowing CPU orchestration and direct GPU data transfer to reduce inference latency.

Jolteon [49] accelerates serverless workflows by predicting downstream runtimes and proactively allocating resources, optimizing for latency and cost via stochastic modelling. StarShip [50] tackles I/O bottlenecks in scientific workflows by co-optimizing storage and compute configurations using Levenberg-Marquardt optimization.

2.2.3 Optimization

Serverless frameworks tune configuration knobs such as model selection [6, 32, 34], memory allocation [25, 28, 51], input-aware tuning [30], and hardware placement [31, 49]. Solutions span analytical, learned, and heuristic methods. μ-Serve [52] applies GPU frequency-aware DVFS and operator partitioning under SLO constraints. Lin and others [28] propose a greedy heuristic (PRCP) with four strategies based on analytical performance and cost models to find optimal memory configurations for serverless applications under performance or budget constraints. UnFaaSener [53] adaptively offloads serverless functions to user-provided hosts using an asynchronous scheduler and cost-latency aware decisions without modifying existing platforms. CATO [54] uses multi-objective Bayesian optimization to jointly tune predictive performance and system costs for ML-based traffic analysis pipelines.

Execution-level systems like Tetris [51], HashCache [55], and Parrotfish [5] reduce memory, latency, and redundant computations through deduplication, caching, and regression-guided tuning. Batch and pipeline optimizers, e.g., BatOpt [56], Liu and others [57], Orca [42], PUZZLE [58], and CacheBlend [59], exploit accelerator characteristics and KV cache reuse to reduce inference latency. MITOSIS

[60] enables fast remote container forking with RDMA for low-latency cold starts. Brooker and others [26] integrate block-level demand loading and deduplication in AWS Lambda to improve cold start scalability. UnFaaSener [53] incorporates network and compute costs in offloading decisions for latency and cost-aware hybrid execution. End-to-end systems like Willump [61], DeepRest [62], and Iyer and others [27] leverage neural cost models for performance optimization.

Chapter 3

Function and Application Modelling

To achieve the optimization objectives, we first need to model the real serverless applications with a proper trade-off between fidelity and tractability. With that modelling in place, we can proceed to the optimization phase, as the cost in the serverless paradigm is proportional to the performance. We begin by modelling serverless functions, which are the smallest unit of granularity (i.e., building block) in this context. Then, we proceed to model a group of interrelated functions, i.e., the application with any possible workflow. Table 3.1 summarizes the notations used throughout this paper and their corresponding explanations.

3.1 Function Modelling

To enable cost-performance optimization, we begin by modelling the behaviour of individual serverless functions. Building upon the approach introduced in Parrotfish [5], we adopt a parametric regression-based model that relates a function's

TABLE 3.1: Definition of the notations used in the modelling and algorithms

Notation	Description
f	A serverless function.
G_s	Serverless workflow as a graph.
G_{dl}	Represents a de-looped graph.
sp	A simple path in the workflow.
$e = (f_i, f_j)$	A directed edge from f_i to f_j.
$\mathcal{P}$	Power set
$ETM(f_i, m)$	The execution time of general function f_i with memory configuration m.
$C(f_i, m)$	The cost of general function f_i with memory configuration m.
$ETM(f_i, m, a)$	The execution time of ML inference function f_i with memory configuration m and normalized ML model accuracy a used in the function.
$C(f_i, m, a)$	The cost of ML inference function f_i with memory configuration m and normalized ML model accuracy a used in the function.
$ET(f_i)$	Execution time of function f_i.
$M(f_i)$	Memory usage of function f_i.
$A(f_i)$	Normalized accuracy of the ML model used in function f_i.
$NI(f_i)$	Number of invocations of function f_i.
$P(f_i, f_j)$	Probability of transition from f_i to f_j.
$EET(G_s)$	End-to-end execution time of serverless workflow G_s.
$TP(s)$	The transition probability of a simple path.
$PET(s)$	Execution time of a simple path s.
$EI(G_c)$	Expected number of cycle iterations in the cycle structure G_c
$EI(G_l)$	Expected number of loop iterations in the self-loop structure G_l
PGS	Price per GB/s execution of functions.
PPI	Price per invocation of functions.
$EAS(G_s)$	End-to-end accuracy score of the serverless application G_s.
BCR	Benefit-cost ratio.

execution time to its allocated memory. They evaluated several functional forms for modelling serverless performance, including linear, logarithmic, and exponential models, and found that an exponential decay function provided the best fit across a diverse set of real-world serverless functions. Specifically, the model is

defined as

$$\mathrm{ETM}(f_i, m) = \alpha + \beta \cdot e^{-\gamma m} \tag{3.1}$$

where m is the allocated memory, and α, β, and γ are parameters learned from empirical runtime data. This formulation captures the diminishing returns of increasing memory on execution time. Given the performance model, the corresponding cost can be directly computed using the serverless pricing model as:

$$C(f_i, m) = NI(f_i)\Big(\Big\lceil \frac{ETM(f_i, m)}{100} \Big\rceil \cdot M(f_i) \cdot PGS + PPI\Big) \tag{3.2}$$

in which, $NI(f_i)$ denotes the number of invocations of function f_i, $M(f_i)$ is the allocated memory to f_i in GBs, PGS is the price per GB-second, and PPI is the price charged per invocation. Note that the execution time of f_i is rounded up to the nearest 100 milliseconds, in accordance with typical serverless billing models.

To build the performance model, the serverless function is first profiled under a small set of seed memory configurations (e.g., 128 MB, 1024 MB, 3008 MB). From this initial data, a belief function is constructed to estimate the likelihood that each unexplored memory configuration will yield the lowest cost. This belief function is approximated as a normal distribution centred around the currently known cost-optimal configuration, with a fixed standard deviation of 200 MB.

At each iteration, the algorithm selects the memory configuration with the lowest expected cost under uncertainty and proceeds to evaluate its execution

time

$$\min_{m} \ \text{Objective}(m) = \text{Cost}(m) \times \text{Belief}(m)\,, \tag{3.3}$$

where Cost(m) is the predicted execution cost at memory configuration m, and Belief(m) represents the probability (estimated via the belief distribution) that m is the cost-optimal configuration.

This process continues until the belief function concentrates tightly around a single configuration, indicating that further sampling is unlikely to change the decision. At that point, the search terminates, and the current model is used for predicting cost and execution time across the entire memory range. This sample-efficient approach avoids exhaustive profiling and enables rapid yet reliable modelling, which is crucial for downstream application-level optimization.

As discussed in detail in [63], performance variance is a key concern in serverless computing due to the platform's black-box deployment and execution model. To mitigate the impact of such variance on our function performance modelling, we sample each memory configuration three times and use the average warm-start execution time as the representative value. The choice of three samples is empirical: fewer samples tend to reduce modelling accuracy, while additional samples yield diminishing returns relative to their cost.

If the three samples show high variability, defined as a coefficient of variation (CoV) greater than 0.05, we continue sampling until the CoV falls below this threshold or remains consistently high. We also filter out anomalous cases where

execution time increases with higher memory allocations, which contradicts expected performance trends. In such cases, we allow the function to return to a cold state, redeploy it, and repeat the profiling process to ensure more stable measurements.

We apply this modelling approach to each function in the application. For machine learning inference tasks, we model the performance and cost of all available model variants associated with each task. This yields models: $\mathrm{ETM}(f_i, m, a)$ for execution time and $\mathrm{C}(f_i, m, a)$ for cost, where m denotes the memory configuration and a represents the normalized accuracy level of the ML model used for inference. With these function-level models established, we proceed to the next stage, which involves performance and cost modelling at the application level. Figure 3.1 presents the performance and cost models for several real-world functions, along with their modelling accuracy. Each function was invoked 100 times per memory configuration, with the resulting durations shown as translucent scatter points. The corresponding execution time and cost models are plotted as dotted lines. Additionally, the $MAPE$ values, quantifying the prediction error between the models and observed medians, are annotated near each curve.

3.2 Application Modelling

After modelling each individual function in the serverless application, we can establish a model for the overall performance and cost of the application as a whole. Extending on the work by [28], we define a serverless application as a weighted directed graph:

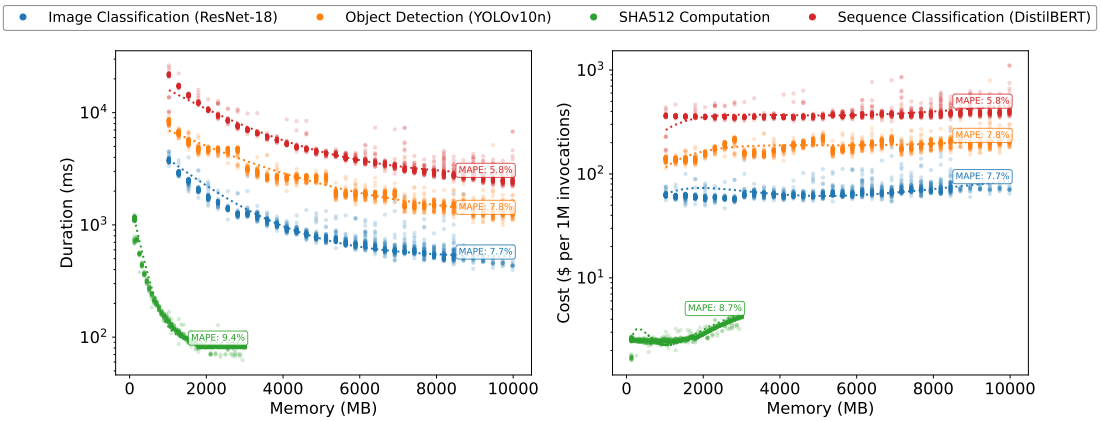

FIGURE 3.1: Performance and cost modelling of some functions

$$\mathrm{G}_s = (V, E, P, M, A, ET, ETTP, NI, C) \tag{3.4}$$

- In which $\mathbf{V}$ is the finite set of functions within the workflow as vertices;
- $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a finite set of directed edges. The directed edge from $f_i \in V$ to $f_j \in V$, specified as $e = (f_i, f_j)$, stands for the interaction between f_i and f_j defined by the user;
- $\mathbf{P} : \mathbf{V} \times \mathbf{V} \to [0, 1]$ is a transition probability function. $P(f_i, f_j)$ stands for the probability of invoking f_j after f_i is executed;
- $\mathbf{M} : \mathbf{V} \to \mathbb{N}$ maps each function within the workflow to its memory size. $M(f_i)$ is the memory size of function f_i;
- $\mathbf{A} : \mathbf{V} \to [0, 1]$ maps each ML inference function in the workflow to the normalized accuracy of the ML model used for inference in that function. $A(f_i)$ is the normalized accuracy value of ML function f_i;

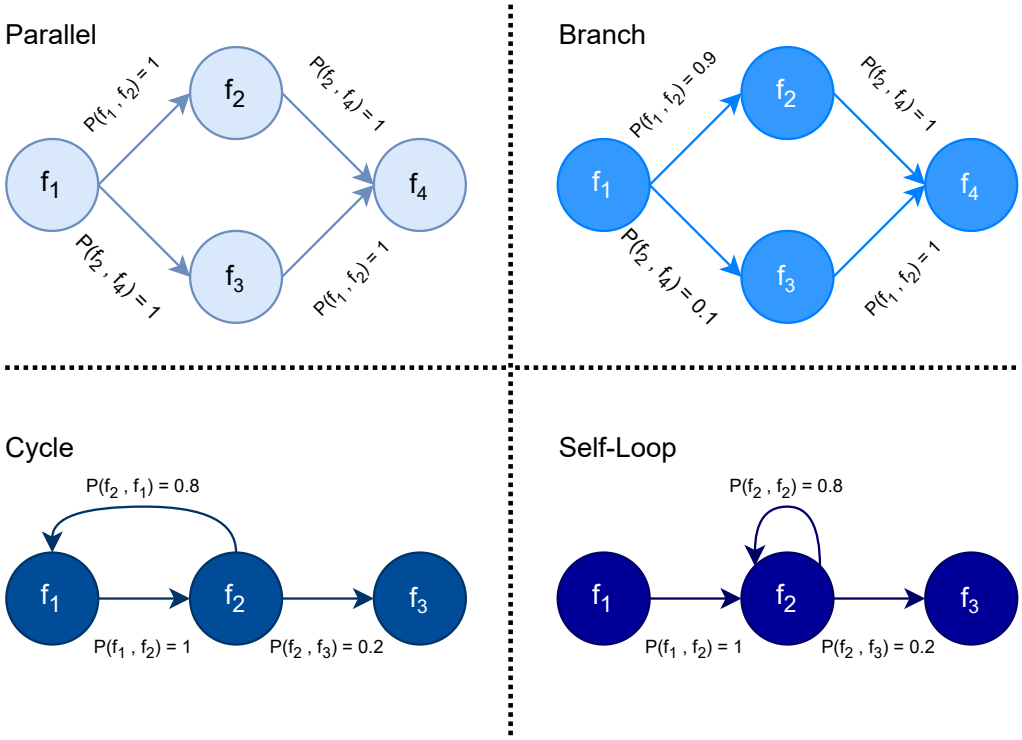

FIGURE 3.2: All the possible structures within a workflow

- **ET** : $\mathbf{V} \rightarrow [0, +\infty)$ stands for the execution time of a function. For instance, $ET(f_i)$ is the execution time of f_i with the current M and A configurations;
- **ETTP** : $\mathbf{V} \rightarrow \mathcal{P}([0, +\infty) \times [0, 1])$ is a function holding the execution time and the corresponding possibility of executing that node;
- **NI** : $\mathbf{V} \rightarrow [0, +\infty)$ is a function representing the average number of invocations of each function within the application, per each invocation of the serverless workflow;
- **C** : $\mathbf{V} \rightarrow [0, +\infty)$ is a cost function. $C(f_i)$ is the cost per each invocation of function f_i.

A serverless workflow typically begins with a trigger function and proceeds through a sequence of invocations. We represent this by adding a start node f_{start} and an end node f_{end} to the set V, denoting the entry and exit points of the workflow. In addition to defining the serverless workflow, we introduce the following notations used throughout our modelling and optimization process:

- A simple path $s = f_1 e_1 f_2 e_2 \dots e_{n-1} f_n$ has a transition probability defined as:

$$TP(s) = \prod_{i=1}^{n-1} P(f_i, f_{i+1}) \tag{3.5}$$

- The total execution time of a simple path s, denoted as $PET(s)$, is the sum of the execution times of all functions along the path:

$$PET(s) = \sum_{i=1}^{n} ET(f_i) \tag{3.6}$$

- The set of all simple paths from function f_i to f_j, including both endpoints, is denoted by $ASP(f_i, f_j)$.

With the serverless workflow defined, the next step is to build performance and cost models, denoted as G_{perf} and G_{cost}. To understand the modelling process, it is essential first to analyze the structural patterns that can appear within a workflow graph. As illustrated in Figure 3.2, serverless workflows commonly exhibit four fundamental structures: parallelism, branching, cycles, and self-loops.

3.2.1 Application Performance Modelling

We propose a performance model to estimate the end-to-end execution time of a serverless application. To do this, we transform the original serverless workflow G_s from Equation 3.4 into a probabilistic directed acyclic graph (DAG), denoted as G_{ET}. The execution time model is defined as follows:

$$\mathrm{G}_{perf} = (V, E, P, M, A, ET, ETTP) \tag{3.7}$$

- In which G_{perf} is a DAG without any loops and cycles;
- $\sum_{s \in ASP(f_{start}, f_{end})} TP(s) = 1$, this ensures that the transition probabilities of all simple paths from the start node to the end node in G_{perf} add up to 1.
- $\mathbf{P} : \mathbf{V} \times \mathbf{V} \rightarrow [0, 1]$ is a transition probability function. $P(f_i, f_j)$ stands for the probability of invoking f_j after f_i is executed.

Having these conditions, the end-to-end execution time of the serverless workflow G_s, denoted as $EET(G_s)$, is calculated as follows:

$$EET(G_s) = \sum_{s \in ASP(f_{start}, f_{end})} TP(s) \cdot PET(s) \tag{3.8}$$

where $ASP(f_{start}, f_{end})$ denotes the set of all simple paths between the start and end nodes in G_{perf}, which is the probabilistic DAG derived from the original workflow G_s using our execution time model. To construct this probabilistic DAG, the

model eliminates cycles and self-loops and prunes parallel paths. The process involves handling the four fundamental workflow structures illustrated in Figure 3.2, which we describe in the following subsections.

Branches. In a branch structure, the workflow follows only one of the available paths based on the condition that holds at runtime. Each condition is associated with a probability, defined by the transition probability function. To simplify such a structure, we introduce a temporary node, f_B, that replaces the entire branch. The response time of f_B is set to the weighted average of the response times of all branch paths. Importantly, we also preserve the transition probabilities of each path, as they are needed later in the $ETTP$ computation.

Parallels. In a parallel structure between f_i and f_j, the workflow executes all parallel paths following f_i, and proceeds to f_j only after all paths complete. Based on this behaviour, we replace the parallel block with a single node f_P, whose execution time is set to the longest execution time among the parallel paths. If any of the paths contain f_B nodes, we also update the $ETTP$ list accordingly.

Cycles. To process cycles, we replace each cycle structure G_c (starting at f_i and ending at f_j) with a single node. The execution time of this node accounts for the expected number of times the cycle is executed. We define this expected iteration count as $EI(G_c)$, which represents the average number of times the workflow re-enters the cycle after reaching f_j. It can be computed as:

$$EI(G_c) = \sum_{n=1}^{\infty} \Big[1 - P(f_i, f_j)\Big] \cdot \Big[P(f_i, f_j)\Big]^{n-1} \cdot (n-1)$$
$$= \frac{P(f_i, f_j)}{1 - P(f_i, f_j)} \tag{3.9}$$

The expected execution time of the cycle is then calculated by multiplying this average iteration count by the time required for one cycle iteration. The total execution time of the node that replaces the cycle is the sum of the execution time of f_i and this expected value.

Self-Loops. Similar to the approach used for processing cycles, we handle a self-loop structure G_l consisting of only f_i by estimating the expected number of loop iterations. This is given by:

$$EI(G_l) = \frac{P(f_i, f_i)}{1 - P(f_i, f_i)} \tag{3.10}$$

The expected execution time of the self-loop is then calculated by summing the execution time of f_i, denoted as $ET(f_i)$, and the product of the expected number of iterations and $ET(f_i)$. This gives:

$$ET(G_l) = [1 + EI(G_l)] \cdot ET(f_i)$$

This transformation allows us to replace the self-loop structure with a single node that captures its expected execution cost. Figure Figure 3.3 illustrates an

example of how an initial serverless workflow is transformed into a probabilistic DAG using the processing techniques described earlier. The numbers on the edges represent transition probabilities between functions. For each f_B node, $ETTP$ is shown as a set of (probability, execution time) tuples. After converting the workflow into a DAG, we compute the average execution time by aggregating the execution times of all nodes, weighted by their $ETTP$ values.

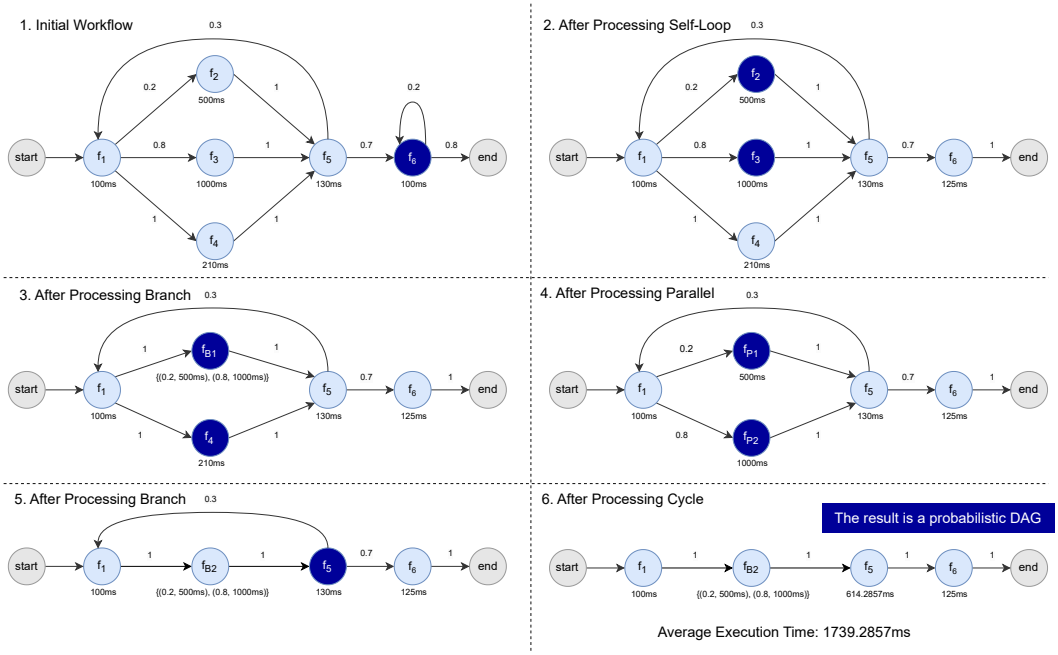

FIGURE 3.3: An illustration of the performance modelling process for a serverless workflow

3.2.2 Application Cost Modelling

We define the following cost model to estimate the average cost of a serverless workflow. Under the standard GB-second pricing model, the cost of each function depends on three main factors: the allocated memory size, the rounded-up execution time (which varies based on both the memory size and the ML model

used for inference), and the number of times the function is invoked. Taking these factors into account, we express the cost model as follows:

$$G_{cost} = (V, E, P, M, A, ET, NI, C) \tag{3.11}$$

The cost modelling process begins by eliminating cycles and self-loops, as outlined in subsection 3.2.1. Once the workflow has been de-looped, we account for the effects of parallel and branching structures. Specifically, the number of invocations NI for each node is updated based on the sum of transition probabilities across all simple paths from the start node to that node. After updating the invocation counts, we compute the cost of each function, both general-purpose and ML inference functions, using Equation 3.2. The total workflow cost is then obtained by summing the costs of all remaining functions.

Chapter 4

Tri-Objective Optimization

In this chapter, we outline three key challenges commonly encountered in serverless machine learning (ML) applications. Currently, most commercial serverless platforms primarily allow developers to configure the allocated memory for functions. Some platforms have also started to expose CPU core customization, but in most cases, the number of CPU cores scales proportionally with the allocated memory size.

Alongside memory tuning, we introduce model switching as a second optimization knob. In ML-centric workflows, the ability to switch between different model variants (e.g., lightweight vs. high-accuracy models) offers a practical way to navigate the trade-offs between cost, latency, and accuracy. In particular, this mechanism is crucial for satisfying user-defined accuracy constraints, which are often critical in real-world applications.

Figure 4.1 illustrates how different combinations of memory sizes and ML model

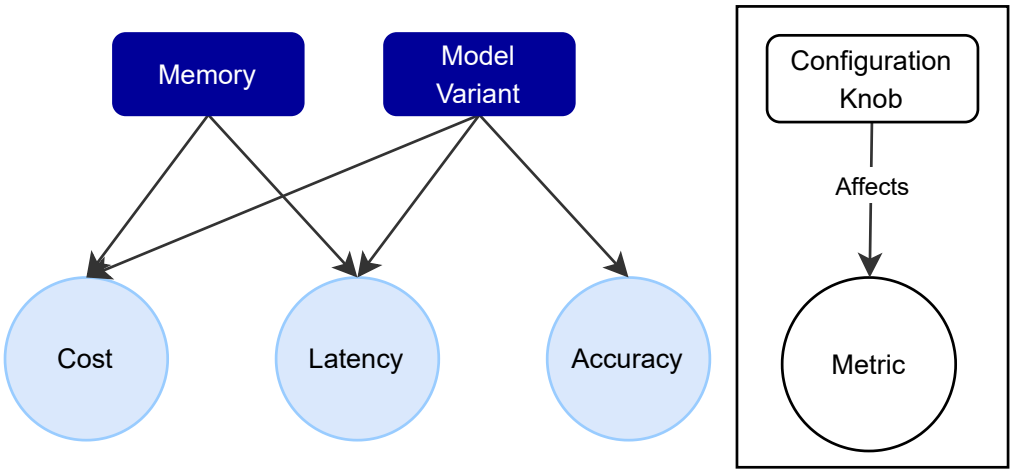

FIGURE 4.1: The effects of changing configuration knobs

variants impact system metrics, including latency, cost, and accuracy. For instance, changing memory without changing the inference model used in a function affects cost and latency.

By jointly tuning memory and model configurations, serverless applications can better meet diverse optimization goals under constrained budgets or strict service-level objectives (SLOs).

4.1 End-to-End Accuracy Definition

Before presenting our methodology, we define *end-to-end accuracy* for a serverless application. A workflow may include multiple ML inference functions, each using a different model variant. We introduce a metric that captures the combined effect of these models on overall inference quality, where higher values indicate the use of larger, more accurate models.

Prior work has estimated application-level accuracy using the product of individual model accuracies along the pipeline [18]. However, there is no widely accepted definition of application-level accuracy in practice. Unlike latency or cost, accuracy cannot be uniformly defined across functions, as models may target different tasks or datasets. In addition, accuracy errors are often correlated, with errors propagating and compounding as data flows through the application.

To handle application-specific accuracy needs, users provide a custom formula that defines how the accuracies of individual ML functions should be combined. This expression may include any mathematical operations and allows users to emphasize functions that play a more critical role in their workflow. We use this formula both to verify accuracy constraints and to guide accuracy-aware optimization under budget and latency limits. For example, in a workflow G_s with four ML inference functions f_1–f_4, a user might define the end-to-end accuracy as:

$$EAS(G_s) = 2 \cdot A(f_1) + A(f_2) + (A(f_3) \cdot A(f_4)) \tag{4.1}$$

In this formulation, the user assigns higher importance to the accuracy of function f_1, reflecting its critical role. The accuracies of f_3 and f_4 are multiplied to capture the importance of their joint correctness.

4.2 Problem Statement

Building upon the execution time and cost models introduced in section 3.1, we assume that for each function f_i in the workflow, the execution time under a

specific memory size m and model accuracy a is captured by $ETM(f_i, m, a)$. Using this, we define two mapping functions:

$$\begin{aligned} \text{MLIST} &: V \to \mathcal{P}(\mathbb{N}) \\ \text{ALIST} &: V \to \mathcal{P}([0,1]) \end{aligned} \tag{4.2}$$

Here, MLIST maps each function to its set of viable memory configurations, and ALIST maps each function to its corresponding set of normalized accuracy options (i.e., model variants). Considering a serverless workflow G_s composed of n functions, where $V = \{f_1, f_2, \ldots, f_n\}$. We define:

- π as the memory configuration of the workflow:

$$\pi \in \text{MLIST}(f_1) \times \text{MLIST}(f_2) \times \cdots \times \text{MLIST}(f_n)$$

- δ as the ML model configuration of the workflow:

$$\delta \in \text{ALIST}(f_1) \times \text{ALIST}(f_2) \times \cdots \times \text{ALIST}(f_n)$$

For a given function f, $\pi(f)$ denotes the memory allocated to f, while $\delta(f)$ represents the normalized accuracy of the ML model used for inference in f. Let $EET^{\pi,\delta}$ represent the end-to-end execution time of the workflow G_s under the configurations π and δ, as estimated by the performance model from subsection 3.2.1. Likewise, let $C^{\pi,\delta}(f)$ denote the cost of function f, derived from the cost model, subject to (i) $M(f_i) = \pi(f_i)$ for all $1 \leq i \leq n$, (ii) $A(f_i) = \delta(f_i)$ for all $1 \leq i \leq n$,

(iii) $ET(f_i) = ETM(f_i, \pi(f_i), \delta(f_i))$ for all $1 \leq i \leq n$. Given a performance constraint PC, a budget constraint BC, and an end-to-end accuracy score constraint $EASC$, we now formulate the following three optimization problems.

Performance Optimization. The first optimization objective is to identify the memory configuration π and model configuration δ that minimize the average end-to-end execution time of the workflow, subject to the following constraints: the total average cost must not exceed the budget constraint BC, and the overall application accuracy must be at least as high as the accuracy constraint $EASC$.

$$\begin{aligned} \arg\min_{\pi,\delta} \quad & EET^{\pi,\delta}(G_s) \\ \text{subject to} \quad & \sum_{i=1}^{n} C^{\pi,\delta}(f_i) \leq BC, \\ & EAS^{\delta} \geq EASC \end{aligned} \tag{4.3}$$

Cost Optimization. The second optimization objective is to determine the memory configuration π and model configuration δ that minimize the average cost of the workflow. This optimization is subject to two constraints: the average execution time of the workflow must not exceed the performance constraint PC, and the overall application accuracy must meet or exceed the accuracy constraint $EASC$.

$$\begin{aligned} \arg\min_{\pi,\delta} \quad & \sum_{i=1}^{n} C^{\pi,\delta}(f_i) \\ \text{subject to} \quad & EET^{\pi,\delta}(G_s) \leq PC, \\ & EAS^{\delta} \geq EASC \end{aligned} \tag{4.4}$$

Accuracy Optimization. The last optimization objective is to find the memory configuration π and model configuration δ that maximize the end-to-end accuracy score of the workflow. This optimization is carried out under two constraints: the average execution time must not exceed the performance constraint PC, and the average cost of the workflow must remain within the budget constraint BC.

$$\begin{aligned} \arg\max_{\pi,\delta} \quad & EAS^{\delta}(G_s) \\ \text{subject to} \quad & EET^{\pi,\delta}(G_s) \leq PC, \\ & \sum_{i=1}^{n} C^{\pi,\delta}(f_i) \leq BC \end{aligned} \tag{4.5}$$

4.3 Computational Complexity of the Optimizations

We solve three separate optimization problems: (i) minimize end-to-end execution time given budget and accuracy constraints (Equation 4.3), (ii) minimize cost given performance and accuracy constraints (Equation 4.4), (iii) maximize accuracy given performance and budget constraints (Equation 4.5). Each problem has one objective and exactly two constraints. We prove that all of these problems are NP-hard by showing that their *decision* version is NP-complete.

Model and notation. For each function $f \in V$ in the workflow $G_s = (V, E)$, let Π_f be the finite set of memory options and Δ_f the finite set of model variants. A choice $(\pi_f, \delta_f) \in \Pi_f \times \Delta_f$ induces cost $C_f(\pi_f, \delta_f)$, execution time $ET_f(\pi_f, \delta_f)$,

and accuracy contribution δ_f. Given a full assignment (π, δ), we can compute

$$C^{\pi,\delta}(G_s), \quad EET^{\pi,\delta}(G_s), \quad EAS^{\delta}(G_s)$$

in polynomial time (a single traversal over G_s suffices).

Decision versions For each optimization goal, we define a decision problem by bounding the objective with a threshold.

Perf-Dec: Given bounds BC, $EASC$, and P, does there exist (π, δ) with $\sum_{i=1}^{n} C^{\pi,\delta}(f_i) \leq BC$, $EAS^{\delta}(G_s) \geq EASC$, and $EET^{\pi,\delta}(G_s) \leq P$?

Cost-Dec: Given bounds PC, $EASC$, and B, does there exist (π, δ) with $EET^{\pi,\delta}(G_s) \leq PC$, $EAS^{\delta}(G_s) \geq EASC$, and $\sum_{i=1}^{n} C^{\pi,\delta}(f_i) \leq B$?

Acc-Dec: Given bounds PC, BC, and A, does there exist (π, δ) with $EET^{\pi,\delta}(G_s) \leq PC$, $\sum_{i=1}^{n} C^{\pi,\delta}(f_i) \leq BC$, and $EAS^{\delta}(G_s) \geq A$?

Theorem 1. *Perf-Dec, Cost-Dec, and Acc-Dec are NP-complete. Therefore, the optimization problems in Equation 4.3–Equation 4.5 are NP-hard.*

Proof. In NP. For any (π, δ) we can evaluate $C^{\pi,\delta}(G_s)$, $EET^{\pi,\delta}(G_s)$, and $EAS^{\delta}(G_s)$ in polynomial time, so all three decision problems can be verified in polynomial time.

NP-hardness. We reduce from the 0/1 *Multi-dimensional Knapsack Problem* (MDKP) with two resource constraints, which is NP-complete [64, 65]. An MDKP instance gives n items, each with value v_i and two weights $w_i^{(1)}$ and $w_i^{(2)}$, capacities $W^{(1)}, W^{(2)}$, and a required total value V. The question is whether there exists a

subset S such that

$$\sum_{i \in S} w_i^{(1)} \leq W^{(1)}, \quad \sum_{i \in S} w_i^{(2)} \leq W^{(2)}, \quad \sum_{i \in S} v_i \geq V.$$

Construct a workflow G_s that is a chain of n functions $f_1, \ldots, f_n$. For each f_i create exactly two configuration pairs:

$$(\pi_i^0, \delta_i^0) \text{ (skip)} \quad \text{and} \quad (\pi_i^1, \delta_i^1) \text{ (take)}.$$

Define

$$C^{\pi_i^0, \delta_i^0}(f_i) = 0, \quad ET^{\pi_i^0, \delta_i^0}(f_i) = 0, \quad \delta_i^0(f_i) = 0,$$

$$C^{\pi_i^1, \delta_i^1}(f_i) = w_i^{(1)}, \quad ET^{\pi_i^1, \delta_i^1}(f_i) = w_i^{(2)}, \quad \delta_i^1(f_i) = v_i.$$

Set the global thresholds as follows and observe which decision problem we are answering:

For Perf-Dec*:* set $BC = W^{(1)}$, $EASC = V$, and $P = W^{(2)}$.

For Cost-Dec*:* set $PC = W^{(2)}$, $EASC = V$, and $B = W^{(1)}$.

For Acc-Dec*:* set $PC = W^{(2)}$, $BC = W^{(1)}$, and $A = V$.

Selecting (π_i^1, δ_i^1) corresponds to including item i in the knapsack. In our system, the end-to-end accuracy $EAS^{\delta}(G_s)$ is computed by a formula. For the hardness proof, we intentionally restrict to a special case in which this formula collapses to a plain sum of per-function accuracies. Since the workflow DAG is a chain and metrics are additive, the three inequalities in each decision problem exactly match

the MDKP constraints and value requirement. Hence, a feasible configuration exists if and only if the MDKP instance is feasible. The reduction is polynomial.

Thus, each decision problem is NP-hard and, with membership in NP, NP-complete. Any polynomial-time optimal solver for the corresponding optimization problem would decide its decision version, so each optimization problem is NP-hard. □

Remark on generality. The reduction employs a highly restricted instance: a linear DAG, two options per node, additive aggregation, and accuracy that depends solely on the model. Our real setting allows for arbitrary DAGs, numerous options, and non-linear path effects, so the full problems are at least as challenging.

Implication. Since the decision versions of our three configuration problems are NP-complete, the corresponding optimization problems are NP-hard. Thus, computing exact optimal solutions is impractical for realistic workflows with dozens of memory and model options. We therefore turn to heuristic search strategies that trade optimality guarantees for tractable runtime. Our design goals are (i) fast convergence, (ii) good solution quality in practice, and (iii) scalability with respect to the number of functions and configuration choices.

4.4 Algorithm Design

As proven in section 4.3, the optimization problem is NP-hard. Therefore, we resort to greedy algorithms with heuristics to efficiently search the solution space. One such heuristic is the Critical Path Method (CPM), which identifies the path

with the most significant cumulative execution time and focuses optimization on that path first. CPM originated with Kelley and Walker in 1959 and has since been applied in scheduling for scientific workflows [28, 66, 67].

In serverless workflows, the "longest path" is not necessarily the most critical for response time or cost because (i) paths are traversed with different transition probabilities, (ii) loops and cycles inflate expected invocations, and (iii) ML functions may use different model variants with different accuracies and runtimes. Prior work on serverless optimization therefore refines CPM by weighting paths with probabilistic and resource-aware terms (e.g., PRCP) [28]. We follow this idea and assign each simple path $s = f_1 e_1 f_2 e_2 \dots e_{n-1} f_n$ within the workflow a weight:

$$W(s) = TP(s) \cdot \sum_{i=1}^{n} ET(f_i) \cdot NI(f_i), \tag{4.6}$$

where $ET(f_i)$ is the execution time of function f_i, $NI(f_i)$ is its expected number of invocations after accounting for loops and branches, and $TP(s)$ is the transition probability of path s. The path with the highest $W(s)$ is treated as the current critical path. Given a set of M simple paths in the workflow, we define the path with the i^{th} highest weight as the i^{th} critical path, where $1 \leq i \leq M$.

Consider a function f_u with a new memory assignment $mem_v \in MLIST(f_u)$ and a new model configuration $acc_v \in ALIST(f_u)$. By assigning mem_v and the model with accuracy acc_v to f_u, we define the corresponding changes in system-level metrics as follows:

- $\Delta EET(f_u, mem_v, acc_v) = EET^{\pi', \delta'}(G_s) - EET_{\text{curr}}$,

- $\Delta C(f_u, mem_v, acc_v) = \sum_{i=1}^{n} C^{\pi',\delta'}(f_i) - C_{\text{curr}}$,
- $\Delta EAS(f_u, acc_v) = EAS^{\delta'}(G_s) - EAS_{\text{curr}}$,

such that (i) EET_{curr}, C_{curr}, and EAS_{curr} denote the end-to-end execution time, cost, and accuracy score under the previous configuration π and δ; (ii) $\pi'(f_u) = mem_v$, and $\delta'(f_u) = acc_v$; (iii) $\pi'(f_i) = \pi(f_i)$ and $\delta'(f_i) = \delta(f_i)$ for all $1 \leq i \leq n$, $i \neq u$.

4.4.1 Performance Optimization under Budget and Accuracy Constraints

algorithm 2 outlines the procedure for solving the performance optimization problem. The algorithm takes as input the budget constraint BC, the end-to-end accuracy score constraint $EASC$, and the serverless workflow G_s composed of n functions, where $V = \{f_1, f_2, \ldots, f_n\}$. It begins by assigning each function the lowest available memory configuration, denoted $\pi_{\min}$, and the smallest ML model for inference, denoted $\delta_{\min}$. That is, for each function f_i, we set $\pi_{\min}(f_i) = \min(MLIST(f_i))$ and $\delta_{\min}(f_i) = \min(ALIST(f_i))$. This configuration results in the maximum possible end-to-end execution time, serving as the worst-case baseline.

Guided by the fact that "for the same memory configurations, a larger ML model (with more parameters) results in higher latency and cost," we adopt a two-phase optimization strategy.

In the first phase, the algorithm aims to identify the minimum viable model configuration that satisfies the accuracy constraint. The key idea is to ensure

Algorithm 1: MinimumFeasibleModelConfig

Input : Budget constraint BC, accuracy constraint $EASC$, serverless workflow $G_s = (V, E, P, M, A, ET, ETTP, NI, C, MLIST, ALIST, ETM)$

Output: Memory configuration π and model configuration δ satisfying the constraints

$A \leftarrow A[f \mapsto \min(\text{ALIST}(f))], \forall f \in V$; `// Use minimal model configuration` δ_{min}

$G_{dl} \leftarrow (V, E', P', M, A, ET, NI, C, MLIST, ALIST, ETM)$; `// De-looped graph`

$NSP \leftarrow |\text{ASP}_{G_{dl}}(f_{str}, f_{end})|$; `// Number of simple paths in` G_{dl}

$\pi_{curr} \leftarrow \pi_{min}$; `// Initial memory configuration`

$\delta_{curr} \leftarrow \delta_{min}$; `// Initial model configuration`

$C_{curr} \leftarrow \sum_{i=1}^{n} C^{\pi_{min}, \delta_{min}}(f_i)$; `// Current cost`

$EAS_{curr} \leftarrow EAS^{\delta_{min}}(G_s)$; `// Current end-to-end accuracy score`

$t \leftarrow 1$;

while BC - $C_{curr} > 0$ *and* $EAS_{curr} < EASC$ *and* $t \leq NSP$ **do**

  $s_{cp} \leftarrow \text{FindCriticalPath}(G_{dl}, t)$; `// t-th critical path`

  foreach $f_i \in s_{cp}$ **do**

    foreach $acc_j \in ALIST(f_i)$ **do**

      Compute $\Delta EAS(f_i, acc_j)$ and $\Delta C(f_i, \pi_{curr}(f_i), acc_j)$;

  $(f_{tmp}, acc_{tmp}) \leftarrow \arg\max_{f_u, acc_v} BCR(f_u, acc_v)$

    s.t. $\Delta C(f_u, \pi_{curr}(f_i), acc_v) \leq BC - C_{curr}$;

  In cases where multiple model configurations yield the same maximum BCR, we break the tie by choosing the one that results in the smallest increase in cost.

  if f_{tmp} *and* acc_{tmp} *exist* **then**

    $A[f_{tmp}] \leftarrow acc_{tmp}$;

    $ET[f_{tmp}] \leftarrow ETM(f_{tmp}, \pi_{curr}(f_{tmp}), acc_{tmp})$;

    $\delta_{curr}(f_{tmp}) \leftarrow acc_{tmp}$;

    $C_{curr} \leftarrow C_{curr} + \Delta C(f_{tmp}, \pi_{curr}(f_{tmp}), acc_{tmp})$;

    $EAS_{curr} \leftarrow EAS^{\delta_{curr}}(G_s)$;

  else

    $t \leftarrow t + 1$;

return δ_{curr}

that no other model assignment could lead to lower latency while still meeting the required $EASC$. To do this, the algorithm iterates over all critical paths (sorted by their criticality) and considers changes to ML inference functions along these paths. For each candidate model configuration δ' (such that $\delta'(f_u) = acc_v$ and $\delta'(f_i) = \delta(f_i)$ for all $1 \leq i \leq n$, $i \neq u$) for function f_u, it computes the resulting end-to-end accuracy $EAS^{\delta'}(G_s)$, the cost increase $\Delta C(f_u, \pi_{\text{curr}}, \delta')$, and the accuracy gain $\Delta EAS(f_u, \delta')$. Each change is ranked using the following benefit-cost ratio (BCR) heuristic:

$$\begin{aligned} \text{BCR}_{\text{acc}}(f_u, acc_v) = w \cdot \min\Big(EAS^{\delta'}(G_s) - EASC,\, 0\Big) \\ - \Delta C(f_u, \pi_{\text{curr}}, \delta') \end{aligned} \tag{4.7}$$

where w is a weight used to penalize configurations that do not contribute to satisfying the accuracy constraint. The use of the min function ensures that accuracy gains beyond the threshold do not inflate the benefit term. The first term represents the benefit (improvement in accuracy), and the second term captures the cost penalty. For each function, the algorithm selects the model configuration that yields the highest BCR. It continues this process for all ML inference functions along all critical paths. This phase terminates once the $EASC$ constraint is satisfied. At the end of this phase, the algorithm has identified the lowest-cost, lowest-latency model configuration that satisfies the accuracy constraint.

We introduce two heuristics for the second phase of the optimization process, which focuses on improving performance through memory reconfiguration, assuming the model configuration has already been optimized in the previous phase. The

first heuristic, denoted as $\mathrm{BCR}_{\max}$, selects the memory configuration that yields the greatest reduction in execution time per unit increase in cost.

$$\mathrm{BCR}_{\max}(f_u, mem_v) = \left| \frac{\Delta \mathrm{EET}(f_u, mem_v)}{\Delta C(f_u, mem_v)} \right| \tag{4.8}$$

The second heuristic introduces a threshold-based strategy. We define a benefit–cost ratio threshold $\mathrm{BCR}_{\mathrm{th}}$ and use the slope of the latency–memory tradeoff curve as the basis for selection. Specifically, we estimate the slope using two adjacent memory configurations and define:

$$\mathrm{BCR}_{\mathrm{slope}}(f_u, mem_v) = \frac{\begin{array}{r}\mathrm{ETM}\Big(f_u, mem_{v+1}, \pi_{\mathrm{curr}}(f_u)\Big) \\ -\,\mathrm{ETM}\Big(f_u, mem_v, \pi_{\mathrm{curr}}(f_u)\Big)\end{array}}{mem_{v+1} - mem_v} \tag{4.9}$$

Let $mem_{v+1} \in \mathrm{MLIST}(f_u)$ denote the next higher selectable memory size such that $mem_{v+1} > mem_v$. For each function $f_i \in V$, the $\mathrm{BCR}_{\mathrm{slope}}$ strategy first estimates the slope of the latency–memory curve using least squares regression, denoted as $\beta(f_u)$. The algorithm then filters out all memory configurations whose BCR is less than $\beta(f_u) \cdot \mathrm{BCR}_{\mathrm{th}}$ from the set of viable memory options. After this step, each function f_i retains only memory configurations that satisfy the threshold condition $\beta(f_i) \cdot \mathrm{BCR}_{\mathrm{th}}$ for all $1 \leq i \leq n$ and for all $mem_j \in \mathrm{MLIST}(f_i)$. In each critical path iteration, the algorithm selects the function f_u and memory size $mem_v \in \mathrm{MLIST}(f_u)$ that yields the maximum reduction in end-to-end response time.

With the two heuristics defined, we now complete the description of the second phase. Similar to the first phase, the algorithm iterates over the critical paths. For each function and its corresponding memory configurations, it computes the respective heuristic values. If a function–memory pair exists such that the cost is reduced without increasing the end-to-end response time of the workflow (i.e., $\Delta C < 0$ and $\Delta \text{EET} \leq 0$), the algorithm selects the pair that achieves the most significant cost reduction.

If no such configuration exists, the algorithm chooses the function and memory size with the highest BCR value. When no feasible memory configuration is found within the t^{th} critical path (where t is initialized to 1), the algorithm moves on to the $(t+1)^{th}$ critical path in the next iteration. The process continues until no feasible memory configuration can be found in the least critical path under the current budget surplus.

4.4.2 Cost Optimization under Performance and Accuracy Constraints

We follow a similar approach to the one described in the previous optimization phase. The workflow is initially configured with the maximum memory assignment $\pi_{\max}$ and the most lightweight model configuration $\delta_{\min}$. Specifically, for each function f_i, we set $\pi_{\max}(f_i) = \max(MLIST(f_i))$ and $\delta_{\min}(f_i) = \min(ALIST(f_i))$. This initialization ensures the workflow achieves the fastest possible end-to-end execution time.

The first phase focuses on identifying the minimum viable model configuration that satisfies both latency and accuracy constraints. The algorithm begins by selecting the least critical path and examines all feasible memory allocations for functions along that path. For each combination, it computes the change in execution time (ΔEET) and cost (ΔC).

Suppose there exists a memory setting that reduces the overall execution time without increasing the cost (i.e., $\Delta\text{EET} < 0$ and $\Delta C \leq 0$), the algorithm selects the configuration that yields the most significant reduction in response time. If such a setting is not available, it instead chooses the one that produces the largest BCR, while still respecting the performance constraint.

Suppose no suitable memory configuration is found in the current t^{th} critical path (starting from the least critical one). In that case, the algorithm proceeds to the following more critical path $(t-1)$, and continues this process iteratively. The algorithm terminates when no improvement is possible even along the most crucial path, given the current surplus in performance.

It is notable that in this optimization phase, we treat the increase in execution time as the cost metric in Equation 4.7. Accordingly, the heuristics defined in Equation 4.8 and Equation 4.9 are applied in an inverted manner to reflect this change.

4.4.3 Accuracy Optimization under Performance and Cost Constraints

Unlike the cost and performance optimization scenario, we cannot rely on a two-phase strategy when tackling the accuracy-driven optimization problem. In the cost-performance setting, we could begin with a configuration that yields the fastest execution time and gradually reduce the cost, as long as the performance constraints were still met. This stepwise approach, however, breaks down when optimizing for accuracy. Starting from low-accuracy (and typically faster) model configurations often leads to solutions that fall short of the application's required accuracy. Furthermore, simply upgrading to more accurate models does not guarantee feasibility under tight cost and latency constraints, as the trade-offs involved are highly non-linear and model-specific. Consequently, the optimization must treat accuracy, cost, and latency as interdependent objectives and consider them jointly rather than optimizing them in separate phases.

To initialize the greedy algorithm with a meaningful starting point, we begin by assigning the lowest possible model configuration $\delta_{\min}$ to the application. Specifically, for each function f_i, we set $\delta_{\min}(f_i) = \min(ALIST(f_i))$. With the model configuration fixed, we then determine the memory configuration that yields the lowest cost while satisfying the given performance constraint. This step mirrors the second phase of the cost optimization procedure discussed in the previous section. The resulting memory configuration is guaranteed to be the most cost-efficient among all combinations that respect the performance constraint under the chosen model configuration.

To guide the optimization, we define a benefit-cost ratio for memory and accuracy selection, denoted as:

$$BCR_{mem-acc} = \frac{\Delta EAS}{\Delta C_{norm} + \Delta EET_{norm}}, \tag{4.10}$$

in which the benefit is defined as the positive increase in accuracy, while the cost corresponds to the sum of the normalized increases in average cost and average execution time observed after applying a memory-model configuration change to the workflow.

At each iteration, the algorithm selects the function and memory-model pair with the highest $BCR_{\text{mem-acc}}$ value. If no feasible configuration can be found within the current t^{th} critical path (starting with $t = 1$), the algorithm proceeds to the following less critical path $(t + 1)$, and so on. The process terminates when no valid memory-model configuration exists in the least critical path that satisfies the remaining budget and performance constraints.

Algorithm 2: Performance Optimization

Input : Budget constraint BC, accuracy constraint $EASC$, serverless workflow G_s

Output: Memory configuration π and model configuration δ

$M \leftarrow M[f \mapsto \min(\text{MLIST}(f))], \forall f \in V$; `// Use minimal memory configuration` π_{min}

$G_{dl} \leftarrow (V, E', P', M, A, ET, NI, C, MLIST, ALIST, ETM)$; `// De-looped graph`

$NSP \leftarrow |\text{ASP}_{G_{dl}}(f_{str}, f_{end})|$; `// Number of simple paths in` G_{dl}

$\pi_{curr} \leftarrow \pi_{min}$; `// Initial memory configuration`

$\delta_{curr} \leftarrow \text{MinimumFeasibleModelConfig}(BC, EAS, G_s)$

$C_{curr} \leftarrow \sum_{i=1}^{n} C^{\pi_{min}, \delta_{curr}}(f_i)$; `// Current cost`

$EET_{curr} \leftarrow EET^{\pi_{min}, \delta_{curr}}(G_s)$; `// Initial end-to-end runtime`

$t \leftarrow 1$;

while $BC - C_{curr} > 0$ *and* $t \leq NSP$ **do**

  $s_{cp} \leftarrow \text{FindCriticalPath}(G_{dl}, t)$; `// t-th critical path`

  foreach $f_i \in s_{cp}$ **do**

    foreach $mem_j \in MLIST(f_i)$ **do**

      Compute $\Delta EET(f_i, mem_j, \delta_{curr}(f_i))$ and $\Delta C(f_i, mem_j, \delta_{curr}(f_i))$;

  if $\exists mem_v \in MLIST(f_u) : \Delta EET(f_u, mem_v, \delta_{curr}(f_u)) \leq 0 \wedge \Delta C(f_u, mem_v, \delta_{curr}(f_u)) < 0$ **then**

    $(f_{tmp}, mem_{tmp}) \leftarrow \arg\min_{f_u, mem_v} \Delta C(f_u, mem_v, \delta_{curr}(f_u))$

  else

    $(f_{tmp}, mem_{tmp}) \leftarrow \arg\min_{f_u, mem_v} \Delta EET(f_u, mem_v, \delta_{curr}(f_u))$

      s.t. $\Delta C(f_u, mem_v, \delta_{curr}(f_u)) \leq BC - C_{curr}$ and $\Delta EET(f_u, mem_v, \delta_{curr}(f_u)) < 0$;

    This is the stage where both $BCR_{\max}$ and BCR_{slope} are applied. We rank options using their corresponding BCR values.

  if f_{tmp} *and* mem_{tmp} *exist* **then**

    $M[f_{tmp}] \leftarrow mem_{tmp}$;

    $ET[f_{tmp}] \leftarrow ETM(f_{tmp}, mem_{tmp}, \delta_{curr}(f_{tmp}))$;

    $\pi_{curr}(f_{tmp}) \leftarrow mem_{tmp}$;

    $C_{curr} \leftarrow C_{curr} + \Delta C(f_{tmp}, mem_{tmp}, \delta_{curr}(f_{tmp}))$;

    $EET_{curr} \leftarrow EET_{curr} + \Delta EET(f_{tmp}, mem_{tmp}, \delta_{curr}(f_{tmp}))$;

  else

    $t \leftarrow t + 1$;

return $\pi_{curr}, \delta_{curr}$

Chapter 5

Experimental Evaluation

To evaluate our analytical models and optimization techniques, we conducted extensive experiments with realistic workloads on a public cloud. Here, we will explain the setup of the evaluation environment and how OptiServe was evaluated. All framework code and experimental results are available in the OptiServe artifact repository[1].

5.1 Cloud Platform and Deployment Setup

We use AWS as the cloud platform to implement and evaluate OptiServe. As illustrated in Figure 1.1, the evaluation begins by collecting the application logic and service-level objectives (SLOs) from the user. We deploy the serverless functions on AWS Lambda, the application control flow on AWS Step Functions, and the machine learning models used for inference tasks in AWS S3. We also use Amazon CloudWatch as the monitoring service for collecting and analyzing logs.

[1]https://github.com/pacslab/OptiServe

TABLE 5.1: Unified summary of function benchmarks and corresponding ML model variants with parameter counts in millions

Function	Type	Description	Model Variant	Param (M)
FileWriteDelete	General	Disk I/O-intensive	NA	NA
HashCompute	General	CPU-intensive	NA	NA
NetworkDownload	General	Network-intensive	NA	NA
Image Classification	ML Inference	Using ResNet variants.	ResNet-18 [68]	11.7
			ResNet-34	21.8
			ResNet-50	25.6
			ResNet-101	44.5
			ResNet-152	60.2
Object Detection	ML Inference	Using YOLO variants.	YOLOv10n [69]	∼2.0
			YOLOv10s	∼11.0
			YOLOv10m	∼25.0
			YOLOv10l	∼49.0
Sequence Classification	ML Inference	Using BERT variants.	DistilBERT-base-uncased [70]	66
			BERT-base-uncased [71]	110
			RoBERTa-base [72]	125

5.2 Function Benchmarks

We use six different functions, each designed to stress a specific aspect of the system. Three of them are machine learning tasks: image classification with ResNet

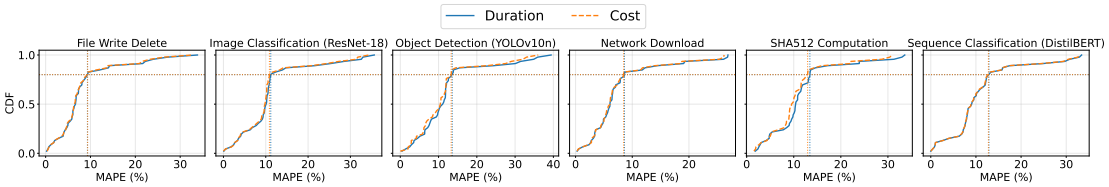

FIGURE 5.1: CDF of the MAPE for the execution-time and cost models across the six profiled functions

[68] variants, object detection with YOLO [69] variants, and sequence classification with BERT [71] variants. The other three are general-purpose functions targeting CPU usage, disk I/O, and network bandwidth. Table 5.1 provides details of the functions used in the optimization phase. For the image classification and object detection tasks, we use a batch of 24 images during evaluation. For sequence classification, we evaluate the model on a batch of 16 text samples, each composed of 8 distinct paragraphs.

The general functions were sampled and modelled across a memory range of 128MB to 3008MB. In contrast, the ML functions required significantly more resources, with a minimum of 1024MB just to run properly, and were tested up to 10GB of memory. We summarize the ML model variants used in our experiments for modelling and optimization in Table 5.1.

Notably, the general functions were deployed directly on AWS Lambda using the standard ZIP method. For the ML functions, we used container image deployment to gain full control over the runtime environment and dependencies, which also made it easier to include custom libraries.

We begin by modelling each general-purpose and ML function along with all the model variants used for their respective tasks, following the methodology outlined in section 3.1. To mitigate the impact of performance variability inherent in

serverless execution, we sample each memory configuration three times for each model variant. If we encounter anomalies, such as an unexpected increase in execution time with higher memory, we redeploy and retest the function until the anomaly is resolved.

Once we extract the exponential execution time model for each function based on memory allocation, we immediately compute the corresponding cost model using Equation 3.2.

The cumulative distribution functions (CDFs) of the mean absolute percentage error (MAPE) for the execution-time and cost models across all six serverless functions are shown in Figure 5.1. Each configuration was profiled using 100 warm invocations every six hours, for a total of four rounds, with memory varied in 128 MB increments for general functions and 512 MB increments for ML functions. The vertical and horizontal dashed lines mark the 80th-percentile MAPE values, which are approximately 10 percent. Across all functions, both models exhibit tightly concentrated error distributions, with 80 percent of predictions falling below this level. With these models in place, we proceed to the application-level modelling phase.

5.3 Application Workflows

We designed six different workflows to capture the full range of structural patterns commonly found in serverless applications, including parallelism, branching, loops, and cycles. As noted in [73], approximately 82% of serverless applications deployed on commercial platforms consist of five or fewer functions. To ensure

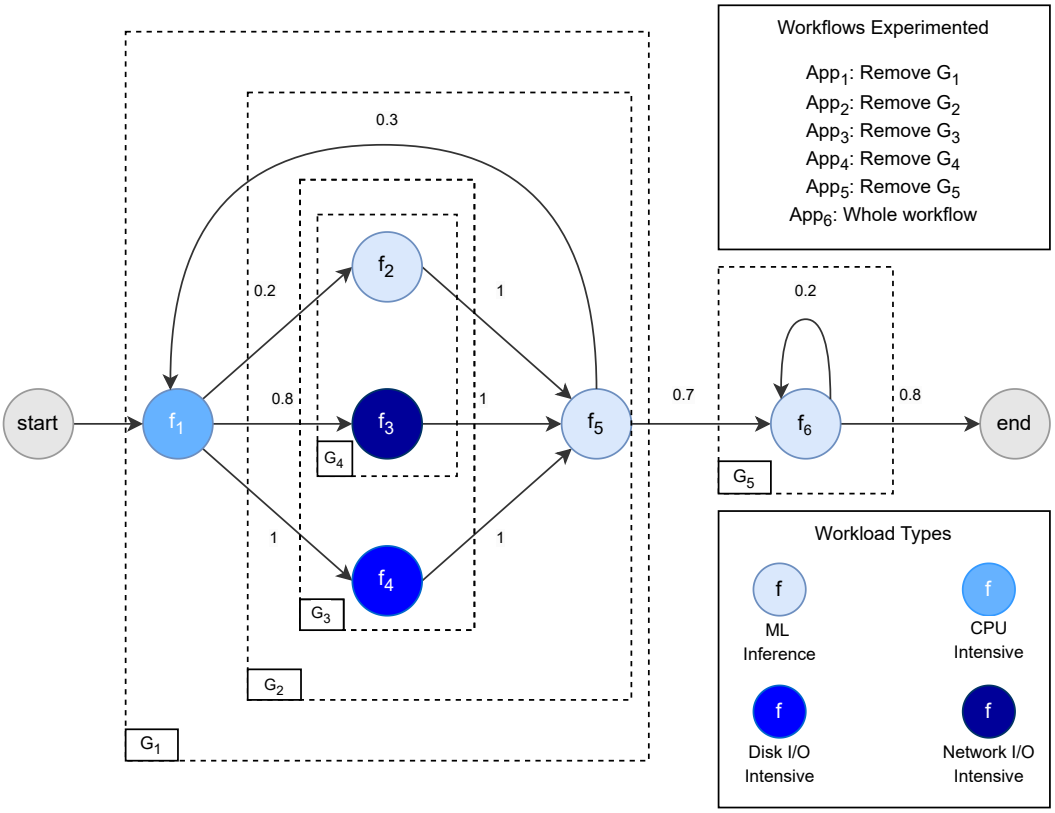

FIGURE 5.2: An overview of the serverless workflows modelled and optimized

that OptiServe applies to real-world scenarios, we evaluate and optimize applications ranging from a single function up to six functions. As also noted in [74], serverless applications that include machine learning inference typically have only one or two functions dedicated to ML serving. To reflect this common structure, we evaluate applications with up to three ML inference functions, ensuring our experiments align with realistic deployment patterns. This allows us to validate that OptiServe remains effective across the majority of practical deployment cases. Profiling every possible combination of memory and model variant for all functions within each workflow is prohibitively expensive and time-consuming. To

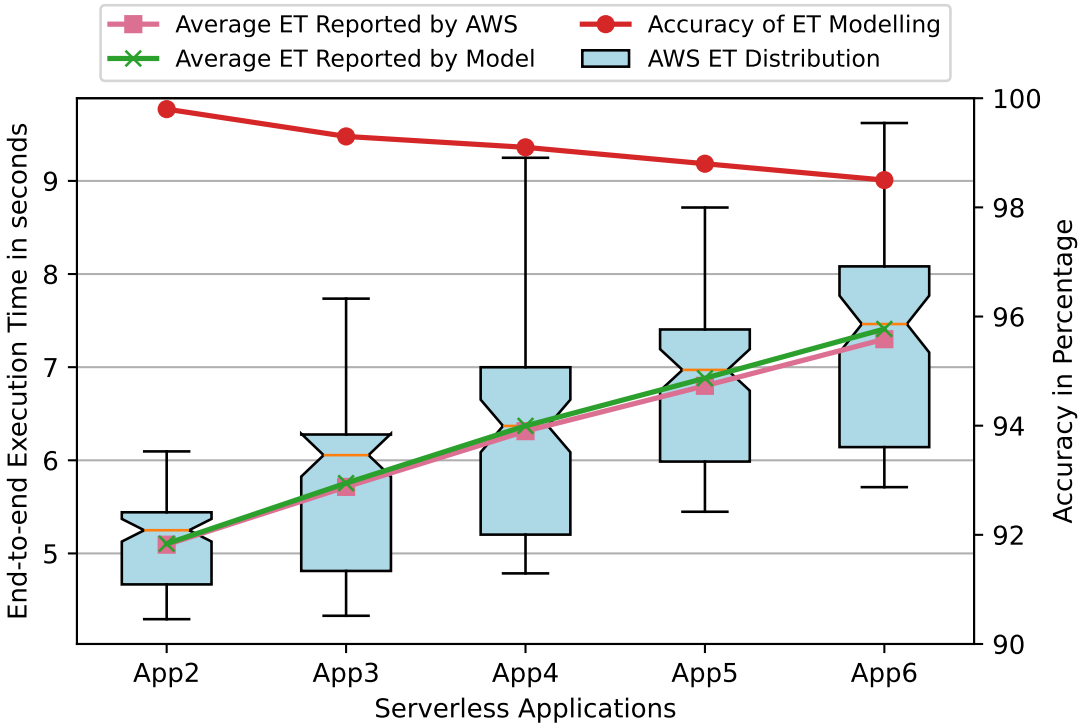

FIGURE 5.3: The performance model achieves an average accuracy of 99.1%. The box plot reports the minimum, 25th percentile, median, 75th percentile, and maximum end-to-end execution times. The notches represent the 95% confidence interval for the median execution time.

address this, we leverage the performance and cost models developed in the function modelling phase and apply them to the application-level models described in section 3.2. This approach enables us to efficiently estimate the average execution time and cost of each workflow in real time. Figure 5.2 summarizes all the workflows evaluated and modelled in our study. The numbers on the edges represent the transition probabilities between functions. In total, we tested six different applications. Each workflow corresponds to a variation of the full graph, created by removing a specific subgraph. App_i consists of i different functions. The functions are grouped into four types of workloads, as indicated in the figure.

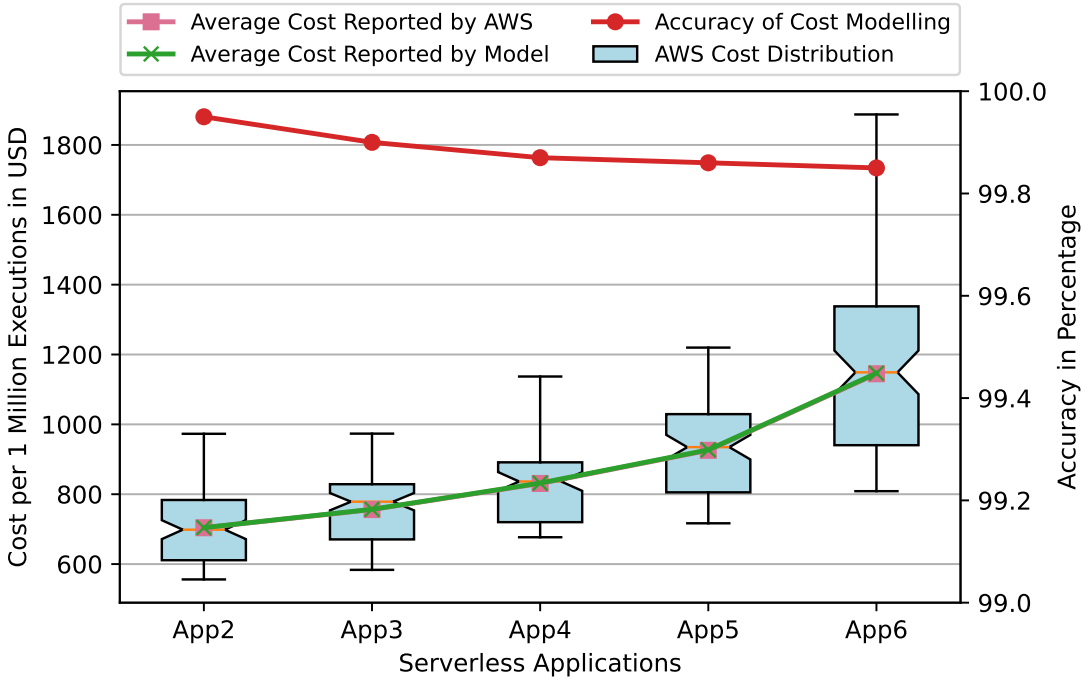

FIGURE 5.4: The experimental evaluation of the cost model shows an average prediction accuracy of 99.89%.

We evaluate the performance and cost modelling of the proposed workflows using their minimum-accuracy configurations. App_1 is excluded from this analysis, as it consists of a single function and lacks structural complexity. The remaining functions and applications are deployed on AWS Lambda and Step Functions, respectively, and each application is executed across five two-hour periods, separated by two-hour intervals. Within each period, applications run continuously with a 10-second gap between consecutive invocations. To avoid transient performance fluctuations, we discard the first and last ten minutes of each period and use only the stable execution window for analysis. This process yields 3,000 warm invocations per application for evaluation.

Using the workflow structure along with the average duration and allocated

memory of each function as inputs, we apply the proposed performance and cost models to estimate the average end-to-end response time and cost of the five serverless applications. We then analyze 3,000 warm invocations per application and compare the model predictions with the execution-time and billing information reported by AWS. The results of this comparison are presented in Figure 5.3 and Figure 5.4.

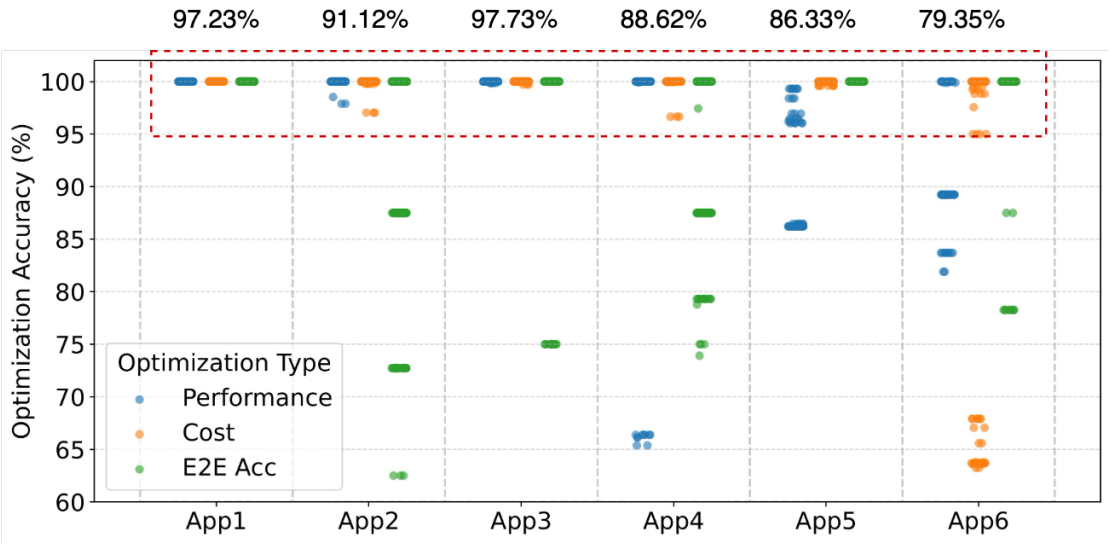

FIGURE 5.5: The results of the tri-objective optimization are presented for each application

5.4 Experimental Evaluation

To evaluate the effectiveness of OptiServe, we applied it to deploy and optimize the serverless applications described in section 5.3. Since there is no prior system that addresses our specific optimization goals, jointly optimizing memory and model configurations under performance, cost, and accuracy constraints for serverless applications with complex workflows, we designed a rigorous self-verification strategy to validate the optimizer's accuracy.

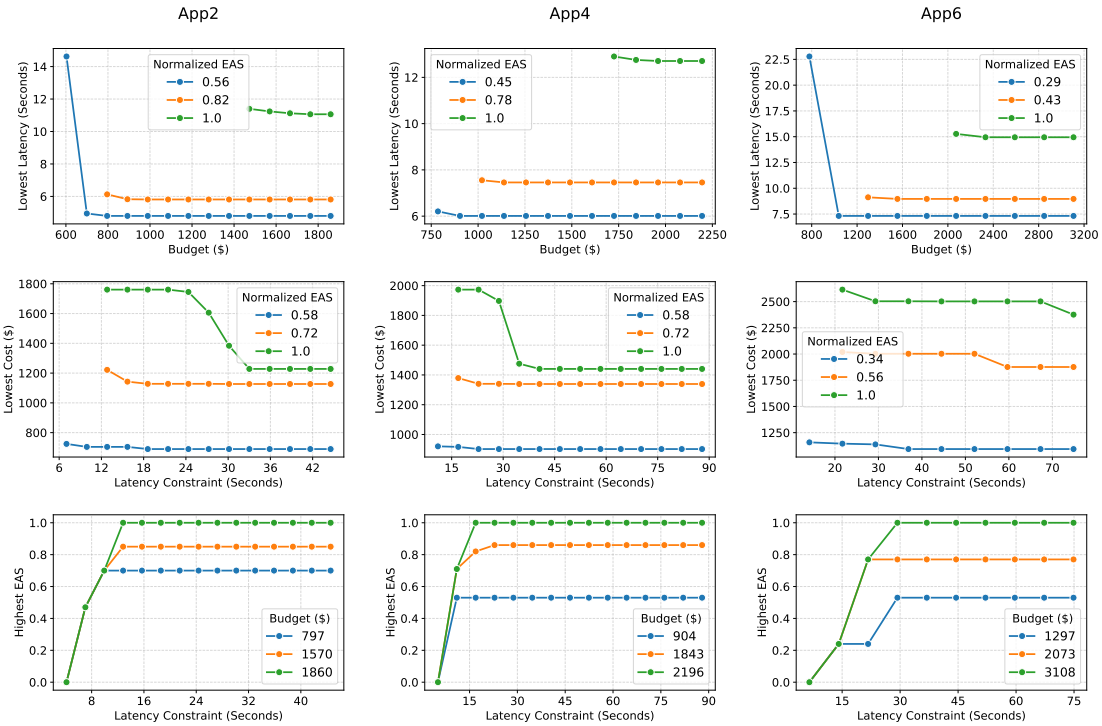

FIGURE 5.6: What-if analysis across three applications

For each application, we defined a set of constrained optimization problems and used OptiServe to compute solutions. To assess the quality of these solutions, we performed an exhaustive search over the space of valid configurations. This involved evaluating all combinations of memory and model settings for every function in the workflow.

Given the large configuration space, evaluating every possible memory size and model configuration was computationally impractical. To make the exhaustive search feasible while still capturing meaningful variations, we discretized the memory space with practical step sizes. For general (non-ML) functions, we used increments of 192 MB within the 128 MB to 3008 MB range. For ML functions, we used a coarser step size of 512 MB, ranging from 1024 MB up to 10 GB. This discretization strikes a balance between search efficiency and modelling precision.

Using the performance and cost models from section 3.2, we calculated the average end-to-end execution time and cost for every configuration in the search space. For each optimization scenario, we identified the best feasible solution, satisfying all constraints, as the global optimum.

We then compared OptiServe's output against this ideal solution. For each case, we report the deviation from the global optimum, which allows us to measure how close the optimizer gets to the best possible result without performing brute-force enumeration.

We begin by evaluating how accurately OptiServe can identify the ideal memory-model configuration. For each application shown in Figure 5.2, we generated 250 distinct optimization problems, each with different constraint settings and EAS formulas, and passed them to OptiServe. The resulting configurations are compared against the ideal solutions, and the outcomes are summarized in Figure 5.5 where problems are grouped by application and optimization type. Each point in the plot represents a distinct optimization problem defined by two constraints. The y-axis indicates the accuracy of OptiServe's solution relative to the optimal one. The percentage displayed above each application's box indicates the proportion of optimization problems in which OptiServe achieved at least 95% of the ideal accuracy for that application. Overall, OptiServe reached this threshold in 89.64% of the cases.

We also performed several what-if analysis experiments on our benchmark applications, with selected results shown in Figure 5.6. Each row corresponds to one of the optimization objectives offered by OptiServe: minimizing latency and cost,

or maximizing end-to-end accuracy. The y-axis indicates the optimized value for each objective, and the problems are grouped based on a fixed constraint to illustrate how varying constraints affect the outcome. The costs are reported in USD per 1 million executions of the application on AWS. Each figure highlights key trade-offs between cost, latency, and model accuracy under different constraint settings. For example, in the middle figure, which represents cost optimization for App_4, we observe that achieving the highest possible model configuration (normalized EAS of 1) while maintaining average latency under 40 seconds requires a minimum budget of approximately \$1420. The plot also reveals alternative options: users can either choose lower-accuracy models to reduce both cost and latency, or accept higher costs to achieve better latency for the same model configuration.

Another example is the lower-right figure, which illustrates EAS optimization for App_6. From this plot, we can see that with a budget of \$3108, it is possible to deploy the most accurate models (EAS = 1), but the average latency cannot go below 30 seconds. These scenarios demonstrate how OptiServe enables users to easily navigate performance, cost, and accuracy trade-offs in real-world deployments.

Chapter 6

Conclusion

6.1 Summary

This thesis studied the problem of *deploying and tuning serverless applications with machine learning (ML) workloads* under competing objectives. In modern FaaS platforms, choosing a memory configuration (and implicitly a CPU share) and selecting ML model variants are tightly coupled decisions that directly affect end-to-end latency, cost, and inference quality. When these decisions are made independently per function, they often fail to satisfy application-level constraints once workflow structure, branching behavior, and critical-path effects are considered.

To address this challenge, we introduced **OptiServe**, a framework that supports *tri-objective optimization* for serverless ML workflows. OptiServe models the performance and cost of individual functions and propagates those models through an application workflow to estimate end-to-end behavior. In addition,

OptiServe incorporates an end-to-end accuracy score (EAS) that captures the impact of ML model selection at the workflow level via a user-defined aggregation function. Building on these models, OptiServe provides optimization capabilities for multiple scenarios, including: (i) performance maximization under budget and accuracy constraints, (ii) cost minimization given latency and accuracy targets, and (iii) accuracy maximization under latency and budget constraints.

We validated OptiServe using real deployments on AWS (Lambda and Step Functions) and a set of benchmark workflows spanning multiple workload types and sizes. The results show that OptiServe can accurately model end-to-end performance and cost, and it can efficiently identify configurations that closely match ideal solutions derived from exhaustive enumeration under discretized configuration spaces. Across a large set of constrained optimization problems, OptiServe achieved solutions within 95% of the ideal accuracy in a majority of cases, while also enabling extensive what-if analyses in under a minute. Overall, these results demonstrate that analytical modelling combined with lightweight, deterministic optimization can make workflow-level rightsizing and model selection practical for real serverless ML deployments.

6.2 Discussions, Limitations, and Future Work

This section puts the results of this thesis into perspective. We first discuss what OptiServe enables in practical deployments and why its analytical, workflow-level approach is useful beyond the specific workloads evaluated. We then summarize the key limitations that stem from the opacity of serverless platforms, the current focus on average-case metrics, and the restricted set of configuration knobs

considered. Finally, we outline concrete directions for extending OptiServe toward broader deployment scenarios, including richer configuration spaces, hybrid CPU/GPU environments, and tail-latency-aware optimization.

6.2.1 Discussion

OptiServe is designed to support practitioners who must balance cost, latency, and ML quality in production settings. Two aspects are particularly important in practice. First, OptiServe supports multiple optimization objectives and constraints, allowing users to express different operational goals (e.g., minimizing cost under strict SLOs versus maximizing quality under a budget). Second, OptiServe enables *fast what-if analysis*, making it possible to explore how tightening or relaxing constraints changes the recommended configuration and the implied trade-offs. This is valuable in early-stage deployment planning as well as ongoing SLO/budget negotiations, where teams often need to quantify trade-offs rather than rely on ad-hoc tuning.

A key design choice in OptiServe is using analytical models of cost and performance, rather than relying on exhaustive profiling across the full configuration space. This improves practicality and supports rapid exploration of alternative configurations. At the same time, OptiServe preserves flexibility by allowing application developers to define how accuracy composes across workflow stages (via EAS), which is necessary because different applications interpret end-to-end quality differently.

6.2.2 Limitations

While OptiServe demonstrates strong results, several limitations remain:

- **Model fidelity under non-stationarity and interference.** Serverless performance can vary due to multi-tenancy, background interference, and platform-level policy changes. While OptiServe models end-to-end behavior accurately in controlled experiments, shifts in runtime conditions can reduce prediction fidelity and may require periodic recalibration.

- **Average-case focus.** The current modelling and optimization primarily target average end-to-end execution time and cost. Many production services care about tail latency (e.g., p95/p99) and worst-case behavior, especially for interactive applications. Extending OptiServe to optimize tail metrics remains an important step toward broader deployment.

- **Limited configuration knobs.** OptiServe currently focuses on memory configurations and ML model selection. Other important knobs, such as batching, concurrency limits, and model partitioning, can substantially change performance and cost, but are not yet part of the optimization space.

6.2.3 Future Work

This thesis opens several natural directions for extending OptiServe:

- **Additional configuration knobs.** Incorporate batching and model partitioning into the optimization space to better capture throughput-aware inference behavior and pipeline-style execution.

- **GPU-aware and hybrid deployments.** Extend OptiServe to support GPU selection as a tunable parameter and enable hybrid deployment across both IaaS and FaaS platforms. This would make the framework applicable to GPU-enabled inference workloads and address limitations of current serverless offerings.

- **Tail-aware optimization.** Extend the modelling and optimization pipeline to incorporate tail latency metrics (e.g., p95/p99) and uncertainty-aware constraints, improving suitability for strict SLO environments.

In summary, OptiServe demonstrates that workflow-level, accuracy-aware optimization for serverless ML applications can be achieved efficiently using analytical models and deterministic heuristics. By extending the configuration space and incorporating tail-aware and GPU-aware capabilities, future versions of OptiServe can broaden its applicability to emerging inference workloads and deployment environments.

Bibliography

[1] "Cloud Computing Services - Amazon Web Services (AWS) — aws.amazon.com." `https://aws.amazon.com/`. [Accessed 13-02-2025].

[2] "Cloud Computing Services | Google Cloud — cloud.google.com." `https://cloud.google.com/`. [Accessed 13-02-2025].

[3] "Cloud Computing Services | Microsoft Azure — azure.microsoft.com." `https://azure.microsoft.com/`. [Accessed 13-02-2025].

[4] Datadog, "The State of Serverless — datadoghq.com." `https://www.datadoghq.com/state-of-serverless`, 2023. [Accessed 30-01-2025].

[5] A. Moghimi, J. Hattori, A. Li, M. Ben Chikha, and M. Shahrad, "Parrotfish: Parametric regression for optimizing serverless functions," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pp. 177–192, 2023.

[6] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "{INFaaS}: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411, 2021.

[7] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "Sloc: Service level objectives for next generation cloud computing," *IEEE Internet Computing*, vol. 24, no. 3, pp. 39–50, 2020.

[8] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A case for serverless machine learning," in *Workshop on Systems for ML and Open Source Software at NeurIPS*, vol. 2018, pp. 2–8, 2018.

[9] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, IEEE, 2020.

[10] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *2018 IEEE International conference on cloud engineering (IC2E)*, pp. 257–262, IEEE, 2018.

[11] J. Chen, F. Xu, Y. Gu, L. Chen, F. Liu, and Z. Zhou, "Harmonybatch: Batching multi-slo dnn inference with heterogeneous serverless functions," *arXiv preprint arXiv:2405.05633*, 2024.

[12] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, "Gillis: Serving large neural networks in serverless functions with automatic model partitioning," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 138–148, IEEE, 2021.

[13] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "{ServerlessLLM}:{Low-Latency} serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 135–153, 2024.

[14] K. Mahajan and R. Desai, "Serving distributed inference deep learning models in serverless computing," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 109–111, IEEE, 2022.

[15] Z. Hong, J. Lin, S. Guo, S. Luo, W. Chen, R. Wattenhofer, and Y. Yu, "Optimus: warming serverless ml inference via inter-function model transformation," in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 1039–1053, 2024.

[16] C. Zhang, M. Yu, W. Wang, and F. Yan, "Enabling cost-effective, slo-aware machine learning inference serving on public cloud," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1765–1779, 2020.

[17] Y. Hu, W. Pang, X. Liu, R. Ghosh, B. Ko, W.-H. Lee, and R. Govindan, "Rim: Offloading inference to the edge," in *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, pp. 80–92, 2021.

[18] S. Ghafouri, K. Razavi, and M. Salmani, "[solution] ipa: Inference pipeline adaptation to achieve high accuracy and cost-efficiency," *Journal of Systems Research, 4 (1)*, 2024.

[19] J. R. Gunasekaran, C. S. Mishra, P. Thinakaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Cocktail: A multidimensional optimization for model serving in cloud," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 1041–1057, 2022.

[20] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg, "{Model-Switching}: Dealing with fluctuating workloads in {Machine-Learning-as-a-Service} systems," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[21] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A {Low-Latency} online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 613–627, 2017.

[22] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "Jalad: Joint accuracy- and latency-aware deep structure decoupling for edge-cloud execution," in *2018 IEEE 24th international conference on parallel and distributed systems (ICPADS)*, pp. 671–678, IEEE, 2018.

[23] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX annual technical conference (USENIX ATC 20)*, pp. 205–218, 2020.

[24] S. Luccioni, Y. Jernite, and E. Strubell, "Power hungry processing: Watts driving the cost of ai deployment?," in *Proceedings of the 2024 ACM conference on fairness, accountability, and transparency*, pp. 85–99, 2024.

[25] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pp. 1868–1877, IEEE, 2022.

[26] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, "On-demand container loading in {AWS} lambda," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 315–328, 2023.

[27] V. Iyer, S. Lee, S. Lee, J. J. Kim, H. Kim, and Y. Shin, "Automated backend allocation for multi-model, on-device ai inference," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 3, pp. 1–33, 2023.

[28] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, 2020.

[29] J. Duan, S. Qian, H. Hu, D. Yang, J. Cao, and G. Xue, "Pipeco: Pipelining cold start of deep learning inference services on serverless platforms," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 9, no. 2, pp. 1–23, 2025.

[30] T. Pusztai and S. Nastic, "Chunkfunc: Dynamic slo-aware configuration of serverless functions," *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[31] J. Cho, D. Zad Tootaghaj, L. Cao, and P. Sharma, "Sla-driven ml inference framework for clouds with heterogeneous accelerators," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 20–32, 2022.

[32] L. Li, S. Pandey, T. Flynn, H. Liu, N. Wheeler, and A. Hoisie, "Simnet: Accurate and high-performance computer architecture simulation using deep learning," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–24, 2022.

[33] G. Yang, C. Shin, J. Lee, Y. Yoo, and C. Yoo, "Prediction of the resource consumption of distributed deep learning systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–25, 2022.

[34] Y. Liu, B. Jiang, T. Guo, Z. Huang, W. Ma, X. Wang, and C. Zhou, "Funcpipe: A pipelined serverless framework for fast and cost-efficient training of deep learning models," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 3, pp. 1–30, 2022.

[35] S. S. Shubha, H. Shen, and A. Iyer, "{USHER}: Holistic interference avoidance for resource optimized {ML} inference," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 947–964, 2024.

[36] W. Shin, W.-H. Kim, and C. Min, "Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 663–677, 2022.

[37] Z. Zhou, X. Wei, J. Zhang, and G. Sun, "{PetS}: A unified framework for {Parameter-Efficient} transformers serving," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 489–504, 2022.

[38] I. Gim, G. Chen, S.-s. Lee, N. Sarda, A. Khandelwal, and L. Zhong, "Prompt cache: Modular attention reuse for low-latency inference," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 325–338, 2024.

[39] Y. Zhao, C.-Y. Lin, K. Zhu, Z. Ye, L. Chen, S. Zheng, L. Ceze, A. Krishnamurthy, T. Chen, and B. Kasikci, "Atom: Low-bit quantization for efficient and accurate llm serving," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 196–209, 2024.

[40] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 419–433, 2020.

[41] X. Pang, L. Liu, Y. Zhang, Z. Chen, Z. Ding, D. Cheng, and X. Zhou, "Featherlight stateful webassembly for serverless inference workflows," *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[42] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

[43] H. Tang, Z. Liu, X. Li, Y. Lin, and S. Han, "Torchsparse: Efficient point cloud inference engine," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 302–315, 2022.

[44] C. Lin, Z. Chen, Z. Zhang, and J. Liu, "Top: task-based operator parallelism for asynchronous deep learning inference on gpu," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[45] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "{ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 303–320, 2022.

[46] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "Wisefuse: Workload characterization and dag transformation for serverless workflows," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–28, 2022.

[47] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: Enable efficient workflow execution for function-as-a-service," in *Proceedings of the 27th acm international conference on architectural support for programming languages and operating systems*, pp. 782–796, 2022.

[48] A. Kumar, A. Sivasubramaniam, and T. Zhu, "Splitrpc: A {Control+ Data} path splitting rpc stack for ml inference serving," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 2, pp. 1–26, 2023.

[49] Z. Zhang, C. Jin, and X. Jin, "Jolteon: Unleashing the promise of serverless for serverless workflows," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 167–183, 2024.

[50] R. Basu Roy and D. Tiwari, "Starship: Mitigating i/o bottlenecks in serverless computing for scientific workflows," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 8, no. 1, pp. 1–29, 2024.

[51] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[52] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. Kalbarczyk, T. Başar, and R. K. Iyer, "Power-aware deep learning model serving with {μ-Serve}," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 75–93, 2024.

[53] G. Sadeghian, M. Elsakhawy, M. Shahrad, J. Hattori, and M. Shahrad, "{UnFaaSener}: Latency and cost aware offloading of functions from serverless platforms," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 879–896, 2023.

[54] G. Wan, S. Liu, F. Bronzino, N. Feamster, and Z. Durumeric, "{CATO}:{End-to-End} optimization of {ML-Based} traffic analysis pipelines," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pp. 1523–1540, 2025.

[55] Z. Wu, Y. Deng, Y. Zhou, L. Cui, and X. Qin, "Hashcache: Accelerating serverless computing by skipping duplicated function execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 12, pp. 3192–3206, 2023.

[56] D. Zhang, Y. Luo, Y. Wang, X. Kui, and J. Ren, "Batopt: Optimizing gpu-based deep learning inference using dynamic batch processing," *IEEE Transactions on Cloud Computing*, vol. 12, no. 1, pp. 174–185, 2024.

[57] M. Liu and X. Tang, "Dynamic bin packing with predictions," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 3, pp. 1–24, 2022.

[58] K. Lei, Y. Jin, M. Zhai, K. Huang, H. Ye, and J. Zhai, "{PUZZLE}: Efficiently aligning large language models through {Light-Weight} context switch," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 127–140, 2024.

[59] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang, "Cacheblend: Fast large language model serving for rag with cached knowledge fusion," in *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 94–109, 2025.

[60] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen, "No provisioned concurrency: Fast {RDMA-codesigned} remote fork for serverless computing," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 497–517, 2023.

[61] P. Kraft, D. Kang, D. Narayanan, S. Palkar, P. Bailis, and M. Zaharia, "Willump: A statistically-aware end-to-end optimizer for machine learning inference," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 147–159, 2020.

[62] K.-H. Chow, U. Deshpande, S. Seshadri, and L. Liu, "Deeprest: deep resource estimation for interactive microservices," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 181–198, 2022.

[63] J. Wen, Z. Chen, F. Sarro, and S. Wang, "Unveiling overlooked performance variance in serverless computing," *Empirical Software Engineering*, vol. 30, no. 2, p. 59, 2025.

[64] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Springer, 2004.

[65] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[66] J. Kelley Jr, "Critical-path planning and scheduling: Mathematical basis," *Operations Research*, vol. 10, pp. 912–915, 1962.

[67] M. Rahman, S. Venugopal, and R. Buyya, "A dynamic critical path algorithm for scheduling scientific workflow applications on global grids," in *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pp. 35–42, IEEE, 2007.

[68] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

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

[70] V. Sanh, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter.," in *Proceedings of Thirty-third Conference on Neural Information Processing Systems (NIPS2019)*, 2019.

[71] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, vol. 1, pp. 4171–4186, 2019.

[72] L. Zhuang, L. Wayne, S. Ya, and Z. Jun, "A robustly optimized BERT pre-training approach with post-training," in *Proceedings of the 20th Chinese National Conference on Computational Linguistics* (S. Li, M. Sun, Y. Liu, H. Wu, K. Liu, W. Che, S. He, and G. Rao, eds.), (Huhhot, China), pp. 1218–1227, Aug. 2021.

[73] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2021.

[74] X. Hui, Y. Xu, and X. Shen, "Exploring function granularity for serverless machine learning application with gpu sharing," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 9, no. 1, pp. 1–28, 2025.

Appendix A

Chapter 4 Supplement

Serverless workflows may include multiple ML inference functions, potentially solving heterogeneous tasks (e.g., image classification, object detection, NLP). Consequently, there is no universal, task-agnostic definition of "end-to-end accuracy" that is comparable across all applications. Instead of imposing a one-size-fits-all metric, OptiServe exposes an *application-defined end-to-end accuracy score* (EAS) that acts as a configurable quality proxy for optimization and constraint checking.

Per-function accuracy and normalization. For each ML inference function $f \in V$, we assume a finite set of model variants (or configurations) with task-specific evaluation results obtained offline on a fixed validation dataset and metric appropriate for that task (e.g., top-1 accuracy, mAP, F1). We denote the resulting (raw) quality of a chosen variant for f by $\tilde{A}(f)$. Since different functions may use different metrics and ranges, we normalize $\tilde{A}(f)$ to $A(f) \in [0, 1]$ on a per-function basis (e.g., min–max normalization across the variants available for f). OptiServe uses only the normalized values $A(f)$ in the optimization.

Accuracy aggregation as an application interface. Let $\mathcal{F}_{ML} \subseteq V$ be the set of ML inference functions in the workflow. We define the end-to-end accuracy score as

$$EAS(G_s) \triangleq \Phi\Big(\{A(f)\}_{f \in \mathcal{F}_{ML}}\Big), \tag{A.1}$$

where $\Phi(\cdot)$ is a user-provided aggregation function that reflects application semantics (e.g., which stages are more important, whether errors compound, etc.). OptiServe requires Φ to satisfy the following *contract*:

- **Monotonicity:** $EAS(G_s)$ is non-decreasing in each $A(f)$. Upgrading a model variant for any ML function cannot reduce the workflow score.
- **Boundedness/Normalization:** $EAS(G_s)$ must be mapped to a bounded range (e.g., $[0, 1]$) to enable meaningful constraints such as $EAS(G_s) \geq EASC$.
- **Task consistency:** each $A(f)$ is computed using a fixed validation set and metric per task to ensure comparable variant ordering within the same function.

These requirements are sufficient for feasibility checks and for the greedy heuristics used by OptiServe, while avoiding assumptions about cross-task error correlations that are application-specific.

Recommended templates. OptiServe supports any Φ satisfying the above contract. In practice, we recommend the following common templates:

$$\textbf{Weighted sum: } EAS(G_s) = \sum_{f \in \mathcal{F}_{ML}} w_f \cdot A(f), \quad \sum_f w_f = 1,\ w_f \geq 0; \tag{A.2}$$

$$\textbf{Bottleneck (min): } EAS(G_s) = \min_{f \in \mathcal{F}_{ML}} A(f); \tag{A.3}$$

$$\textbf{Geometric mean: } EAS(G_s) = \prod_{f \in \mathcal{F}_{ML}} A(f)^{w_f}, \quad \sum_f w_f = 1,\ w_f \geq 0. \tag{A.4}$$

The weighted-sum template is useful when stages have different importance, the bottleneck template models pipelines dominated by the weakest stage, and the geometric mean captures compounding correctness across stages.

Example. For a workflow with ML functions f_1–f_4, an instance of the weighted-sum template is:

$$EAS(G_s) = w_1 A(f_1) + w_2 A(f_2) + w_3 A(f_3) + w_4 A(f_4), \tag{A.5}$$

where weights $\{w_i\}$ encode application priorities (e.g., upstream gating stages may be assigned larger weights). The accuracy constraint $EAS(G_s) \geq EASC$ thus enforces a minimum acceptable quality level under the selected proxy.