PAMBA: Partition-Aware and Multi-SLO Batching for Serverless Inference on Heterogeneous Clouds

Alireza Abedini,

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

February, 2026

Abstract

Serverless computing offers elasticity and fine-grained billing for machine learning inference, but efficiently supporting large models under diverse latency service-level objectives (SLOs) remains challenging. In particular, existing approaches face a wide cost–performance gap between CPU and GPU execution, while batching and resource selection become increasingly complex under heterogeneous workloads and multiple SLOs. This thesis presents PAMBA, a partition-aware and multi-SLO batching system for serverless inference on heterogeneous clouds. PAMBA combines multi-SLO batching with analytical latency and cost models for CPU, GPU, and partitioned execution, enabling consistent provisioning decisions across different execution modes. To bridge the CPU–GPU gap, the system employs a customized partitioning strategy derived from latency-optimal partitioning, adapted to satisfy serverless resource constraints and jointly consider latency feasibility and per-request cost. This adaptation allows partitioned execution to emerge as an effective intermediate regime between monolithic CPU and GPU deployments. By jointly optimizing execution mode selection, batching, and resource allocation, PAMBA enables flexible inference deployment across a wide range of SLOs and arrival rates, including scenarios where GPU resources are unavailable or inefficiently utilized. Experimental results on convolutional neural networks demonstrate that PAMBA identifies distinct execution frontiers and reduces inference cost compared to existing serverless batching techniques, while maintaining SLO feasibility across heterogeneous workloads.

Acknowledgements

I would like to extend my deepest gratitude to my supervisor, Dr. Hamzeh Khazaei, for his unwavering support and guidance. His invaluable advice and guidance were instrumental in the success of my graduate studies.

I also want to thank my friends and colleagues from the Performant and Available Computing Systems (PACS) Lab. Their support has been an important part of this journey, greatly contributing to my research and personal growth.

A heartfelt thank you to my parents for always loving me and supporting me. Without them, none of this would be possible.

Table of Contents

List of Tables

List of Figures

Declaration of Authorship

I, Alireza Abedini, hereby declare that this thesis titled, "PAMBA: Partition-Aware and Multi-SLO Batching for Serverless Inference on Heterogeneous Clouds", 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.

Chapter 1

Introduction

Serverless computing has gained attention due to its elasticity, automatic resource management, and pay-as-you-go pricing model, allowing applications to scale without explicit capacity planning [1]. Major providers offer serverless platforms that simplify deployment and operations for latency-sensitive services [2]. Commercial serverless offerings include AWS Lambda, Google Cloud Functions, Azure Functions, Alibaba Cloud Function Compute, and IBM Cloud Code Engine, among others [3, 4, 5, 6, 7]. Recent surveys and workload studies further characterize the state of serverless applications and the diversity of their performance and cost behaviours in production [8, 9, 10]. Industry reports similarly highlight the widespread adoption of serverless across organizations and workloads [11]. Provider documentation also specifies how usage is metered and billed (e.g., request-count and compute-time components) under pay-as-you-go pricing [12, 13, 14, 15].

With the rise of machine learning (ML) applications, serverless platforms are increasingly used for inference workloads [16]. Inference is typically stateless and

request-driven, making it a natural fit for event-triggered execution and rapid scaling [17, 18]. More broadly, model serving systems emphasize low-latency prediction under dynamic demand, motivating research on serving-time optimizations across platforms [19]. Prior systems improve serverless inference efficiency via batching and resource tuning, adapting to workload variation using profiling and performance models [20, 21, 22].

Despite these advances, cost-efficient inference with latency service-level objectives (SLOs) remains challenging in practice because serverless deployments often face a *coarse-grained* choice of execution resources. CPU functions are inexpensive but may fail to meet tighter latency targets for modern deep models, while GPU-backed execution (when available) is commonly exposed in large units (e.g., an entire GPU card) and cannot be scaled down to match modest demand. This is particularly problematic for multi-application services where each application may have low request rates and heterogeneous SLOs: even with batching, GPU utilization can be insufficient to amortize cost, while monolithic CPU execution can be too slow. Prior work on serverless GPU inference and GPU sharing illustrates both the performance opportunities and the utilization challenges that arise under platform constraints [23, 24, 25]. These factors create a *missing middle* between CPU and GPU execution under vendor-imposed resource granularity.

Batching addresses part of this problem by amortizing overhead and improving utilization. Prior serverless inference systems dynamically batch and tune resources to reduce cost while meeting SLO constraints (e.g., [20, 26, 27]). However, batching alone cannot always bridge the CPU–GPU gap: monolithic CPU execution may remain infeasible for tight SLOs, while GPU execution can still

be cost-inefficient under modest demand due to coarse-grained allocation. Recent work further explores serverless inference for larger model families such as large language models, reinforcing the importance of cost-efficient, SLO-aware serving under constrained execution environments [28].

A complementary approach is *partitioned execution*, which splits a model into stages executed across multiple serverless functions [29, 30, 31]. Partitioning can reduce per-function memory and compute requirements, enabling deployment of larger models under serverless resource limits and making the execution plan compatible with environments where only smaller execution units are available. Yet existing partitioning systems typically optimize for latency or for a single SLO setting, and do not integrate partitioned execution as a first-class option within a multi-application, multi-SLO batching and provisioning pipeline. As a result, simply combining existing batching/provisioning with partitioned inference is insufficient: execution-mode choice, batching slack, and partition feasibility are *coupled* under serverless billing and platform constraints, and must be optimized jointly. Moreover, prior batching/provisioning and partitioning lines of work typically do not provide an apples-to-apples feasibility and cost comparison across monolithic CPU, GPU, and partitioned CPU under a unified set of constraints.

We present *PAMBA*, a serverless inference optimizer that treats partitioned CPU execution as a first-class mode alongside monolithic CPU and GPU. PAMBA (i) generates a reusable partition template offline by building on latency-optimal partitioning and selecting templates using a cost-aware criterion under serverless feasibility constraints, and (ii) performs online optimization over execution mode, batching, and resource provisioning to satisfy application SLOs at low cost under

platform constraints. We implement PAMBA on Alibaba Cloud Function Compute [6] and evaluate it against HarmonyBatch [27] and Gillis's latency-optimal partitioning [29]. Across multi-application workloads for WRN50-4 and WRN50-5[32], PAMBA reduces end-to-end average cost per request compared to HarmonyBatch, and reduces the system-wide average cost across the two models in the settings where the optimizer can choose among CPU, partitioned, and GPU execution. The main contributions of this thesis are:

- **Unified optimizer:** An end-to-end serverless inference optimizer that jointly selects execution mode (CPU/GPU/partitioned), batching under heterogeneous SLOs, and resource provisioning under platform feasibility constraints.
- **Cost-aware partition template selection:** An offline partition-template generation method that builds on latency-optimal partitioning while selecting templates using a cost-aware criterion relative to a monolithic CPU baseline.
- **System implementation and evaluation:** An implementation on Alibaba Function Compute and an evaluation against HarmonyBatch and Gillis-style latency-optimal partitioning on multi-application workloads.

1.1 Overview

The goal of this thesis is to design a practical optimization framework for serverless ML inference that can meet tail-latency SLOs while minimizing monetary cost under realistic platform constraints. This setting is motivated by (i) the skew and variability of serverless demand across applications [8, 9, 10], and (ii) the

discrete and provider-imposed nature of serverless resource configurations and billing, which shape both feasibility and cost [1, 33].

PAMBA expands the optimizer's decision space by explicitly considering three execution modes—monolithic CPU, monolithic GPU, and partitioned CPU—and selecting among them based on feasibility, SLO compliance, and cost. This choice is coupled with multi-SLO batching and resource provisioning decisions to reduce cost while maintaining SLO adherence [21, 27]. The approach comprises two components: (i) an *offline* phase that generates a tractable set of partition templates suitable for serverless execution, and (ii) an *online* optimizer that selects execution mode, batching parameters, and resource configurations for a given workload.

1.2 Motivation

Coarse-grained resource choices create a CPU–GPU gap. In serverless inference, CPU-only execution can be cost-effective but may be too slow for tight SLOs, while GPU-backed execution can satisfy latency targets yet become cost-inefficient at modest load due to coarse-grained allocation [20, 25, 27]. This tension is a primary contributor to the "missing middle" regime.

Multi-application and multi-SLO serving couples batching and provisioning decisions. In multi-application inference, heterogeneous SLOs constrain how requests can be grouped, and provisioning decisions influence both service time and feasible batching windows [21, 27]. Skewed serverless demand and low per-application request rates further motivate consolidation opportunities across applications when supported by the platform [9, 10].

Partitioned execution expands feasibility but introduces new trade-offs. Partitioning reduces per-function resource requirements and expands feasibility under serverless limits [29, 30, 31]. However, partitioning changes the cost and latency structure by introducing multiple invocations and coordination overheads that must be evaluated under the same feasibility and billing constraints as CPU and GPU execution.

Unified comparison across modes is necessary for practical decisions. Batching, provisioning, and mode selection interact under serverless billing and feasibility constraints. A unified optimization framework is therefore needed to compare monolithic CPU, GPU, and partitioned CPU fairly under consistent assumptions [20, 21, 27, 29].

1.3 Thesis Contributions

This thesis makes the following contributions:

- **Unified optimizer for heterogeneous serverless inference.** An end-to-end optimizer that jointly selects execution mode (CPU/GPU/partitioned CPU), batching under heterogeneous SLOs, and resource provisioning under platform feasibility constraints.

- **Cost-aware partition template selection for serverless constraints.** An offline partition-template generation method that builds on latency-optimal partitioning and selects templates using a cost-aware criterion relative to a monolithic CPU baseline.

- **Implementation and experimental validation on a production serverless platform.** An implementation on Alibaba Cloud Function Compute and an evaluation against a multi-SLO batching baseline and a latency-optimal partitioning method.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background on serverless inference and explains why existing approaches are insufficient for cost-efficient, SLO-aware serving under coarse-grained resource configurations. Chapter 3 reviews related work in batching and multi-tenant serving, model partitioning, GPU provisioning, and serverless performance modeling. Chapter 4 presents the design of PAMBA. Chapter 5 evaluates PAMBA on a production serverless platform. Finally, Chapter 6 concludes the thesis by summarizing the main findings and discussing limitations and directions for future work.

Chapter 2

Background

This chapter introduces background concepts needed to motivate and contextualize the design of PAMBA. We focus on aspects of serverless computing and ML inference that directly affect cost and tail latency, including (i) discrete resource configurations and pay-per-use billing, (ii) latency SLOs for inference, (iii) batching and consolidation in multi-application settings, (iv) heterogeneous CPU/GPU execution, and (v) partitioned execution under serverless constraints. We then summarize the limitations of existing approaches that motivate unified optimization across execution modes, batching, and provisioning. A detailed survey of prior work is provided in Chapter 3.

2.1 Serverless Computing Model and Execution Constraints

Serverless computing abstracts infrastructure management by executing user-defined functions in a managed environment with automatic scaling and pay-per-use pricing [1]. In Function-as-a-Service (FaaS), users deploy functions under discrete resource configurations while the platform handles invocation, scaling, and isolation across tenants [1, 2]. Surveys and workload studies show that serverless applications vary widely in structure and demand, and that performance and cost behaviors can be shaped by platform policies and variability [8, 9, 10]. Public FaaS offerings expose discrete configuration knobs and published pricing/billing rules (e.g., how requests and execution time are charged), which shape both feasibility and cost when selecting configurations [12, 13, 14, 15].

These characteristics make resource configuration and cost prediction non-trivial. Prior work develops analytical and regression-based modeling approaches for serverless performance/cost trade-offs under discrete configuration choices [33], and proposes SLO-aware configuration strategies for workflows and individual functions [34, 35].

2.2 ML Inference on Serverless Platforms

Serverless platforms are increasingly used for ML inference workloads [16]. Inference is typically stateless and request-driven, which aligns well with event-triggered execution and rapid scaling [17, 18]. At the same time, inference often serves latency-sensitive applications where user experience depends on tail latency

rather than only average latency [20, 21]. Recent work explores serverless inference for larger model families (e.g., large language models), further motivating SLO-aware, cost-efficient serving under constrained execution environments [28].

Achieving cost-efficiency typically requires careful selection of execution mode and configuration. Prior work studies how to enable cost-effective, SLO-aware inference on public clouds [22], and how model serving systems should be designed for low-latency prediction under fluctuating demand [19]. In serverless settings, these goals must be achieved under discrete configurations, platform constraints, and fine-grained billing.

2.3 Latency SLOs and Tail Latency

A service-level objective (SLO) specifies a target level of performance that a service aims to maintain. For inference workloads, SLOs are commonly expressed in terms of response time constraints, and system design frequently targets tail metrics (e.g., p99) to capture user experience under variability [20, 27, 36]. In serverless environments, tail behavior can be influenced by configuration choices and runtime effects such as contention and startup overheads, motivating explicitly SLO-aware configuration and tuning strategies [34, 35, 37]. In more controlled serving environments, systems such as Clockwork emphasize predictable tail latency through carefully managed scheduling and resource control, providing a useful contrast to the variability sources common in multi-tenant serverless platforms [38].

Systems research also studies mechanisms to reduce startup-related overheads

that can affect tail latency, including snapshot-based and loading/demand mechanisms [39, 40, 41], as well as inference-specific warming and transformation techniques [42]. PAMBA targets steady-state cost and tail-SLO feasibility under the supported execution modes, while the broader literature highlights complementary directions for mitigating overhead sources that influence tail behavior.

2.4 Batching and Consolidation in Multi-Application Serving

Batching is a widely used lever for improving utilization and amortizing per-invocation overheads in serverless inference [20]. Batching optimizers such as MBS model the trade-off between batching delay and service time, adapting decisions under workload variation [21]. In multi-application settings, the problem becomes more complex: applications may have heterogeneous SLOs, and the system must decide how to group requests while preserving SLO compliance [27].

Large-scale serverless workload analyses highlight skewed arrival patterns and low per-application demand [9, 10], which further motivates consolidation mechanisms when supported by platform isolation and billing. However, multi-SLO batching and provisioning decisions remain constrained by the available execution resources and their granularity (CPU vs GPU), and by platform feasibility constraints.

2.5 Heterogeneous Execution Resources and the CPU–GPU Gap

Heterogeneous execution resources introduce distinct utilization and cost challenges in serverless environments. Early work demonstrated the feasibility of GPU-backed serverless execution [23]. Subsequent systems explored GPU sharing and interference-aware provisioning to improve cost efficiency and predictability [24, 25, 43]. Other work considers SLO-driven inference on heterogeneous accelerators and studies accelerator-side optimization knobs such as DVFS and operator-level decisions [44, 45].

In practice, coarse GPU allocation can make GPU execution cost-inefficient at modest per-application demand [25, 27]. Conversely, CPU execution may be inexpensive but fail to meet tight latency objectives for modern deep models, contributing to a "missing middle" regime between CPU and GPU execution. This regime motivates optimizers that can consider additional execution options (e.g., partitioned CPU) and compare modes under consistent feasibility and billing assumptions.

2.6 Partitioned Execution and Pipeline-Style Inference

Partitioned execution splits a model into stages executed across multiple functions [29, 30, 31]. This approach can reduce per-function memory and compute requirements, enabling deployment under serverless resource limits [29]. Pipeline-oriented

mechanisms can further reduce end-to-end overheads related to coordination and startup effects, such as pipelining cold-start components or separating control and data paths in accelerator-based serving [46, 47].

A practical concept in partitioned execution is the *partition template*: an offline decision that specifies how a model is segmented into stages and mapped onto functions. The template influences latency and cost because it affects the number of stages, the degree of parallelism, and the number of function invocations required to serve a request [29, 30, 31]. Selecting a template must therefore be evaluated within the same feasibility and billing model used to compare CPU and GPU execution plans.

2.7 Limitations of Existing Approaches

This section summarizes limitations of existing approaches that motivate the unified optimization framework developed in this thesis.

Batching-only approaches do not resolve the CPU–GPU gap under coarse-grained accelerators. Batching improves utilization and can reduce cost while meeting SLO constraints [20, 21, 26]. However, batching alone cannot always bridge the practical gap between CPU and GPU execution: CPU may remain infeasible under tight SLOs, while GPU execution may remain cost-inefficient at modest demand due to coarse-grained allocation [25, 27].

Multi-SLO serving couples batching slack, provisioning choices, and feasible execution modes. In multi-application inference, heterogeneous SLOs

constrain how requests can be grouped, and provisioning decisions influence both service time and feasible batching windows [21, 27]. Workload skew and low per-application demand further motivate consolidation opportunities when supported by the platform [9, 10].

Partitioning-only approaches do not provide end-to-end optimization across modes and batching. Partitioned execution expands feasibility under serverless limits [29, 30, 31], but existing partitioning systems typically focus on latency optimization or a single SLO setting. They do not integrate partitioned execution as a first-class option within a multi-SLO batching and provisioning pipeline, leaving the coupled interaction between mode choice, batching slack, partition feasibility, and billing constraints unresolved.

Alternative directions (hybrid execution and pipeline optimization) are complementary. Related work explores hybrid execution and offloading outside the serverless platform [48, 49], and broader inference/pipeline optimization strategies [50]. These directions are complementary to PAMBA: they highlight additional levers for cost-latency improvement, but do not replace the need for unified, mode-aware optimization within serverless inference when execution must occur under platform constraints.

2.8 Summary

This chapter provided background on serverless inference and explained why existing approaches can be insufficient in the presence of coarse-grained execution resources, heterogeneous SLOs, and partition feasibility constraints. Chapter 3

reviews related work in greater depth, and Chapter 4 builds on these foundations to present the design of PAMBA.

Chapter 3

Related Work

Serverless computing has gained traction as a cost-efficient and elastic substrate for ML inference, but achieving predictable tail latency under fine-grained billing remains challenging. Prior work spans batching and multi-tenant serving, automated model partitioning and pipeline-style execution, GPU provisioning and sharing, and profiling-driven performance/cost modeling for configuration selection. Community surveys and workload studies characterize the evolving state of serverless applications and show that demand is often highly skewed and low per application, motivating cross-application consolidation mechanisms (e.g., batching across applications) when supported by the platform's isolation and billing model [8, 9, 10, 11].

3.1 Batching and Cost-Efficient Serverless Inference

Batching is a widely used lever for improving utilization and amortizing invocation overheads in serverless inference. BATCH [20] and follow-up batching optimizers such as MBS [21] model latency and adapt batching decisions under workload variation. HarmonyBatch [27] extends batching to multi-application settings by grouping requests with different SLOs and jointly selecting batching and provisioning decisions across CPU and GPU functions.

Other systems explore complementary cost-aware orchestration and serving-time optimization (e.g., FaaSBatch [26], Cocktail [51], and Harpagon [52]), as well as serverless-native inference designs such as INFless [53]. Related systems also study adaptive inference/model selection under cost and latency constraints (e.g., INFaaS [17]). AsyFunc explores asymmetric function designs for serverless inference, illustrating another approach to improve performance and resource efficiency under FaaS constraints [54]. Earlier work in microservice execution frameworks, such as GrandSLAM, studies mechanisms for meeting SLA constraints under shared resources, which conceptually motivates later SLO-aware scheduling and consolidation directions [55]. In the broader serving literature, model serving systems emphasize low-latency prediction under dynamic workloads and motivate cost-aware runtime decisions [19, 22]. Early work examined feasibility and limitations of deploying deep models on serverless platforms [18, 56].

3.2 Model Partitioning and Pipeline-Style Execution

To address serverless limits on memory and execution time, several works partition models into stages executed across multiple functions. Gillis [29] proposes a latency-optimal partitioning method and an SLO-aware cost-oriented mode, illustrating how offline partition templates and coordination mechanisms can enable scalable serving of large networks. Outside serverless, systems such as AlpaServe use model parallelism to enable statistical multiplexing of large models in cluster serving, highlighting partitioning as a lever for improving serving efficiency under bursty demand [57]. Subsequent work explores alternative formulations for cost/latency trade-offs under serverless billing, including AMPS-Inf [30] and MOPAR [31].

Related pipeline-oriented systems mitigate overheads around function coordination and startup effects. PipeCo [46] pipelines cold-start components for inference services, and SplitRPC [47] separates control and data paths to reduce overheads when accelerators are involved. Complementary work reduces startup overheads more generally via snapshotting and demand/loading techniques [39, 40, 41], which can improve tail behavior in systems where cold starts occur frequently.

3.3 GPU Provisioning and Heterogeneous Acceleration

GPU support in serverless and cloud inference introduces additional challenges in utilization, interference, and cost efficiency. Early frameworks enabled GPU-backed serverless execution [23]. More recent systems study spatio-temporal GPU sharing (FaST-GShare [24]) and interference-aware provisioning for predictable inference (iGniter [43]). In cluster settings, Nexus demonstrates fine-grained GPU scheduling across multiple DNN workloads to improve utilization while meeting latency goals, providing a reference point for what can be achieved when scheduling control is available [58]. Complementary approaches optimize inference under heterogeneous accelerators and SLO constraints [44], and explore accelerator-side knobs such as DVFS and operator-level decisions [45]. Work on function granularity and GPU sharing further highlights the practical challenges of matching accelerator allocations to workload demand in serverless environments [25]. Recent systems also explore serverless inference for large language models, which reinforces the importance of cost-efficient serving when models and memory footprints increase [28]. Related cluster serving systems for transformer-based generative models, such as Orca, further illustrate how scheduling and batching strategies can be tailored to large-model workloads [59]. GPU batching optimizers outside the serverless setting provide additional background on how batching interacts with accelerator utilization [60].

3.4 Performance Modeling, Configuration Tuning, and Workflow Optimizations

Accurate latency/cost modeling and lightweight profiling are central to selecting serverless configurations. Prior work develops analytical or regression-based models for performance/cost trade-offs [33, 61, 62, 63] and SLO-aware configuration strategies for workflows and individual functions [34, 35]. Related work also considers predicting or recommending function sizes directly from observed behavior and configuration choices (e.g., Sizeless) [64]. Several systems reduce practical overheads that strongly affect end-to-end efficiency, including warming and transformation mechanisms for serverless ML inference [42], and interference- and startup-aware configuration selection [37]. At the workflow level, serverless DAG optimizers size, bundle, reorder, and prewarm nodes to reduce critical-path latency [65, 66, 67]. Reducing cold-start and redundant computation has also been studied via mechanisms such as remote container forking and demand loading [40, 41], as well as caching/deduplication and memory-efficiency techniques for serverless execution [68, 69, 70]. Broader pipeline/inference optimization systems also motivate cost-aware, end-to-end reasoning about execution plans [50].

3.5 Positioning

Our work builds on these directions by incorporating multi-SLO batching ideas [27] while treating partitioned execution as a first-class mode alongside CPU and GPU. In contrast to systems that focus solely on batching or solely on partitioning, we

study unified end-to-end optimization over execution mode, provisioning, and consolidation decisions under a production serverless platform's feasibility and billing constraints, using offline partition-template generation informed by profiling and online selection within the optimizer's search space.

Chapter 4

Methodology: Partition-Awareness and Multi-SLO Batching

This chapter presents the design of *PAMBA*, our partition-aware and multi-SLO batching framework for cost-efficient serverless inference under latency constraints. The key challenge is to jointly optimize *execution mode* and *request consolidation* under practical serverless platform constraints: monolithic CPU execution may be inexpensive but infeasible for tight SLOs, GPU execution may satisfy latency targets yet be cost-inefficient at modest demand due to coarse-grained allocation, and partitioned execution can expand feasibility but introduces additional invocations and coordination overheads. PAMBA addresses this challenge by treating partitioned CPU execution as a first-class serving option alongside monolithic CPU and GPU, and by integrating it directly into a unified optimization pipeline that accounts for heterogeneous SLOs, feasibility constraints, and pay-per-use billing.

At a high level, PAMBA combines two complementary components. First, an *offline* phase generates a compact set of reusable partition templates by building on latency-optimal partitioning and selecting templates using a cost-aware criterion under serverless feasibility constraints. Second, an *online* optimizer jointly selects the execution mode (CPU/GPU/partitioned CPU), batching and consolidation decisions across applications with different SLOs, and the corresponding resource configurations that minimize cost while meeting tail-latency objectives. The remainder of this chapter details PAMBA's system model, design choices, and algorithms, and explains how the offline template generation and online optimization interact to enable practical, SLO-aware, cost-efficient serverless inference.

4.1 System Architecture and Modelling

In this section, we describe the architecture of our serverless inference system and introduce the modelling foundations that drive execution-mode selection, resource allocation, batching, and grouping decisions. Our system supports three execution paths: CPU, GPU, and Partitioned inference under a unified optimizer that evaluates feasibility and cost for multi-tenant workloads. We first present a high-level overview of the architecture, then describe the execution modes, followed by detailed latency and cost models.

4.1.1 System Overview

Our system extends a conventional serverless inference pipeline by supporting heterogeneous execution on CPU functions, GPU functions, and multi-stage partitioned CPU functions. Figure 4.1 illustrates the architecture. Each application

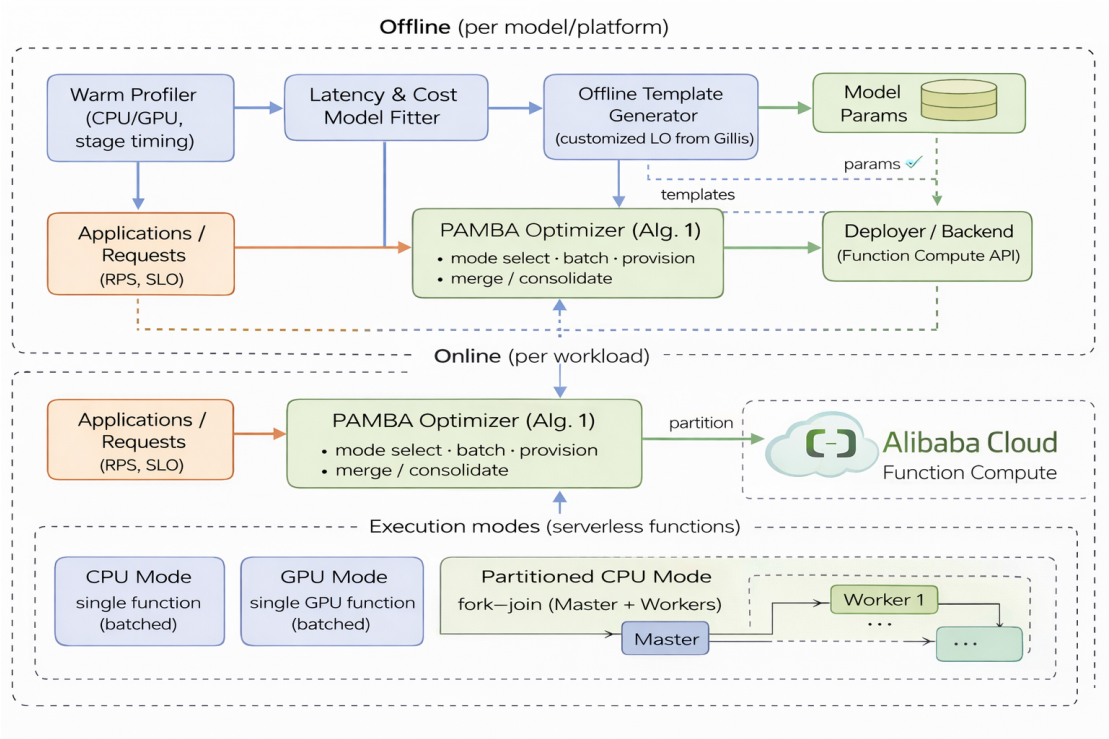


FIGURE 4.1: Overview of PAMBA. Offline profiling fits latency and cost models and produces partition templates, and the online optimizer selects an execution mode and configuration for each group.

is described by its request rate (RPS) and service-level objective (SLO). Using offline profiling artifacts (latency/cost models and partition templates), the Optimizer searches feasible configurations across all execution modes and outputs a deployment plan, execution mode, resource configuration, batch size, and any consolidation decisions. The Serverless Backend (e.g., Alibaba Function Compute) provisions and executes the selected configuration.

4.1.2 Execution Modes

We support three execution modes, each occupying a distinct point in the performance–cost design space.

CPU Mode

The entire neural network executes within a single CPU-based serverless function. This mode is suitable for smaller models or applications with relaxed SLOs. Latency follows a profile-driven exponential model, and cost scales with allocated vCPUs and execution time.

GPU Mode

In GPU mode, the full model executes on a GPU-backed serverless function. Latency is batch-dependent and derived from profiling curves.

Partitioned Mode

Partitioned mode executes the model across multiple CPU workers organized into stages. Each stage processes a subset of neural network layers, passing the output forward after the master's aggregation per stage. Rather than optimizing a single SLO-specific partitioning policy online, PAMBA computes a reusable partition *template* offline once per model (Section 4.2.1) that optimizes both cost and latency. The online optimizer then treats partitioned execution as a first-class mode alongside CPU and GPU, and tunes per-stage resources and batch sizes for the fixed template (Algorithm 2).

4.1.3 Latency Models

Latency prediction is central to determining SLO feasibility and optimal batching. For each execution mode, we construct profiling-driven models for both the *average* latency and the *tail* latency (p99), as required by our optimizer. For CPU and

GPU modes, we follow prior profiling-driven latency modeling methodology [27], calibrated using our own profiling on Alibaba Function Compute; the partitioned model extends this methodology with stage/worker and communication terms.

CPU Latency Model

The latency of CPU-based inference decreases nonlinearly with allocated vCPU cores. We model average CPU latency using an exponential decay form: :

$$L_c^{\text{avg}}(c) = \alpha_b^{\text{avg}} \cdot \exp\left(-\frac{c}{\beta_b^{\text{avg}}}\right) + \gamma_b^{\text{avg}}, \tag{4.1}$$

where c is the number of allocated vCPU cores, and $\alpha_b^{\text{avg}}, \beta_b^{\text{avg}}, \gamma_b^{\text{avg}}$ are coefficients learned from profiling for model b.

To ensure SLO feasibility for tail behaviour, we also model the p99 latency:

$$L_c^{\text{p99}}(c) = \alpha_b^{\text{p99}} \cdot \exp\left(-\frac{c}{\beta_b^{\text{p99}}}\right) + \gamma_b^{\text{p99}}. \tag{4.2}$$

These two curves enable the optimizer to determine whether CPU execution meets per-application SLO constraints under both mean and tail latency requirements. We fit separate parameters for a small set of batch sizes (e.g., $b \in \{1, 2\}$) using the same functional form.

GPU Latency Model

GPU latency depends primarily on the batch size. Following prior GPU latency modeling for serverless batching [27], we express the baseline (batch-dependent)

latency as:

$$L_g^{\text{avg}}(b) = \xi_1 \cdot b + \xi_2, \tag{4.3}$$

where b is the batch size and (ξ_1, ξ_2) are profiling-derived coefficients for the target GPU (e.g., NVIDIA A10).

However, serverless GPU execution exhibits additional effects due to GPU time-slicing and memory partitioning. HarmonyBatch models these constraints by computing the maximum latency needed under worst-case scheduling:

$$L_g^{\max}(b, m) = \left(\left\lceil \frac{L_g^{\text{avg}}(b)}{m \cdot \tau} \right\rceil \cdot (M_{\max} - m) \cdot \tau \right) + L_g^{\text{avg}}(b), \tag{4.4}$$

where m is the fraction of GPU memory allocated to the function, $M_{\max}$ is the full GPU memory capacity, and τ is the GPU time-slice duration. This expression captures contention-sensitive scheduling latency due to GPU multiplexing.

Analogous to CPU mode, we compute the GPU p99 latency:

$$L_g^{\text{p99}}(b) = \kappa_1 \cdot L_g^{\max}(b, m) + \kappa_2, \tag{4.5}$$

where (κ_1, κ_2) capture tail inflation observed in profiling. The p99 value determines whether a given batch size is admissible under an application's SLO.

Partitioned Inference Latency Model

In partitioned mode, the model is split into S stages, each executed by a group of CPU workers. Let $\mathcal{W}_s$ denote the set of workers assigned to stage $s \in \{1, \ldots, S\}$.

We model each worker's latency using the same exponential structure as CPU mode, but with stage- and worker-specific parameters learned from profiling.

For worker $w \in \mathcal{W}_s$ with allocated vCPU cores $c_{s,w}$, the average latency is

$$L_{s,w}^{\text{avg}}(c_{s,w}) = \alpha_{s,w}^{\text{avg}} \cdot \exp\left(-\frac{c_{s,w}}{\beta_{s,w}^{\text{avg}}}\right) + \gamma_{s,w}^{\text{avg}}, \tag{4.6}$$

and the p99 latency is

$$L_{s,w}^{\text{p99}}(c_{s,w}) = \alpha_{s,w}^{\text{p99}} \cdot \exp\left(-\frac{c_{s,w}}{\beta_{s,w}^{\text{p99}}}\right) + \gamma_{s,w}^{\text{p99}}. \tag{4.7}$$

Within a stage, the master must wait for all workers to complete, so that the stage latency is determined by the slowest worker. The average and p99 latencies for stage s are therefore

$$L_s^{\text{avg}} = \max_{w \in \mathcal{W}_s} L_{s,w}^{\text{avg}}(c_{s,w}), \tag{4.8}$$

$$L_s^{\text{p99}} = \max_{w \in \mathcal{W}_s} L_{s,w}^{\text{p99}}(c_{s,w}). \tag{4.9}$$

The total computation latency of a partitioned pipeline is the sum of per-stage latencies plus inter-stage communication overhead. We model communication using a fixed per-stage transition cost θ_0 and a batch-dependent term $\theta_1 b$, where b is the batch size. The end-to-end average latency of partitioned inference is

$$L_{\text{part}}^{\text{avg}}(b) = \sum_{s=1}^{S} L_s^{\text{avg}} + (S-1)\left(\theta_0 + \theta_1 b\right), \tag{4.10}$$

and similarly, the p99 latency is

$$L_{\text{part}}^{\text{p99}}(b) = \sum_{s=1}^{S} L_s^{\text{p99}} + (S-1)\left(\theta_0 + \theta_1 b\right). \tag{4.11}$$

Both the worker-level coefficients $\alpha_{s,w}^{(\cdot)}, \beta_{s,w}^{(\cdot)}, \gamma_{s,w}^{(\cdot)}$ and the communication parameters θ_0, θ_1 are obtained from profiling the partitioned model under different resource allocations and batch sizes. This allows the optimizer to evaluate candidate stage–worker topologies and batch sizes using realistic per-stage and end-to-end latency estimates. We check SLO feasibility using end-to-end tail latency $L_{\text{e2e}}^{\text{p99}}(g,b) = W^{\text{p99}}(b) + L_{\text{exec}}^{\text{p99}}(g,b)$ against $S_g = \min_{i \in g} S_i$.

4.1.4 Cost Models

We develop a unified cost model that captures the billing rules of heterogeneous serverless execution on commercial platforms. The model is used by the optimizer to compare CPU, GPU, and partitioned configurations under a consistent cost-per-request formulation.

Resource-Based Billing

For a function instance with c vCPUs, m GB of memory, and g GPU units, the platform charges fixed per-second rates[15]:

$$\lambda_c = 1.6{\times}10^{-5}, \qquad \lambda_m = 2.4{\times}10^{-6}, \qquad \lambda_g = 3.4{\times}10^{-5},$$

together with an invocation charge of $\lambda_{\text{inv}} = 1.2\times10^{-7}$ per request. Let L denote the execution time (in seconds) of a batch of size b. The base cost per request is:

$$C_{\text{base}} = \frac{\lambda_{\text{inv}} + (\lambda_c\, c + \lambda_m\, m + \lambda_g\, g) \cdot L}{b}, \tag{4.12}$$

Here λ_{inv} is the per-*invocation* charge, amortized over a batch of size b; L is billed execution time.

Partitioned Execution Cost

Partitioned inference consists of a master function and multiple workers distributed across stages. Let L_0 be the master's execution time and $L_{s,w}$ the execution time of worker w in stage s. The per-request cost is computed by applying (4.12) independently to the master and each worker:

$$C_{\text{part}} = C_{\text{master}} + \sum_{s=1}^{S} \sum_{w \in \mathcal{W}_s} C_{s,w}. \tag{4.13}$$

This formulation reflects the fact that each stage worker is invoked separately for each batch and billed according to its allocated vCPU and memory resources.

Note: The platform publishes per-invocation and resource-time rates but does not expose per-request bills, so we compute per-request cost from published pricing and the billed execution time of each configuration. Billed duration is rounded up to 100 ms quanta before applying resource-time rates.

4.2 Algorithm Design

This section describes the optimization workflow that maps a set of applications to a cost-minimizing serverless deployment plan under per-application SLO constraints. The optimizer maintains an ordered list of application groups and iteratively applies two core operations: (i) provisioning a group by selecting its lowest-cost SLO-feasible configuration across CPU, GPU, and partitioned execution modes, and (ii) greedily merging adjacent groups when reprovisioning the merged group reduces total cost. Provisioning jointly determines execution mode, resource allocation, and batching parameters using the modelling described in the previous section. The overall workflow proceeds in three phases: initial per-application provisioning, consolidation of non-GPU groups (CPU and partitioned), and consolidation of GPU-involved adjacent pairs. The complete optimization procedure is summarized in Algorithm 1.

Notation. FindBestConfig(g) (defined in Sec. 4.2.3) returns the minimum-cost configuration for group g that satisfies the end-to-end p99 SLO constraint.

4.2.1 Offline Partition Template Generation

Partitioned execution requires a concrete *partition template* that specifies how model layers are grouped into sequential stages. PAMBA computes this template offline once per model; the online pipeline then treats partitioned execution as a first-class mode alongside CPU and GPU, and optimizes per-stage resource sizing and batching for the fixed template (Algorithm 2).

Why LO-style partitioning as a base. Gillis [29] describe two partitioning

strategies: (i) latency-optimal (LO) partitioning and (ii) SLO-aware cost-optimal partitioning targeting a single latency threshold. Our system must support multiple applications with heterogeneous SLOs, so an SLO-specific partitioning policy would not yield a single reusable template. In addition, making partitioning decisions online would add overhead to the provisioning loop. We start from a latency-optimal partitioning procedure and then customize it for serverless deployment by selecting among candidate templates using a cost-aware criterion under feasibility constraints. This yields reusable partition templates that are not only low-latency, but also cost-effective so it outperforms monolithic CPU and GPU in some regimes for the optimizer.

LO partitioning recap [29]. Let a model be a sequence of layers $1..N$. A partition plan chooses cut points $0 = b_0 < b_1 < \cdots < b_S = N$, forming S contiguous stages, where stage s covers layers $(b_{s-1}+1)..b_s$. Following LO-style modeling, we estimate the stage service time as:

$$\ell(b_{s-1}+1, b_s) = t_{\text{cpu}}(b_{s-1}+1, b_s) + t_{\text{xfer}}(b_{s-1}, b_s), \tag{4.14}$$

where $t_{\text{cpu}}(\cdot)$ is the compute time for the stage and $t_{\text{xfer}}(\cdot)$ captures activation transfer overhead across the stage boundary. The LO objective minimizes predicted end-to-end service time subject to feasibility:

$$p_{\text{LO}} = \arg\min_{p \in \mathcal{P}} \sum_{s=1}^{S} \ell(b_{s-1}+1, b_s) \quad \text{s.t. feasible}(p). \tag{4.15}$$

We reuse this LO-style formulation to (i) define candidates over contiguous stages and (ii) enforce feasibility constraints, while modifying how the final template is

selected to incorporate serverless cost (below).

Profiling and feasibility constraints. For the target model, we profile (1) monolithic CPU execution and (2) per-stage execution to obtain t_{cpu} terms, and we profile activation sizes to estimate t_{xfer}. During candidate evaluation, we enforce serverless feasibility constraints, including: (i) per-function memory limits (model weights + activations), (ii) a bounded number of stages $S \leq S_{\max}$ to limit invocation overhead, and (iii) successful execution under the platform runtime limits. These constraints define the feasible set $\mathcal{P}$ in Eq. (4.15).

Cost-aware selection relative to a monolithic CPU baseline. Using LO directly can favor templates that reduce latency but increase per-request cost disproportionately due to additional invocations and transfer overheads. To bias selection toward cost-efficient templates while still using LO-style candidate generation, PAMBA selects the template using a marginal trade-off score relative to the monolithic CPU baseline with the same config.

Let $(L_{\text{cpu}}, C_{\text{cpu}})$ denote the measured (or estimated) latency and per-request cost of executing the full model in a single CPU function under the reference CPU configuration used by our system.

For each feasible candidate template p, let (L_p, C_p) denote its predicted end-to-end latency and per-request cost under the same billing model. Among candidates with a meaningful latency improvement over monolithic CPU, we select:

$$\text{Score}(p) = \frac{L_{\text{cpu}} - L_p}{\max(C_p - C_{\text{cpu}}, \epsilon)}, \tag{4.16}$$

where ϵ is a small constant for numerical stability. We discard candidates with negligible latency improvement ($L_{\text{cpu}} - L_p < \delta$) to avoid selecting templates that change cost without material latency benefit. Intuitively, Eq. (4.16) favors templates that provide larger latency reductions per additional cost compared to a single-function CPU deployment, without targeting any single application SLO.

Output artifact and interface to the online optimizer. This offline step outputs only the stage boundaries $\{b_s\}$ (i.e., which layers belong to each stage). Given this fixed template, the online optimizer evaluates partitioned execution as one mode alongside CPU and GPU, and tunes per-stage resources and batch sizes using Algorithm 2.

4.2.2 Initialization and Threshold Estimation

The optimizer begins by constructing singleton groups, one per application, and sorting them by increasing SLO. For each SLO class, a feasibility threshold is computed to bound subsequent consolidation of CPU and partitioned groups. This threshold restricts candidate merges to those that can remain SLO-feasible after aggregation and reduces the search space explored during optimization.

4.2.3 Group Provisioning and Configuration Selection

Group provisioning is the core decision step of the optimizer and is invoked during initialization as well as after every tentative group merge. Given a group g of applications, provisioning selects an execution mode and a feasible configuration that minimizes cost under the tightest group deadline $S_g = \min_{i \in g} S_i$.

Provisioning jointly determines (i) the execution mode (CPU, GPU, or partitioned), (ii) the associated resource allocation, and (iii) batching parameters where applicable. Candidates are evaluated using the profiling-driven latency and cost models from Section 4.1. A configuration is SLO-feasible if its predicted end-to-end tail latency satisfies $L_{\text{e2e}}^{\text{p99}}(g, b) \leq S_g$; if no candidate is feasible, the group is marked infeasible.

For each group, candidates are evaluated under each execution mode. *CPU:* enumerate feasible vCPU allocations and retain the minimum-cost feasible option. *GPU:* enumerate candidate batch sizes and validate them using a single equivalent timeout T_{eq} derived from the group's $\{(S_i, r_i)\}$; the largest admissible batch size is denoted $b_{\max}$. *Partitioned:* for each batch size, optimize per-stage resources subject to $L_{\text{e2e}}^{\text{p99}} \leq S_g$ using stage-wise ternary search with an SLO penalty, and retain only feasible configurations. Ternary search iteratively narrows a bracketed interval to minimize a unimodal objective over a 1D resource parameter.

Finally, provisioning selects the lowest-cost feasible configuration and associates it with the group; after every accepted merge, provisioning is re-invoked to keep the selected mode, resources, and batching optimal as group composition evolves.

4.2.4 Phase I: Consolidation of Non-GPU Groups

Following initial provisioning, the optimizer performs a consolidation phase over groups provisioned using non-GPU execution modes, namely CPU and partitioned execution. The goal of this phase is to reduce invocation overhead and improve resource efficiency by merging adjacent groups when joint execution yields lower total cost.

Let $L = [g_1, g_2, \ldots, g_n]$ denote the ordered list of groups sorted by increasing SLO. Phase I considers only contiguous segments $[g_i, g_{i+1}, \ldots, g_j]$ such that all groups in the segment are provisioned using non-GPU execution. Candidate segments are further constrained by an SLO-specific feasibility threshold on aggregate arrival rate, which bounds the maximum load under which CPU or partitioned execution can remain SLO-feasible. This threshold prunes infeasible merges early and reduces unnecessary reprovisioning.

For each candidate segment, the optimizer tentatively merges the applications in the segment to form a new group $g^\star = \bigcup_{k=i}^{j} g_k$. The merged group is then reprovisioned using the group provisioning procedure described in Section 4.2.3. Let $C(g^\star)$ denote the cost of the reprovisioned merged group, and let $C_{\text{sep}} = \sum_{k=i}^{j} C(g_k)$ denote the total cost of executing the groups separately. The merge is accepted if and only if

$$C(g^\star) < C_{\text{sep}}$$

and the reprovisioned configuration satisfies all SLO constraints.

Merging proceeds greedily from left to right over the ordered group list. When a merge is accepted, the merged group replaces the original segment in L, and the consolidation process restarts from the beginning of the merged segment. This process repeats until no further cost-reducing merges are possible within any contiguous non-GPU segment.

By restricting consolidation to non-GPU groups in this phase, the optimizer avoids premature GPU allocation while aggressively exploiting the cost benefits of shared execution under CPU and partitioned modes.

4.2.5 Phase II: Consolidation of GPU-Involved Groups

After completing consolidation of non-GPU groups, the optimizer performs a second consolidation phase focused on GPU execution. This phase targets adjacent groups, with the objective of improving GPU utilization and reducing related cost while preserving SLO feasibility.

Let $L = [g_1, g_2, \ldots, g_n]$ denote the ordered list of groups after Phase I. Phase II considers adjacent pairs (g_i, g_{i+1}) such that at least one of the two groups is provisioned using GPU execution. For each candidate pair, the optimizer forms a tentative merged group

$$g^\star = g_i \cup g_{i+1}$$

and reprovisions it using the group provisioning procedure described in Section 4.2.3.

Unlike Phase I, GPU consolidation imposes an additional constraint: the merged group must be provisioned using GPU execution. This restriction prevents merges that eliminate GPU execution entirely and ensures that Phase II specifically targets GPU utilization rather than reverting to CPU or partitioned modes. If the reprovisioned configuration of $g^\star$ does not select GPU execution, the merge is immediately rejected.

Let $C(g_i)$ and $C(g_{i+1})$ denote the costs of executing the two groups separately, and let $C(g^\star)$ denote the cost of executing the merged group under GPU execution. The merge is accepted if

$$C(g^\star) < C(g_i) + C(g_{i+1})$$

#Apps	WRN50-4	WRN50-5
1	2	3
2	8	15
4	18	48
8	36	84

TABLE 4.1: PAMBA optimizer computation time (ms) to generate a provisioning plan as the number of applications increases.

and the reprovisioned configuration satisfies the SLO constraints of all applications in $g^{\star}$.

GPU consolidation proceeds greedily over the ordered group list. When a merge is accepted, the merged group replaces (g_i, g_{i+1}) in L, and the optimizer re-evaluates neighbouring pairs that may be affected by the merge. This process repeats until no further GPU-involved pairwise merges yield a cost reduction.

By isolating GPU consolidation into a separate phase and explicitly accounting for GPU-specific cost behaviour during reprovisioning, the optimizer avoids over-consolidation while effectively amortizing GPU overhead across compatible workloads.

Optimizer overhead. Table 4.1 reports PAMBA's computation time to generate a provisioning plan as the number of applications increases. This overhead is a control-plane cost incurred only when (re)computing a plan (e.g., upon workload or SLO changes), not on the per-request inference path. Across both models, the planning time remains below 0.1 s up to 8 applications, which is practically negligible when amortized over the many requests served under each plan.

Final plan. The optimizer outputs groups and their selected mode/resources/-batching parameters. This plan is consumed by the runtime to provision functions and route requests accordingly.

Algorithm 1 End-to-end Optimization

Input: Applications $\mathcal{A}$ with RPS and SLO
Output: Groups L, configs $\mathcal{F}$, batches $\mathcal{B}$
—– *Phase 1: Initial Provisioning*
Initialize groups $L \leftarrow \{\{a_1\}, \ldots, \{a_n\}\}$
for each $g \in L$ **do**
  $(C_g, f_g, b_g) \leftarrow \textsc{FindBestConfig}(g)$
end for
$L \leftarrow \textsc{SortWithSLO}(L)$
—– Phase 2: CPU/PART Consolidation
$i \leftarrow 0$
while $i < |L|$ **do**
  if $L[i].mode \in \{\text{CPU}, \text{PART}\}$ **then**
    $j \leftarrow i$
    while $j < |L|$ and $L[j].mode \in \{\text{CPU}, \text{PART}\}$ **do**
      $(L, m) \leftarrow \textsc{Merge}(L, i, j+1)$
      $j \leftarrow (m?i : j+1)$
    end while
  end if
  $i \leftarrow i + 1$
end while
—– *Phase 3: GPU Consolidation*
$i \leftarrow 0$
while $i < |L| - 1$ **do**
  if $L[i].mode = \text{GPU}$ or $L[i+1].mode = \text{GPU}$ **then**
    $(L, m) \leftarrow \textsc{Merge}(L, i, i+2)$
    $i \leftarrow (m?i-1 : i+1)$
  else
    $i \leftarrow i + 1$
  end if
end while
return $L, \mathcal{F}, \mathcal{B}$
—– **Merge Procedure**
Procedure $\textsc{Merge}(L, \ell, h)$:
  $\mathcal{X} \leftarrow \bigcup_{k=\ell}^{h-1} L[k]$
  $(C_{\mathcal{X}}, f_{\mathcal{X}}, b_{\mathcal{X}}) \leftarrow \textsc{FindBestConfig}(\mathcal{X})$
  if $C_{\mathcal{X}} < \sum_{k=\ell}^{h-1} C_{L[k]}$ and SLO-feasible **then**
    Replace $L[\ell : h]$ with $\mathcal{X}$
    return (L, True)
  return (L, False)

Algorithm 2 Partitioned Provisioning via Stage-wise Ternary Search

Input: Group applications $\mathcal{X}$ with $\{(S_i, r_i)\}$ and group deadline S_g
Output: Best partitioned configuration or None
Compute equivalent timeout T_{eq} and rate r_{eq} {summarizes heterogeneous (S_i, r_i) as a single batching constraint}
Determine maximum feasible batch size $b_{\max}$
for $b = b_{\max}$ down to 1 **do**
  Initialize per-stage resources $\mathbf{c}$
  for each optimization iteration **do**
    for each stage s **do**
      Define 1D objective $J_s(x)$ with other stages fixed
      $c_s \leftarrow \text{TernarySearch}(J_s)$
    end for
  end for
  Evaluate cost and p99 latency for $(\mathbf{c}, b)$
  if p99 latency $\leq S_g$ **then**
    Update best configuration if cost improves
  end if
end for
return best configuration

Chapter 5

Experimental Validation of PAMBA

This section evaluates the effectiveness of our system in minimizing cost while satisfying application-level SLO constraints. We compare against existing baselines and analyze how execution-mode selection and partitioned inference affect performance across different workload regimes.

5.1 Experimental Setup

We implement PAMBA on Alibaba Cloud Function Compute (FC) using MXNet [71], with both CPU-backed and GPU-backed functions. Unless stated otherwise, all measurements use warm invocations to capture steady-state service time. We evaluate two CNNs: WRN50-5 and WRN50-4.

Configuration space. Our optimizer searches across monolithic CPU, monolithic GPU, and partitioned CPU execution. CPU and partitioned stages are sized

by vCPU allocations supported by FC; GPU execution sweeps feasible batch sizes under the platform's GPU allocation constraints (Section II-C). For partitioned execution, the partition template is generated offline and reused online; unless stated otherwise, the master function is pinned to the minimum feasible configuration, while stage/worker functions are sized by their vCPU allocations.

Workloads. To capture burstiness beyond stationary arrivals, we also evaluate time-varying request rates by replaying trace-driven arrival patterns (normalized and scaled to target mean RPS), following prior serverless inference evaluation practice [27]. We evaluate both single-application sweeps (varying SLO or arrival rate) and multi-application workloads with 1 to 8 concurrent applications. Each configuration is executed for 10 minutes and repeated multiple times; we report the average across runs.

Baselines. We compare PAMBA against HarmonyBatch (HB), which optimizes CPU and GPU execution without partitioned inference, and against the Gillis latency-optimal partition template for the partitioned-template.

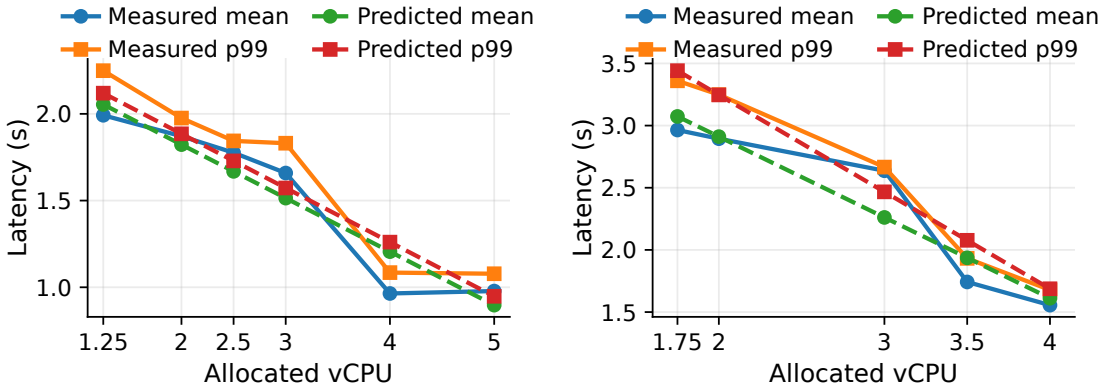
5.2 Latency Model Validation

Our optimizer relies on the profiling-driven latency models in Section 4.1.3 (Eqs. (1)–(11)) to (i) test SLO feasibility and (ii) compare cost across CPU, GPU, and partitioned execution. We therefore validate the accuracy of these models on two representative CNNs (WRN50-4 and WRN50-5) using *warm* runs to isolate steady-state service-time behavior.[1]

[1]We do not explicitly model cold-start behavior; profiling and validation are performed on warm invocations.

For each execution mode, we sweep the key configuration dimension used by the optimizer: allocated CPU vCPUs for monolithic CPU execution, batch size for GPU execution, and per-stage vCPU for partitioned CPU execution under the selected offline partition template (Section III-A). At each sweep point, we compare the measured mean and p99 latency against the model prediction and report mean absolute percentage error (MAPE) across the sweep.

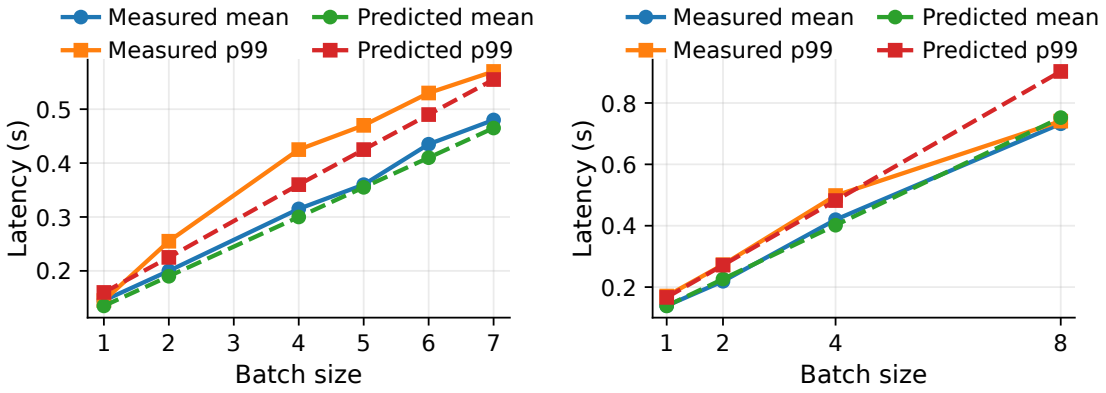
Figures 5.1, 5.2, and 5.3 show that the fitted curves track the measurements closely for both models across all three execution modes. The resulting MAPEs are low (single-digit in all cases for mean latency), which supports using these models inside the provisioning and consolidation loop.



(A) WRN50-4.

(B) WRN50-5.

FIGURE 5.1: Comparison of the observed and predicted inference latency of WRN50-4 and WRN50-5 executed on CPU functions using warm invocations (b=1), with mean (p99) MAPE of 9.00% (7.24%) for WRN50-4 and 6.64% (3.61%) for WRN50-5.



(A) WRN50-4.

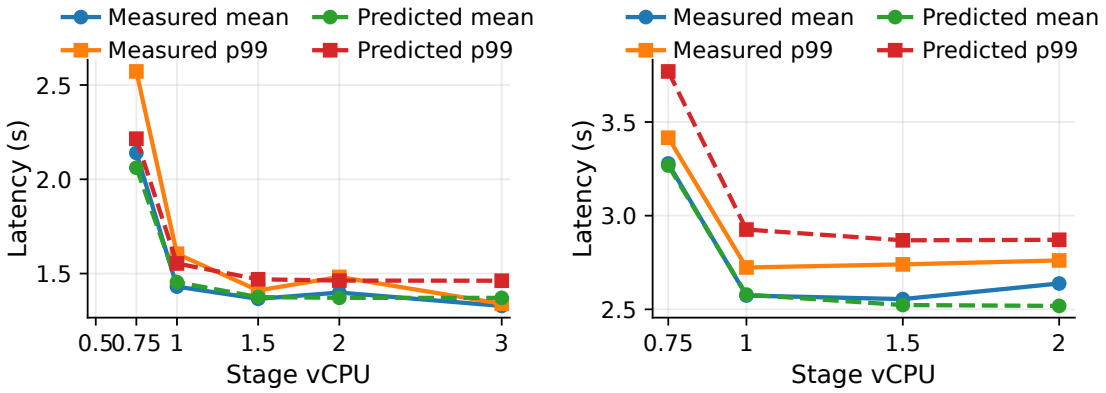
(B) WRN50-5.

FIGURE 5.2: Comparison of the observed and predicted inference latency of WRN50-4 and WRN50-5 executed on GPU functions, with mean (p99) MAPE of 3.90% (9.80%) for WRN50-4 and 2.68% (7.06%) for WRN50-5.

5.3 Single-Application Performance and Mode Frontiers

We study execution-mode selection for a single application by sweeping (i) the latency SLO at a fixed arrival rate, and (ii) the request arrival rate (RPS) at a fixed SLO. These sweeps expose *mode frontiers*—knee points where the cost-optimal provisioning strategy changes across CPU, GPU, and partitioned CPU execution. Figures 5.4 and 5.5 each report both sweeps for a given model (left: SLO sweep, right: RPS sweep).

WRN50-5. In Figure 5.4 (left), tight SLOs select GPU execution because CPU and partitioned CPU are infeasible. As the SLO relaxes, the optimizer crosses a knee point where CPU becomes feasible and reduces cost. With additional slack, a second knee point appears where partitioned CPU becomes cost-optimal, capturing an intermediate regime that avoids GPU over-provisioning while still



(A) WRN50-4.

(B) WRN50-5.

FIGURE 5.3: Comparison of the observed and predicted end-to-end inference latency of partitioned WRN50-4 and WRN50-5 executed on CPU functions using warm invocations (b=1), with mean (p99) MAPE of 2.24% (6.34%) for WRN50-4 and 1.58% (6.62%) for WRN50-5.

meeting the SLO. Overall, WRN50-5 exhibits a clear three-region frontier: GPU under tight SLOs, CPU under moderate SLOs, and partitioned CPU under looser SLOs.

Figure 5.4 (right) fixes the SLO and sweeps RPS. At low arrival rates, partitioned CPU is selected since GPU batching cannot sufficiently amortize GPU cost. As RPS increases, GPU execution becomes cost-optimal and the chosen GPU batch size increases in discrete steps, yielding multiple knee points corresponding to batching transitions.

WRN50-4. Figure 5.5 shows the same analysis for WRN50-4. In the SLO sweep (left), the frontier shifts: the transition moves earlier and the CPU region is reduced, with the optimizer switching from GPU to partitioned CPU once the SLO becomes sufficiently slack. In the RPS sweep (right), WRN50-4 shows a single dominant transition from partitioned CPU at low rates to GPU execution

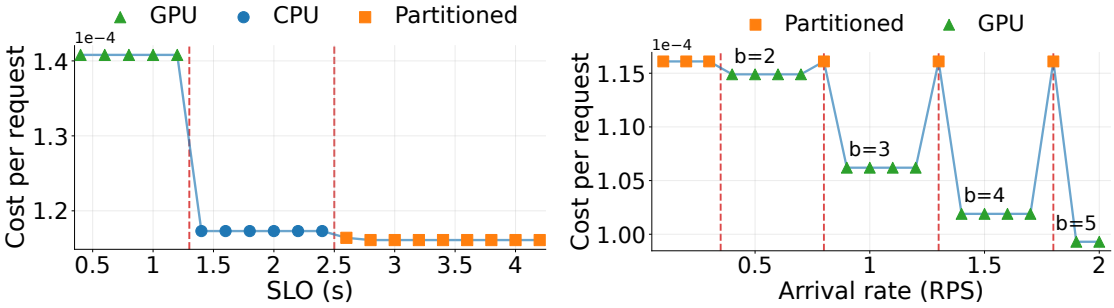


FIGURE 5.4: Normalized cost of the optimal provisioning plan for WRN50-5 under varying SLOs (left) and varying arrival rates (right). Markers denote the selected execution mode, dashed vertical lines indicate knee points, and the selected GPU batch size is annotated in the RPS sweep.

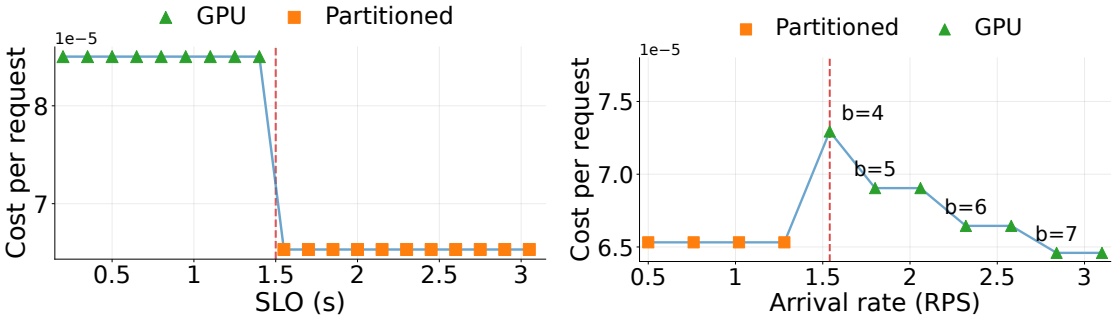


FIGURE 5.5: Normalized cost of the optimal provisioning plan for WRN50-4 under varying SLOs (left) and varying arrival rates (right). Markers denote the selected execution mode and dashed vertical lines indicate knee points.

as arrival rate grows, after which GPU remains cost-optimal with increasing batch sizes.

Key takeaway. Across both models, partitioned CPU occupies a non-trivial and consistent region of the design space: it is neither a replacement for GPU under tight SLOs nor simply a fallback, but an intermediate regime that becomes cost-optimal when GPU provisioning is inefficient (e.g., low/moderate load or sufficient latency slack), while batching dominates at higher arrival rates.

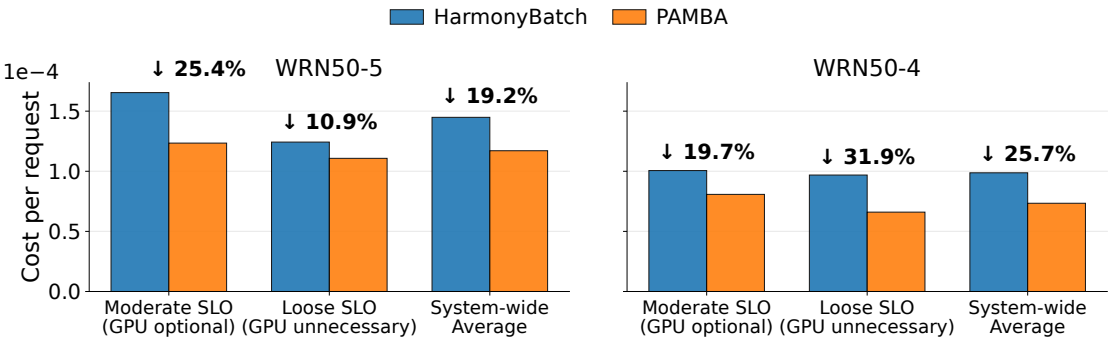


FIGURE 5.6: Comparison of the average monetary cost per request for multi-application workloads under moderate and loose SLO settings.

5.4 End-to-End Multi-Application Optimization

We evaluate the end-to-end effectiveness of our optimizer under multi-application workloads by comparing it against HarmonyBatch (HB), which supports single CPU and GPU execution. Our evaluation focuses on how execution-mode selection and consolidation interact across different SLO regimes and model complexities.

Figure 5.6 summarizes the average per-request cost achieved by both systems across representative operating regimes for two model variants. Where GPU acceleration is required under tighter latency constraints, both systems exhibit similar behaviour when GPU execution is selected, as they deploy identical GPU-backed functions. As a result, cost differences in this regime are negligible.

In the *moderate SLO* regime, multiple execution modes become feasible. While HarmonyBatch continues to rely on single-function CPU or GPU execution, our optimizer selectively deploys partitioned execution when it yields a lower cost while satisfying SLO constraints. This adaptive choice avoids over-provisioned resources and enables consistent cost reductions for both WRN50-5 and WRN50-4.

Under *loose SLOs*, GPU execution is unnecessary for either model. HarmonyBatch falls back to CPU-only single-function deployments, whereas our system favours partitioned execution. By decomposing inference into multiple lightweight functions, partitioned execution reduces per-function resource requirements and achieves substantially lower cost. This effect is particularly pronounced for WRN50-4, where partitioned execution yields the largest relative cost savings compared to HarmonyBatch.

Finally, the system-wide averages in Figure 5.6 confirm that these improvements are not isolated to a specific model or operating point. Across both WRN50-5 and WRN50-4, our system consistently reduces average cost per request relative to HarmonyBatch. These results demonstrate that incorporating partitioned execution into the optimization space is critical for achieving end-to-end cost efficiency in multi-application serverless inference workloads, especially when GPU acceleration is optional or unnecessary.

Other key results. PAMBA reduces SLO violations relative to HarmonyBatch. For WRN50-4, the violation rate drops from 4.36% to 1.09%, and for WRN50-5 it drops from 2.88% to 0.83%. In addition, partitioned CPU reduces the peak per-function memory requirement by 40% compared to monolithic CPU for both models.

5.5 Partitioned Execution Strategy Comparison

This subsection evaluates how the choice of *partition template* affects partitioned execution. We compare two templates: (i) our LO-based, cost-aware template

selection (Section 4.2.1) and (ii) the latency-optimal (LO) template produced by Gillis. We report end-to-end latency and per-request cost under identical Alibaba Function Compute constraints.

Methodology. We evaluate both templates in the same serverless environment. To isolate the effect of the partition template, we fix the master function to a minimal feasible configuration across all runs. We then vary the resource setting of the *worker functions* by sweeping the per-worker vCPU allocation; within each run, all workers use the *same* vCPU setting. Per-request cost follows our Alibaba billing model and includes the master plus all worker functions.

Latency–cost comparison. Figures 5.7 and 5.8 compare the resulting latency–cost trade-offs for WRN50-4 and WRN50-5. At the median evaluated per-worker vCPU setting, our template achieves **42.2% lower** per-request cost than the Gillis LO template on WRN50-4 and **43.2% lower** per-request cost on WRN50-5, with comparable end-to-end latency.

Why cost-aware templates help. Our partitioning favors templates with fewer stages and fewer workers per stage, keeping the total number of invoked functions (and thus billed compute) closer to the cost of single CPU execution. In addition, reducing the number of stages reduces per-stage coordination overhead on Alibaba FC, which can also improves end-to-end latency.

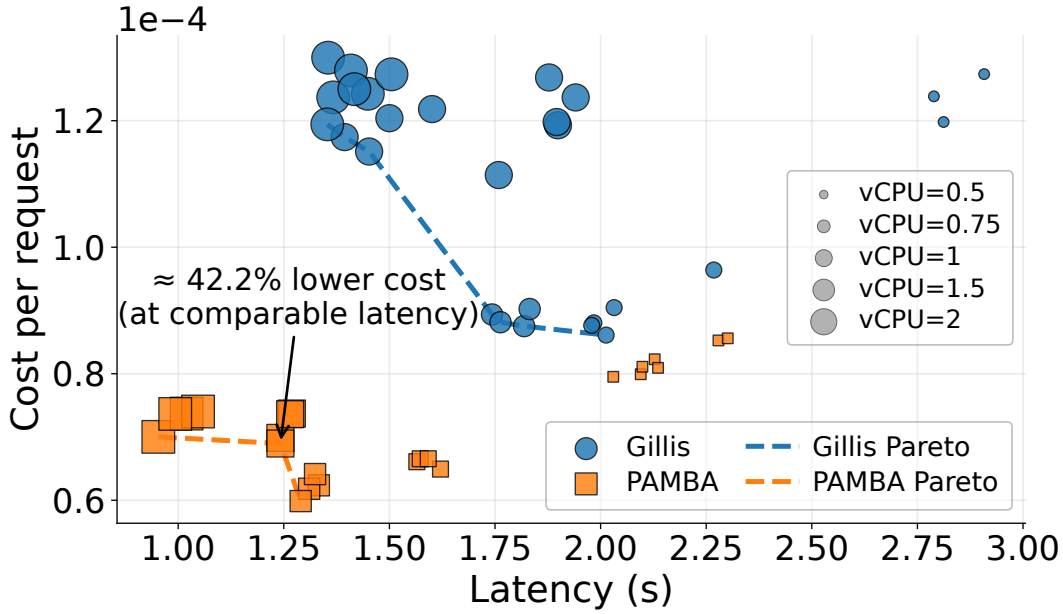


FIGURE 5.7: Latency–cost comparison of partitioned execution templates on WRN50-4 across evaluated per-worker vCPU settings. Each point corresponds to one deployment at a specific per-worker vCPU; marker size indicates the per-worker vCPU. The master is fixed to a minimal feasible configuration. Per-request cost includes the master and all stage/worker functions under Alibaba billing.

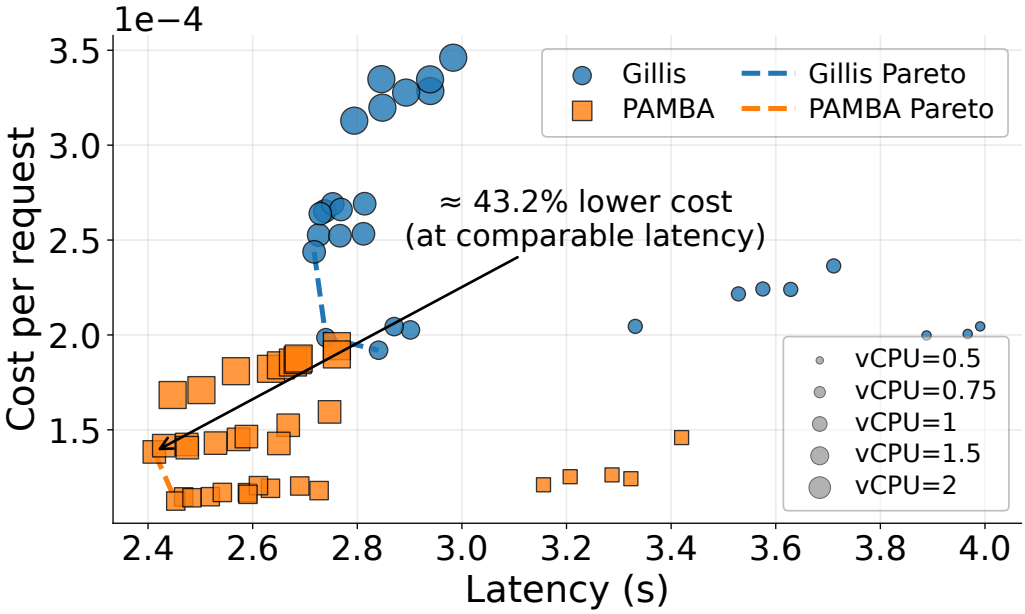


FIGURE 5.8: Latency–cost comparison of partitioned execution templates on WRN50-5 across evaluated per-worker vCPU settings, using the same setup as Figure 5.7. Marker size indicates the per-worker vCPU; the master is fixed to a minimal feasible configuration; and per-request cost includes the master and all stage/worker functions under Alibaba billing.

Chapter 6

Conclusion

This chapter concludes the thesis by summarizing the main contributions and experimental findings, followed by a discussion of limitations and directions for future work. The focus of this thesis is cost-efficient and SLO-aware serverless inference under heterogeneous execution options, where the serving system must jointly decide *how* to execute a model (CPU, GPU, or partitioned CPU), *how* to provision resources under platform feasibility and billing constraints, and *how* to batch and consolidate requests across multiple applications with different tail-latency objectives.

6.1 Summary

This thesis introduced **PAMBA**, a partition-aware and multi-SLO batching framework for serverless inference. The main contributions and findings are summarized at a high level as follows:

- **Unified decision-making for cost-efficient, SLO-aware inference.** PAMBA formulates serverless inference planning as an end-to-end optimization problem that jointly determines *execution mode* (monolithic CPU, GPU, or partitioned CPU), *batching/consolidation* under multiple latency SLOs, and *resource provisioning* under discrete platform configurations and pay-per-use billing.

- **Partitioned CPU as a first-class serving option.** Rather than treating partitioning only as a feasibility fallback, PAMBA explicitly includes **partitioned CPU execution** as a candidate in the optimization space alongside CPU and GPU. This enables an intermediate operating regime that can be selected when it offers a better cost–SLO trade-off than committing to CPU-only or GPU-only execution.

- **Profiling-driven models to support tail-SLO planning.** PAMBA uses profiling-based performance and cost models to compare candidate configurations across execution modes while targeting **tail latency** (p99) objectives. This modeling supports consistent feasibility checks and cost comparisons under realistic serverless constraints.

- **Two-phase workflow for practical optimization.** PAMBA combines **offline partition-template generation** (to construct a tractable set of partition options) with an **online optimizer** that selects among feasible execution modes, batching decisions, and resource configurations to satisfy SLOs at low cost.

- **Empirical evidence of improved cost–SLO trade-offs.** Across the evaluated workloads and multi-application settings, PAMBA identifies distinct regimes where the cost-optimal plan transitions between GPU, monolithic CPU, and partitioned CPU execution, and achieves meaningful **cost reductions** while improving or maintaining **SLO compliance** relative to a batching-based baseline.

- **Cost-aware partition template selection.** The evaluation shows that partition template choice materially affects the cost–latency outcome. PAMBA's cost-aware template selection improves the efficiency of partitioned execution by favoring templates that reduce unnecessary overheads and billed work while preserving comparable end-to-end latency.

6.2 Discussion, Limitations, and Future Work

6.2.1 Discussion

The results in Chapters 4 and 5 reinforce that cost-efficient serverless inference is fundamentally a *joint* decision problem. Execution mode selection, batching/consolidation decisions, and resource provisioning interact through platform feasibility constraints and pay-per-use billing. Within this coupled space, explicitly modeling partitioned CPU execution expands the feasible and cost-optimal operating region, particularly when GPU acceleration is available but cannot be justified economically for the current workload.

A recurring empirical pattern is the presence of *knee points* where the cost-optimal plan changes discretely. These knee points arise from two sources: (i)

feasibility transitions as an SLO becomes sufficiently slack for CPU (or partitioned CPU) to meet the tail-latency constraint, and (ii) batching transitions as GPU execution becomes cost-effective only after the arrival rate grows enough to amortize the accelerator cost through larger batch sizes. In this context, partitioned CPU is neither a replacement for GPU under tight SLOs nor merely an emergency fallback; it represents an intermediate regime that is selected when GPU provisioning is inefficient yet monolithic CPU remains suboptimal.

The partition-template comparison further clarifies why a serverless setting benefits from cost-aware partition choices. Templates with fewer stages and fewer workers per stage keep the number of invoked functions (and thus billed compute) closer to the cost envelope of single-function CPU execution. Reducing the number of stages also reduces per-stage coordination overhead, which can improve end-to-end latency. This aligns with the observed latency–cost trade-offs: cost-aware templates provide substantial cost savings at comparable latency under identical platform constraints.

6.2.2 Limitations

Several limitations follow from the current scope and design choices. First, the implementation and measurements are conducted on Alibaba Cloud Function Compute because it provides GPU-backed serverless functions and supports heterogeneous CPU/GPU/partitioned execution within a single platform. While the overall optimization structure is not provider-specific, quantitative decisions depend on profiling-based performance models and the platform's feasibility and billing constraints; deploying the approach on a new provider would require re-profiling

target configurations and instantiating the corresponding cost/constraint model.

Second, profiling and validation in this thesis are performed on warm invocations to capture the steady-state service time, and cold-start behavior is not explicitly modeled. In environments where cold starts occur frequently, unmodeled startup effects can influence tail latency and may reduce the accuracy of feasibility checks for stringent p99 SLOs.

Third, the evaluation focuses on the two representative convolutional workloads used throughout Chapters 4 and 5. Although these workloads are sufficient to demonstrate execution-mode frontiers, multi-application cost reductions, and the impact of template choice, additional architectures and operator mixes may alter where knee points occur and how large the partitioned-execution region becomes.

Finally, although GPU-partitioned execution was explored, profiling indicated it is typically not beneficial under current serverless constraints. Partitioning introduces orchestration and inter-stage communication overheads that are difficult to amortize on GPU unless per-stage GPU compute time is sufficiently large (e.g., very large models). Coarse-grained GPU allocation (e.g., one entire GPU per function) further limits the benefits of splitting a model across multiple GPU functions.

6.2.3 Future Work

The most direct future directions are to broaden both platform and model coverage. One path is to extend PAMBA to additional serverless platforms by re-profiling and instantiating platform-specific performance and cost/constraint models, enabling the optimizer to reflect each provider's configuration catalog and billing granularity. A second path is to broaden model support beyond the evaluated convolutional workloads, including larger architectures such as transformer-based models that may require different partitioning strategies and template-generation choices.

Another practical direction is to strengthen tail-latency robustness by explicitly incorporating cold-start behavior into feasibility and cost planning. Extending the modeling approach to capture startup effects would help preserve SLO compliance in bursty or scale-to-zero operating regimes where cold starts may contribute non-trivially to p99 latency.

Finally, it is valuable to revisit GPU-partitioned execution if serverless platforms evolve toward finer-grained accelerator allocation. If GPU resources can be provisioned in smaller units, the overhead-amortization balance may shift, making GPU partitioning more attractive and enabling it to be integrated into the same end-to-end optimization framework.

In conclusion, this thesis demonstrates that partition-aware, multi-SLO batching for serverless inference benefits from an integrated optimization view over execution mode, resource provisioning, and consolidation. By expanding the optimizer's decision space to include partitioned CPU execution and validating the

resulting execution-mode frontiers, cost reductions, and SLO improvements under multi-application workloads, PAMBA provides a practical foundation for cost-efficient and SLO-aware inference planning on heterogeneous serverless platforms.

Bibliography

[1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, *et al.*, "Serverless computing: Current trends and open problems," *Research advances in cloud computing*, pp. 1–20, 2017.

[2] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 13–24, 2019.

[3] Amazon Web Services, "AWS Lambda: Serverless Compute Service." Available at https://aws.amazon.com/lambda/, 2024. Accessed: March 3, 2025.

[4] Google Cloud, "Google Cloud Functions: Event-Driven Serverless Functions." Available at https://cloud.google.com/functions, 2024. Accessed: March 3, 2025.

[5] "Azure functions." https://azure.microsoft.com/en-us/products/functions/. Accessed: 2026-01-29.

[6] Alibaba Cloud, "Alibaba Cloud Function Compute: Serverless Cloud Service." Available at https://www.alibabacloud.com/product/function-compute, 2024. Accessed: March 3, 2025.

[7] "Ibm cloud code engine." https://cloud.ibm.com/docs/codeengine. Accessed: 2026-01-29.

[8] 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.

[9] 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.

[10] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker, "How does it function? characterizing long-term trends in production serverless workloads," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 443–458, Oct 2023.

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

[12] "Aws lambda pricing." https://aws.amazon.com/lambda/pricing/. Accessed: 2026-01-29.

[13] "Google cloud functions pricing." https://cloud.google.com/functions/pricing. Accessed: 2026-01-29.

[14] "Alibaba cloud function compute pricing." https://www.alibabacloud.com/product/function-compute/pricing. Accessed: 2026-01-29.

[15] "Alibaba cloud function compute billing overview." https://www.alibabacloud.com/help/en/functioncompute/fc/product-overview/billing-overview-of-fc. Accessed: 2026-01-29.

[16] A. Barrak, F. Petrillo, and F. Jaafar, "Serverless on machine learning: A systematic mapping study," *IEEE Access*, vol. 10, pp. 99337–99352, 2022.

[17] 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, USENIX Association, July 2021.

[18] 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, 2018.

[19] 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.

[20] 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.

[21] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Optimizing inference serving on serverless platforms," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, 2022.

[22] 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.

[23] J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim, "Gpu enabled serverless computing framework," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 533–540, IEEE, 2018.

[24] J. Gu, Y. Zhu, P. Wang, M. Chadha, and M. Gerndt, "Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference," in *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 635–644, 2023.

[25] 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.

[26] Z. Wu, Y. Deng, Y. Zhou, J. Li, and S. Pang, "Faasbatch: Enhancing the efficiency of serverless computing by batching and expanding functions," in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pp. 372–382, IEEE, 2023.

[27] J. Chen, F. Xu, Y. Gu, L. Chen, F. Liu, and Z. Zhou, "Harmonybatch: Batching multi-slo dnn inference with heterogeneous serverless functions," in *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, pp. 1–10, 2024.

[28] 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.

[29] 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.

[30] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Amps-inf: Automatic model partitioning for serverless inference with cost efficiency," in *Proceedings of the 50th International Conference on Parallel Processing*, pp. 1–12, 2021.

[31] J. Duan, S. Qian, D. Yang, H. Hu, J. Cao, and G. Xue, "Mopar: A model partitioning framework for deep learning inference services on serverless platforms," *arXiv preprint arXiv:2404.02445*, 2024.

[32] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving {DNNs} like clockwork: Performance predictability from the

bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 443–462, 2020.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] F. Xu, J. Xu, J. Chen, L. Chen, R. Shang, Z. Zhou, and F. Liu, "igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 812–827, 2022.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] Z. Zhao, Y. Hu, Z. Gong, G. Yang, W. Li, X. Liu, K. Li, and H. Wang, "Harpagon: Minimizing dnn serving cost via efficient dispatching, scheduling and splitting," in *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*, pp. 1–10, IEEE, 2025.

[53] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 768–781, 2022.

[54] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "Asyfunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pp. 324–340, 2023.

[55] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–16, 2019.

[56] 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.

[57] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, J. E. Gonzalez, *et al.*, "{AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, 2023.

[58] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 322–337, 2019.

[59] 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.

[60] 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.

[61] N. Mahmoudi and H. Khazaei, "Temporal performance modelling of serverless computing platforms," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pp. 1–6, 2020.

[62] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2834–2847, 2020.

[63] N. Mahmoudi and H. Khazaei, "Performance modeling of metric-based serverless computing platforms," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1899–1910, 2022.

[64] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proceedings of the 22nd International Middleware Conference*, pp. 248–259, 2021.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] "Apache mxnet." https://mxnet.apache.org/. Accessed: 2026-01-29.