SERFLOW: A CROSS-SERVICE COST OPTIMIZATION FRAMEWORK FOR SLO-AWARE DYNAMIC ML INFERENCE

Zongshun Zhang [1] **Ibrahim Matta** [1]

ABSTRACT

Dynamic offloading of Machine Learning (ML) model partitions across different resource orchestration services, such as Function-as-a-Service (FaaS) and Infrastructure-as-a-Service (IaaS), can balance processing and transmission delays while minimizing costs of adaptive inference applications. However, prior work often overlooks real-world factors, such as Virtual Machine (VM) cold starts, requests under long-tail service time distributions, etc. To tackle these limitations, we model each ML query (request) as traversing an acyclic sequence of *stages*, wherein each stage constitutes a contiguous block of sparse model parameters ending in an internal or final classifier where requests may exit. Since input-dependent exit rates vary, *no single resource configuration suits all query distributions*. IaaS-based VMs become underutilized when many requests exit early, yet rapidly scaling to handle request bursts reaching deep layers is impractical. SERFLOW addresses this challenge by leveraging FaaS-based serverless functions (containers) and using stage-specific resource provisioning that accounts for the fraction of requests exiting at each stage. By integrating this provisioning with adaptive load balancing across VMs and serverless functions based on request ingestion, SERFLOW reduces cloud costs by over 23% while efficiently adapting to dynamic workloads.

1 OVERVIEW

The broad adoption of Machine Learning as a Service (MLaaS) enables cost-effective orchestration across edge and core clouds. Recent deployments at companies (Adobe, 2023; Amazon Web Services, 2024; Campbell Webb, 2024) use Infrastructure-as-a-Service (IaaS) and containers to cut cost and scale quickly. Building on this trend, we combine IaaS with Function-as-a-Service (FaaS), whose fine-grained billing lowers costs for sparsely activated ML models while meeting latency service-level objectives (SLOs).

Infrastructure-as-a-Service (IaaS) lets tenants lease preconfigured virtual machines (VMs) (ama, 2020; goo, 2020). To handle dynamic loads, cloud providers offer auto-scaling that adjusts VM counts by user metrics (e.g., CPU, memory) (amz, 2018; goo, 2018). VMs deliver steady compute at low per-unit cost, suiting long-running, predictable workloads. However, scaling out VMs has cold starts: new instances can take hundreds of seconds before serving traffic (Zhang et al., 2019; Shahrad et al., 2020).

Function-as-a-Service (FaaS) uses a lightweight execution model. Tenants provide only code (the "serverless function") and dependencies. On request, the platform provisions a containerized instance and bills by execution time and allocated resources (Castro et al., 2019). Because containers are thin and often kept warm, cold starts can be a few milliseconds (Wang et al., 2018; Amazon Web Services, 2025). But FaaS has higher per-unit cost for sustained compute, making it most cost-effective for transient spikes.

Leveraging IaaS for steady, long-running workloads and FaaS for bursty, short-lived tasks, MLaaS can optimize both monetary cost and performance. Prior work shows that routing transient requests to FaaS and stable ones to IaaS reduces cost while meeting latency SLOs (Zhang et al., 2019; Raza et al., 2021; Gunasekaran et al., 2019). These methods profile runtimes to choose cost-efficient VM and serverless configs, with best savings when VMs run at high utilization and serverless calls are few. However, this assumes dense models with stable request runtimes, often false in modern ML. Today's inference pipelines are directed acyclic graphs (DAGs) of heterogeneous models, each stage showing distinct compute needs and invocation rates (Shen et al., 2024; Mudvari et al., 2024; Feng et al., 2025; Behera et al., 2025).

Focusing on deep learning applications with variable request running times, this paper presents SERFLOW, a load balancing and resource provisioning system that minimizes cost while meeting strict SLOs. SERFLOW introduces an end-to-end configuration paradigm that accounts for per-stage runtimes in a request-specific model DAG. While the framework also applies to independent models with stable

[1]Department of Computer Science, Boston University, Boston, US. Correspondence to: Zongshun Zhang <zhangzs@bu.edu>.

runtimes (as in prior work that ignores ML architectural dependencies (Raza et al., 2021; Zhang et al., 2019)), our main contribution is fine-grained orchestration of dependent stages. We make the following contributions:

- We develop an economic model for IaaS–FaaS cost trade-offs, introducing the *Sparsity Cost Indifference Point* (S-CIP) for model-induced runtime variance and its interplay with the *Traffic Cost Indifference Point* (T-CIP), which links ingestion rate to VM utilization. These parameters guide cost-minimizing offloading of processing requests.
- We present SERFLOW, a hybrid offloading system that sends low-traffic ML model partitions to FaaS while keeping high-traffic partitions on IaaS, minimizing cost under strict SLOs.
- We implement SERFLOW on AWS and evaluate it on multi-stage models with variable runtimes, showing large cost savings and strong SLO adherence.

We find that when S-CIP reflects more runtime variability across requests, SERFLOW achieves greater monetary savings than prior approaches, due to improved VM utilization. In this work, we use the internal classifiers' confidence threshold ($conf\_thres$) to determine S-CIP. Lowering $conf\_thres$ increases cost savings, because more requests exit early before using FaaS, but slightly reduces model accuracy. At $conf\_thres = 0.7$, SERFLOW reduces costs by 23.38% while maintaining 0.85 top-1 accuracy compared to LIBRA (Raza et al., 2021).

The paper is organized as follows. In Section 2, we present the background and motivate SERFLOW by formulating the cost model for IaaS and FaaS during ML inference. In Section 3, we describe the system architecture of SERFLOW. Section 4 evaluates SERFLOW on various ML model DAGs, and Section 5 concludes the paper.

2 Background & Motivation

Prior work leverages FaaS to hide IaaS cold starts (Gunasekaran et al., 2019; Novak et al., 2019; Zhang et al., 2019) and load balances bursty traffic to FaaS while directing stable traffic to IaaS (Raza et al., 2021). This hybrid strategy minimizes cost and maintains SLOs. LIBRA (Raza et al., 2021) defines a "Cost Indifference Point" (CIP) based on the moving average of incoming requests. Traffic above this CIP is treated as transient and sent to FaaS, while the rest is sent to IaaS. However, most prior work overlooks model sparsity in modern ML systems and assumes uniform request runtimes. Instead, SERFLOW focuses on multi-stage inference pipelines in which the runtime for a request varies with its path through the model DAG. We study the interactions between two complementary parameters:

- **Traffic Cost Indifference Point (T-CIP)**: A threshold on residual load $r_{\text{res}} = (\mu + \sigma) \bmod r_{\max}$ (req/s). Here, μ is the moving average and σ the moving deviation of the request arrival rate λ_t. $r_{\max}$ represents the request load that a VM or FaaS instance can handle while maintaining SLO. If $r_{\text{res}} \leq$ T-CIP, FaaS is cheaper; if $r_{\text{res}} >$ T-CIP, IaaS or Hybrid Offloading (i.e., VMs followed by FaaS for long-running requests) is cheaper. Thus, T-CIP indicates whether to scale VMs/Hybrid Offloading instances or offload to FaaS.
- **Sparsity Cost Indifference Point (S-CIP)**: A confidence threshold ($conf\_thres$) for internal classifiers (ICs). At a fixed arrival rate of requests, S-CIP is the $conf\_thres$ where the expected costs of IaaS-only or FaaS-only versus Hybrid Offloading are equal; it captures how exit sparsity of requests (and thus the multi-stage completion-time distribution for requests) affects cost.

First, SERFLOW uses S-CIP to pick the cost-optimal resource configuration for the distribution of observed requests over exits. Next, it applies the T-CIP to scale IaaS and FaaS resources based on online traffic. This two-step process adapts to per-request runtime variance and workload dynamics, lowering cost while meeting strict latency SLOs.

2.1 Multi-Stage Requests

Prior work treats ML inference as a single *stage*, with all requests exiting at the final classifier (Fig. 1a). By contrast, modern pipelines have multiple *stages*, and each request may take a different path in a model DAG. An early example, BranchyNet (Teerapittayanon et al., 2016), adds internal classifiers for early exits (Fig. 1b). To capture this variability, we generalize the cost model to stage-wise runtimes.

We assume a steady ingestion rate of N requests per second and profile running time and cost under a stable queueing scenario. Notice that N differs from $r_{\max}$, which denotes the capacity, i.e., the maximum number of requests per second that a VM, serverless instance, or our hybrid offloading instance (Sec. 2.2) can sustain within the SLO. In this section, we use a long-term average traffic rate N for profiling. In Sec. 4, we consider a time-varying ingestion rate λ_t and provision resources to satisfy λ_t given each instance's capacity r_{max}. Table 1 summarizes the ingestion rate notations.

We profile each stage's mean runtime under FaaS config θ_F, denoted T^F_{stage}. By Little's Law (Little, 1961), the expected in-flight request count is $\sum_{req\_id=1}^{N} \sum_{stage} T^F_{stage}$. Thus, the FaaS cost per second at rate N req/s is

$$C^F = c^{\theta_F} \sum_{req\_id=1}^{N} \sum_{stage} T^F_{stage} \tag{1}$$

Notation	Definition
λ_t	Online ingestion rate at time t
N	The long-term average traffic rate for profiling given requests sampled from historical λ_t
$r_{\max}$	SLO-aware max per-instance ingestion rate (VM, hybrid offloading, or serverless)

Table 1. SERFLOW ingestion rate notations

where c^{θ_F} is the per-second price under θ_F.

Similarly, the per-second VM cost C^I is a function of N, SLO, $r_{\max}$ (profiled from VM capacity/SLO), and the unit VM price c^{θ_I} under θ_I. We need $\lfloor N/r_{\max} \rfloor$ VMs on average. We add one additional VM if the remainder exceeds T-CIP, otherwise, we route the remainder to FaaS.

$$C^I = \left(\left\lfloor \frac{N}{r_{\max}} \right\rfloor + \mathbf{1}(N \bmod r_{\max} > T\_\text{CIP}) \right) c^{\theta_I} \text{SLO} \tag{2}$$

VM cold starts are costly, so prior work keeps VMs active for the entire SLO. But for multi-stage requests with variable durations, keeping idle VMs is cost-inefficient.

2.2 Varying Duration Requests

An example of a multi-stage model is a sequential Deep Neural Network (DNN) with internal classifiers (ICs) that enable early exits (predictions in shallow layers.) (Teerapittayanon et al., 2016) ICs reduce floating point operations (FLOPs) per request on average and maintain accuracy by preserving features extracted at shallow layers (Kaya et al., 2019). Similarly, Mixture of Experts (MoE) (Shazeer et al., 2017; Gross et al., 2017) in Large Language Models (LLMs) and other structured architectures exploit sparsity to lower computational cost. However, when many requests exit early, sparsely activated models may under-utilize provisioned resources (Purandare et al., 2023; Lewis et al., 2021).

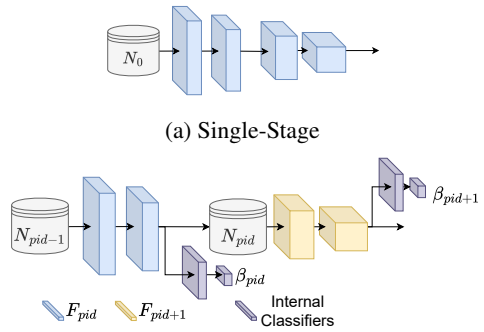
2.2.1 *Model Sparsity*

As shown in Fig. 1b, an NN can be partitioned into layer sequences, each ending at an internal classifier (IC). We index partitions with $pid \in [0, L]$. Partition 0 spans from the input layer to the first IC. For $pid > 0$, partition pid spans from the layer after $(pid-1)^{th}$ IC up to the pid^{th} IC. With confidence threshold $conf\_thres$, a fraction β_{pid} of requests with confidence above the threshold exit at partition pid. Thus, if N_{pid-1} requests enter partition pid,

$$N_{pid} = (1 - \beta_{pid})\, N_{pid-1}$$

requests per second continue to the next partition (stage).

Traditional load-balancing approaches, e.g., LIBRA (Raza et al., 2021), would recompute C^F and C^I for subsequent



(a) Single-Stage

(b) Multi-Stage

Figure 1. Single-Stage (Fig. 1a): All requests from the source (N_0) go through the whole model; Multi-Stage with Internal Classifiers (Fig. 1b): β_{pid} of requests exit at NN partition F_{pid} and the remaining N_{pid} requests per second are fed into partition F_{pid+1}.

partitions and scale resources. Under strict SLO constraints, however, these methods must keep idle VMs to cover worst-case demand (i.e. requests using all stages of the ML model) leading to oversubscription of resources, and thus introducing cost overhead. In contrast, we leverage FaaS's fine-grained billing and rapid scaling for deep partitions to handle late-exiting requests, while provisioning compact VMs only for early partitions with stable, high-volume traffic, achieving improved utilization and reduced cost.

2.2.2 *Global Optimization*

We first derive cost models for FaaS-only and IaaS-only multi-stage ML inference, then extend to a Hybrid Offloading strategy where partitions $\leq cut\_id$ run on VMs and later partitions on FaaS. All models assume a long-term average arrival rate of N req/s.

The FaaS formulation follows Equation 1. Let $T^{\theta_F}_{k+1}$ be the profiled runtime of partition $k+1$ under FaaS config θ_F. Since a fraction $\sum_{i=1}^{k} \beta_i$ of requests exits at partition k, the arrival rate to partition $k+1$ is $(1 - \sum_{i=1}^{k} \beta_i)\, N$. The FaaS cost per second for all L partitions with unit cost c^{θ_F} is:

$$C^F = c^{\theta_F}(NT^F_1 + \sum_{k=1}^{L-1}(1 - \sum_{pid=1}^{k} \beta_{pid}) NT^{\theta_F}_{k+1}) \tag{3}$$

The VM formulation follows Equation (2). We provision the smallest VM that can process $r_{\max}$ requests within the SLO and run it for the full SLO duration.

In hybrid offloading, the VM cost is flat since VMs run for the full SLO duration. As FaaS scales to 10,240 MB (5.78 vCPUs) (AWS, 2024), we use a smaller VM (e.g., `c6i.large`, 2 vCPUs) than VM-only setup (e.g., `c6i.xlarge`, 4 vCPUs) and offload partitions `cut_id`+1

through L to FaaS. The per-second cost for N req/s is:

$$C^H = \left(\left\lfloor \frac{N}{r_{\max}} \right\rfloor + \mathbf{1}(N \bmod r_{\max} > T\_\text{CIP})\right) c^{\theta_I} \text{SLO} + c^{\theta_F} \sum_{k=\text{cutid}}^{L-1} \left(1 - \sum_{\text{pid}=1}^{k} \beta_{\text{pid}}\right) N T_{k+1}^{\theta_F} \quad (4)$$

Three factors determine resource cost: the distribution of β_{pid} (via `conf_thres`), the partition index `cut_id`, and the ingestion rate N. If the remainder $N \bmod r_{\max}$ is below the T-CIP, that portion of traffic is routed to FaaS as it is cheaper than allocating an additional VM. Figures 2 and 3 compare IaaS-only (C^I), FaaS-only (C^F), and hybrid offloading (C^H) as `conf_thres` varies. In this example, we fix `cut_id` $= 5$ and $N = r_{\max} = 100/6$ req/s (100 requests per 6s, matching the target SLO), using `c6i.xlarge` for IaaS-only, 8,845MB for FaaS-only, and `c6i.large` + 8,845MB for the hybrid offloading setup.

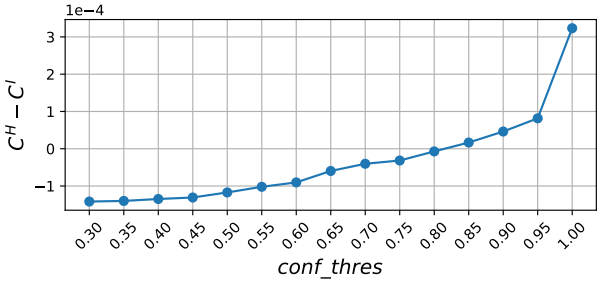


Figure 2. When $conf\_thres \leq 0.8$, Hybrid Offloading is cheaper than IaaS-only, given $r_{max} = 100/6$.

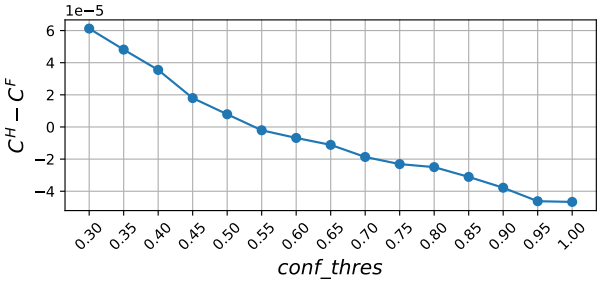


Figure 3. When $conf\_thres \geq 0.55$, Hybrid Offloading is cheaper than FaaS-only, given $r_{max} = 100/6$.

$C^H - C^I < 0$ for $conf\_thres \leq 0.8$, because a lower $conf\_thres$ causes most requests to exit before the deeper partitions. As a result, the larger VMs in the IaaS-only setup are more under-utilized than the smaller VMs in hybrid offloading, making the hybrid approach more cost-efficient. Likewise, $C^H - C^F < 0$ for $conf\_thres \geq 0.55$, as the smaller VMs in the hybrid offloading setup reach higher utilization when deeper partitions dominate, making the hybrid approach less costly. Thus, for $conf\_thres \in [0.55, 0.8]$, hybrid offloading is more cost-effective than IaaS-only or FaaS-only, provided the chosen $conf\_thres$ threshold satisfies the model's accuracy requirement.

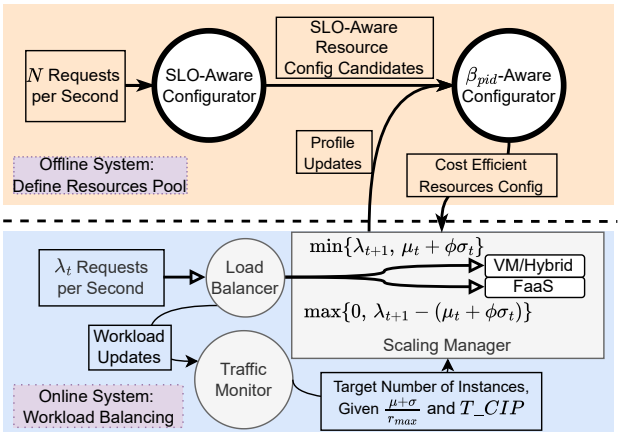


Figure 4. Overview of SERFLOW: Offline, we replay steady traffic with long-term average N req/s from historical traces to profile three candidate setups (VM-only, Hybrid Offloading, FaaS-only). Step 1: the SLO-Aware Configurator keeps only SLO-feasible configs. Step 2: given the observed early-exit rate β, the β-Aware Configurator picks the lowest-cost setup from the three candidates. Online, the Scaling Manager maintains a group of low-cost instances derived from the selected setup, using the EWMA (μ_t) and deviation (σ_t) of the online arrival rate λ_t (req/s). The load balancer then handles spikes at time $t+1$, λ_{t+1}, by first fully utilizing the available low-cost instances provisioned at time t and sending any remaining traffic to FaaS. When the β distribution over early exits drifts, the system triggers targeted re-profiling and re-selection via the β-Aware Configurator.

3 SERFLOW Architecture

SERFLOW dynamically solves the computation-offloading problem by partitioning the DNN into fine-grained stages and assigning each stage to either VMs or FaaS under strict latency SLO constraints. We illustrate this architecture with an image classification pipeline based on a partitioned DNN augmented with internal classifiers.

As shown in Fig. 4, SERFLOW comprises offline and online components. Given an ingestion rate λ_t at time t in the online system, the offline system samples a persistent traffic with a long-term average of N req/s and uses the per-instance capacity $r_{\max}$ to profile VM-only, FaaS-only, and hybrid offloading configurations under various confidence thresholds ($conf\_thres$). The ingestion rate notations are summarized in Table 1. The offline system comprises:

- *SLO-aware Configurator*, which selects VM and/or FaaS configurations that satisfy the latency SLO.

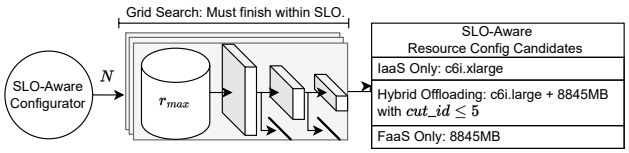


Figure 5. The SLO-aware Configurator searches for the SLO-compliant configurations given a per-instance ingestion rate r_{max} requests per second for VMs, Hybrid Offloading and Serverless.

- The β_{pid}*-aware Configurator* evaluates the cost of each candidate configuration based on the distribution of early-exit rates $\{\beta_{pid}\}$ across different partitions pid.

Using the cost-efficient SLO-satisfying configuration provided by the offline system, the online system consists of the following components:

- *Traffic Monitor*, which updates the EWMA μ and moving standard deviation σ of arrival rate λ_t. It then computes the target VM count based on $\frac{\mu+\sigma}{r_{\max}}$ and the Traffic Cost Indifference Point (T-CIP), directing the Scaling Manager as needed.
- *Scaling Manager*, which provisions or terminates VM instances based on the Traffic Monitor's instructions.
- *Load Balancer*, which routes the stable traffic portion (up to $\mu + \sigma$ req/s) to VMs (each provisioned at $r_{\max}$ req/s) and offloads any excess $\big(\lambda_t - (\mu + \sigma)\big)$ to FaaS.

3.1 Instance Configuration

To minimize cost while meeting latency SLOs online, SERFLOW first profiles and configures instances offline. Given a per-instance capacity $r_{\max}$ (req/s), the *SLO-aware Configurator* enumerates candidate resource configurations, profiles request runtimes, and selects the most cost-efficient configuration whose worst-case latency, when $r_{\max}$ req/s traverse all model partitions with $conf\_thres = 1$, does not exceed the SLO (Fig. 5, Alg. 1). The *SLO-aware Configurator* evaluates three candidate setups, VM-only, FaaS-only, and Hybrid Offloading, as illustrated in Fig. 6. Here, N denotes the long-term average traffic used for offline profiling and configuration, whereas λ_t denotes the instantaneous arrival rate observed by the online system at time t.

For the FaaS-only setup (Fig. 6c), the online system load-balances traffic at a per-function cap $r_{\max}$ req/s. Architects may configure distinct $r_{\max}$ for IaaS, FaaS, or hybrid offloading to meet latency targets. Empirically, for many CPU-intensive FaaS workloads, there is an optimal memory configuration that minimizes the per-invocation cost. This is due to the inherent trade-off in FaaS billing, where higher memory (with a higher per-unit cost) grants more compute and a shorter runtime. While increasing memory size impacts runtime, the cost per invocation often remains within a stable range, though not constant, as compute duration decreases to offset the higher per-unit cost (see Sec. 4.2). On AWS Lambda, allocating multiples of $1,769$ MB avoids hardware sharing. We thus select $8,845$ MB, the largest such multiple below the $10,240$ MB limit.

For the VM-only and hybrid offloading setups (Figs. 6b and 6a), the load balancer routes the stable portion of traffic to the VM or hybrid instance and the transient portion to FaaS (Sec. 3.3). Each VM type has distinct cost and capacity. For our compute-intensive image classification with $r_{\max} = 100/6$ req/s (100 requests per 6s, SLO = 6s), the configurator selects `c6i.xlarge` for VM-only. In the hybrid offloading setup, profiling (Fig. 7) shows that a `c6i.large` VM with $cut\_id \leq 5$ satisfies the SLO.

Next, the β_{pid}*-aware Configurator* lowers $conf\_thres < 1$ to enable early exits and measures exit rates $\{\beta_{pid}\}$ at rate N req/s. It then picks the lowest-cost configuration that meets the target accuracy, as shown in Fig. 8 and Alg. 2.

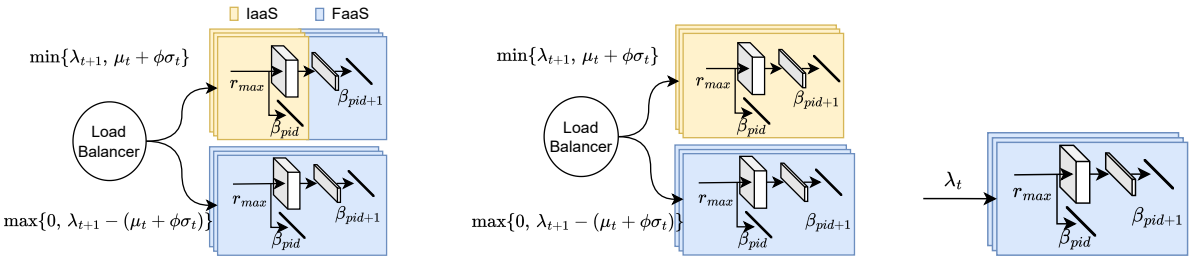
We compute the end-to-end cost via Equations 2, 3, and 4 and, for each $(N, r_{\max}, \beta_{pid})$ triplet, pick the lowest-cost configuration. The β_{pid} distributions under $conf\_thres$ (Figs. 9a, 9c, 9e) yield the estimated costs in Figs. 9b, 9d, and 9f, which plot monetary cost in USD (y-axis) vs. cut_id[1] (x-axis). Homogeneous setups (IaaS-only, FaaS-only) have a fixed cost irrespective of cut_id. In hybrid offloading (Fig. 9d), the VM cost is flat (as VMs are allocated for the full SLO duration), but the FaaS share of the cost drops with increasing cut_id as more work occurs on the VMs, giving a minimum at $cut\_id = 5$ within the SLO. Increasing cut_id further breaks the SLO budget (Fig. 7). We summarize the optimal strategy for each β distribution:

- **Early exits (Fig. 9a).** When most requests exit at shallow internal classifiers, FaaS-only is most cost-efficient ($C^F \leq C^I$ and $C^F \leq C^H$ for $cut\_id \leq 5$) due to under-utilization across VM sizes.
- **Late exits (Fig. 9e).** When most requests exit at deep internal classifiers, VM-only is optimal ($C^I \leq C^F$ and $C^I \leq C^H$ for $cut\_id \leq 5$), due to lower variance in per-request runtimes.
- **Intermediate exits (Fig. 9c).** When exits concentrate around middle internal classifiers, hybrid offloading is optimal ($C^H \leq C^F$ and $C^H \leq C^I$ at $cut\_id = 5$ while meeting the latency SLO).

3.2 Instance Scaling

With the cost-efficient instance configuration, SERFLOW selects a scaling strategy based on the moving average ingestion rate and the Traffic Cost Indifference Point (T-CIP). Fig. 10 summarizes the online system behaviors. The Traffic

[1]The head portion of the DNN consists of the shallow partitions up to `cut_id`, which corresponds to the internal classifier's index.



(a) Hybrid Offloading (b) IaaS Only (c) FaaS Only

Figure 6. Resource configuration setups used by the online system and profiled by the offline system: hybrid offloading (Fig. 6a), IaaS only (Fig. 6b) and FaaS only (Fig. 6c). Instances (VM, hybrid offloading, or serverless) are provisioned with capacity $r_{\max}$ req/s to handle stable $\min\{\lambda_{t+1},\ \mu_t + \phi\sigma_t\}$ and transient $\max\{0,\ \lambda_{t+1} - (\mu_t + \phi\sigma_t)\}$ traffic with a parameter $\phi \in \mathbb{R}$ from ingestion online traffic λ_t over time t. Notice that we use long-term average of N req/s instead of $\mu + \sigma$ for profiling the three setups in the offline system.

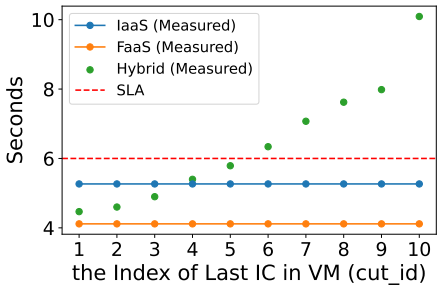


Figure 7. Profiled mean request runtimes, with $r_{\max} = \frac{100}{6}$ and $conf\_thres = 1$. At $cut\_id \leq 5$, the 6s SLO is maintained.

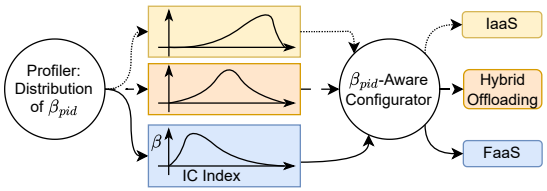


Figure 8. The β_{pid}-aware Configurator evaluates configuration candidates by profiling the distribution of early exits (β_{pid}) given an ingestion rate of N.

Monitor observes the arrival λ_{t+1} and separates persistent and transient load using an exponentially weighted moving average (EWMA) and deviation, updated at time $t{+}1$:

$$\mu_{t+1} = (1 - W^{\mu})\,\mu_t + W^{\mu}\,\lambda_{t+1}, \tag{5}$$

$$\sigma_{t+1} = (1 - W^{\sigma})\,\sigma_t + W^{\sigma}\,\big|\lambda_{t+1} - \mu_{t+1}\big|. \tag{6}$$

Accordingly, the system scales long-lived instances to the target $\mu_{t+1} + \phi\,\sigma_{t+1}$ (with tunable $\phi \in \mathbb{R}$) for stability. We set $\phi = 1$ in evaluation. Related work (Raza et al., 2021) studies cost trade-offs when tuning ϕ.

Based on the offline β_{pid}-aware Configurator's recommendation, the Traffic Monitor and Scaling Manager use one of three candidate pools

$$\{\text{IaaS, FaaS}\}, \quad \{\text{Hybrid Offloading, FaaS}\}, \quad \{\text{FaaS}\}.$$

Let $r_{\max}$ denote per-instance service rate (req/s) and define

$$k_{t+1} = \left\lfloor \frac{\mu_{t+1} + \sigma_{t+1}}{r_{\max}} \right\rfloor,$$

$$r^{\text{res}}_{t+1} = (\mu_{t+1} + \sigma_{t+1}) - k_{t+1}\, r_{\max} \in [0,\ r_{\max}).$$

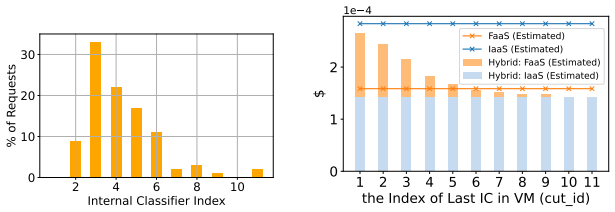
We define the *Traffic Cost Indifference Point* (T-CIP) as a threshold on r^{res}_{t+1}. The Scaling Manager provisions

$$\#\text{instances}_{t+1} = k_{t+1} + \mathbf{1}\left[r^{\text{res}}_{t+1} > \text{T-CIP}\right],$$

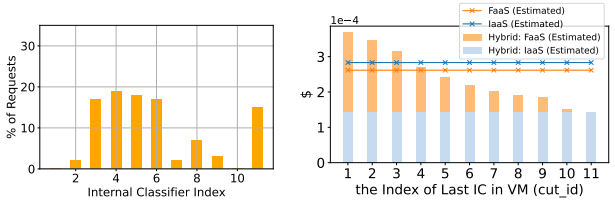
where the instances are *IaaS* or *Hybrid Offloading* according to the selected pool (no VM is created in the {FaaS} pool). Instance scaling depends on the smoothed target $\mu_{t+1}{+}\sigma_{t+1}$ rather than instantaneous λ_{t+1}, as illustrated in Fig. 10 and Alg. 4. Correspondingly, the online load balancer uses all provisioned IaaS/Hybrid Offloading instances and sends any excess demand (requests) to FaaS, as detailed in Sec. 3.3.

To identify T-CIP for each pool, the offline configurator uses our cost models to estimate each configuration's cost as a function of the residual load $N \bmod r_{\max}$. For example, Fig. 11 plots $C^F - C^H$ and $C^F - C^I$ versus N for $r_{\max} = 100$, `c6i.large` (hybrid offloading) or `c6i.xlarge` (VM-only), $cut\_id = 5$, and $conf\_thres = 0.7$. In Fig. 11, the point where $C^F = C^H$ (Hybrid and FaaS have equal cost) occurs at $N \bmod r_{\max} = 15$ req/s. For $N \bmod r_{\max} > 15$, provisioning a hybrid offloading instance is cheaper; for $N \bmod r_{\max} < 15$ req/s, FaaS is more economical. Meanwhile, there is no point where $C^F = C^I$. Larger N improves IaaS-only utilization, but at $N \bmod r_{\max} = 0$, Hybrid Offloading is still cheaper.[2] Thus, in this example, T-CIP = 15 req/s for Hybrid Offloading instances. When $r^{\text{res}}_{t+1} \leq 15$, the scaling manager keeps k_{t+1} instances. Otherwise, it scales to $k_{t+1} + 1$ instances.

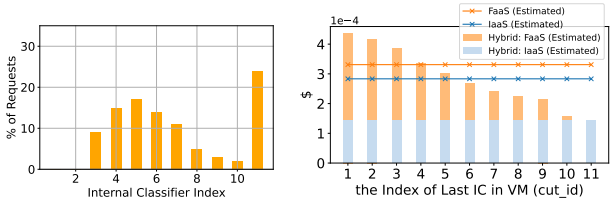
[2]We omit the results with $conf\_thres$ favoring FaaS-only or IaaS-only setups (lower or higher $conf\_thres$, respectively) as the profiling steps are the same.



(a) Most exits at shallow ICs. (b) FaaS is cheaper ($cut\_id \leq 5$). $conf\_thres = 0.5$



(c) Most exits at middle ICs. (d) Hybrid is cheaper ($cut\_id \geq 5$). $conf\_thres = 0.7$



(e) Most exits at deep ICs. (f) IaaS is cheaper ($cut\_id \leq 5$). $conf\_thres = 0.85$

Figure 9. Estimated average costs for $N = \frac{100}{6}$ req/s under varying β_{pid} distributions and setups. At $cut\_id \leq 5$, the SLO holds for all setups (Fig. 7), and the cost-minimizing setup shifts with the confidence threshold: FaaS-only at low $conf\_thres$ (e.g., 0.5), Hybrid Offloading at medium $conf\_thres$ (e.g., 0.7), and IaaS-only at high $conf\_thres$ (e.g., 0.85).

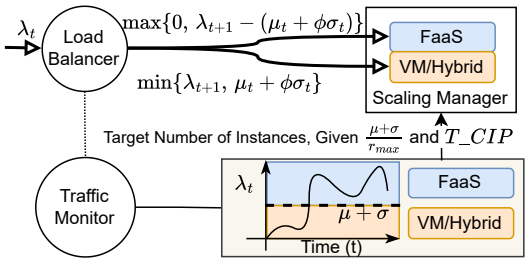


Figure 10. The online system load-balances λ_t requests per second and provisions the resources configured by the offline system.

3.3 Load Balancing

With a cost-efficient scaling policy, SERFLOW keeps enough VM/Hybrid Offloading instances to handle mean traffic cheaply. It must also absorb spikes without waiting for auto scaling. Fig. 10 and Alg. 3 summarize the spike-handling pipeline. SERFLOW sets a load-balancer threshold at $\mu_t + \phi\,\sigma_t$ (tunable $\phi \in \mathbb{R}$). In epoch $t{+}1$, traffic up to this threshold goes to the long-lived path (IaaS or Hybrid Offloading), and any excess goes to FaaS:

$$\text{long-lived load} = \min\{\lambda_{t+1},\ \mu_t + \phi\sigma_t\},$$
$$\text{FaaS spill} = \max\{0,\ \lambda_{t+1} - (\mu_t + \phi\sigma_t)\}.$$

Because resource provisioning has nonzero latency, the balancer in epoch $t{+}1$ uses instances scaled in epoch t. Newly requested capacity becomes available in the next scaling epoch. In our implementation, the balancer fills all available IaaS/Hybrid Offloading instances with the long-lived portion of the demand and forwards the remaining requests to FaaS. We batch up to $r_{\max}$ per instance. Each batch is routed to an available instance. If none is free, a FaaS function is invoked. Residual requests that do not form a full batch are treated identically: they go to a IaaS/Hybrid Offloading instance when available, otherwise to FaaS (Alg. 4). SERFLOW depends on accurate profiles of $\{\beta_{pid}\}$ and the ingestion rate λ_t. In practice, a background process would periodically update these profiles and adjust the resource pool. We leave dynamic re-profiling to future work.

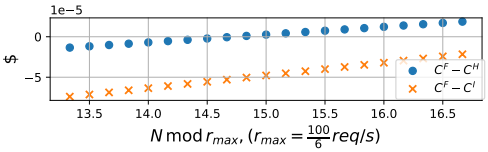


Figure 11. Cost differences $C^F - C^H$ and $C^F - C^I$ as a function of total ingestion rate $N \bmod r_{\max}$, for $r_{\max} = 100/6$, c6i.large (Hybrid), c6i.xlarge (IaaS), FaaS at 8845 MB, a VGG-16 with internal classifiers, and CIFAR-10. In Fig. 11 with $conf\_thres = 0.7$, T-CIP between FaaS and Hybrid is 15 req/s.

Algorithm 1 SLO-Aware Configurator

Input:

- $\{N\}$: persistent traffic with long-term average N req/s
- $r_{\max}$: req/s per VM
- SLO: max runtime per request
- $\{L_{pid}\}$: DNN layers
- $\{\theta_I\}, \{\theta_F\}$: IaaS and FaaS configs

Output: cost-optimal (θ_I^*, θ_F^*) meeting the SLO

$S \leftarrow \text{SAMPLING}(\{N\}, \text{count} = r_{\max})$
$C \leftarrow \{(\theta_I, \theta_F) \mid \sum_{L \in L_{pid}} T(L, S, \theta_I, \theta_F) \leq SLO\}$
if $C \neq \emptyset$ **then**
  $(\theta_I^*, \theta_F^*) \leftarrow \arg\min_{(\theta_I, \theta_F) \in C} \text{Cost}(\theta_I, \theta_F)$
else
  output "No configuration meets SLO."
end if

Algorithm 2 Configurator: β_{pid} Distribution

Input:

- $r_{\max}$: requests/sec per VM
- $\{\beta_{pid}\}$: exit-rate distribution
- $\{L_{pid}\}$: DNN layers
- $\theta_I^0,\ \theta_F^0$: base IaaS/FaaS configs (meet SLO)
- θ_I^{cand}: candidate IaaS configs for hybrid setup
- `cut_id`: offloading cuts
- SLO: max runtime per request
- cost models $C^I(\cdot),\ C^F(\cdot),\ C^H(\cdot)$

Output: Optimal $(\theta_I^*, \theta_F^*, \text{cut}_H)$

$cost_F \leftarrow C^F(\theta_F^0)$ // Cost using only FaaS
$cost_I \leftarrow C^I(\theta_I^0)$ // Cost using only IaaS
$cost_H, cut_H \leftarrow \infty, \infty$
$\theta_I^H \leftarrow$ NULL
for all $\theta \in \{\theta_I^{cand}\}$ **do**
  for all $cut \in \{cut\_id\}$ **do**
    $cost_H^{est} \leftarrow C^H(\theta, \theta_F^0, cut, \{\beta_{pid}\}, \{L_{pid}\}, r_{max})$
    if $cost_H^{est} < cost_H$ **then**
      $cost_H, \theta_I^H, cut_H \leftarrow cost_H^{est}, \theta, cut$
    end if
  end for
end for
if $cost_I \le \min(cost_F, cost_H)$ **then**
  $\theta_I^* \leftarrow \theta_I^0,\ \theta_F^* \leftarrow$ NULL, $cut_H \leftarrow$ NULL
else if $cost_F \le \min(cost_I, cost_H)$ **then**
  $\theta_I^* \leftarrow$ NULL, $\theta_F^* \leftarrow \theta_F^0,\ cut_H \leftarrow$ NULL
else
  $\theta_I^* \leftarrow \theta_I^H,\ \theta_F^* \leftarrow \theta_F^0$
end if

Algorithm 3 Online Load Balancer

Input:

- λ_t: req/s, observed at time t
- $r_{\max}$: requests/sec per instance

$\mu \leftarrow 0$ # EWMA mean of requests per second
$\sigma \leftarrow 0$ # EWMA std. dev.
for each timestamp t **do**
  $images \leftarrow$ DATALOADER(λ)
  $\mu \leftarrow (1-a)\,\mu + a\,\lambda_t$
  $\sigma \leftarrow (1-b)\,\sigma + b\,|\mu - \lambda_t|$
  $vmThres \leftarrow \mu + \phi \cdot \sigma$
  $VMCnt \leftarrow$ GET_HEALTHY_VM()
  Concurrently process mini-batches:
  IAAS(SPLIT($images,\ step = r_{\max}$)[: $VMCnt$])
  FAAS(SPLIT($images,\ step = r_{\max}$)[$VMCnt$:])
  if $time()$ mod `scaleInterval` $= 0$ **then**
    SCALEVMS($vmThres$)
  end if
end for

Algorithm 4 ScaleVMs Function

Input:
vmRequests: Number of VM requests
Output:
requiredVMs: Number of VMs

$requiredVMs \leftarrow \lfloor vmRequests / r\_max \rfloor$
if $vmRequests \mod r\_max > T\_CIP$ **then**
  $requiredVMs \leftarrow requiredVMs + 1$
end if
Update Auto Scaling Group to $requiredVMs$

4 EVALUATION

When submodels are independent, SERFLOW reduces to a classical load-balancing problem. Each request follows a fixed model sequence with deterministic runtime. In contrast, when submodels are dependent and support early exits, SERFLOW performs stage-aware resource provisioning by accounting for the exit distribution $\{\beta_{pid}\}$, thereby minimizing end-to-end cloud cost under strict SLOs. In this evaluation, we focus on the dependent submodel scenario.

4.1 Experimental Implementation and Setup

We evaluate SERFLOW through both offline profiling and online load balancing. Our experiments use a VGG-16 network with internal classifiers on the CIFAR-10 classification task (Krizhevsky & Hinton, 2009). Inference latency varies across requests exiting at different classifiers.

For offline profiling, we measure each VGG-16 partition's runtime on AWS EC2 (`c6i.large/xlarge`) and on AWS Lambda across memory configs. We also record EC2–Lambda transmission latency in the same AWS region via a public gateway. The profiling allows us to estimate the worst-case latency of running the full model and the expected runtime and cost under exit distributions $\{\beta_{pid}\}$.

For AWS Lambda (Fig. 12a), we measure cost at $conf\_thres = 0.85$ across memory sizes and cut indices (cut_id). The monetary cost is stable whenever memory is an integer multiple of 1769 MB, ensuring full vCPU allocation. Accordingly, we choose 8845 MB (5 vCPUs) to minimize resource contention and latency.

For online evaluation, we replay a sequence of numbers of requests from the WITS traffic trace (University of Waikato, 2020) for 400 epochs. We batch requests into 6-second SLO intervals in groups of 100 ($r_{\max} = 100/6$ req/s). Full batches are sent to available VMs. Any leftover (including partial) batches are offloaded to Lambda. We set the Traffic Cost Indifference Point $T_{\text{CIP}} = 90$, and every 25 epochs (150 s) we scale the VM count based on $\mu + \sigma$.

VMs are managed by an AWS Auto Scaling Group (disabled auto-scaling) and an Application Load Balancer. Serverless functions scale automatically. In hybrid offloading, EC2 instances send LZ4-compressed hidden variables via POST to Lambda when offloading late-exit requests, making transmission overhead negligible. VMs await Lambda responses before returning final logits to clients.

The online system (Load Balancer, Traffic Monitor, and Scaling Manager) executes on a node in Chameleon Cloud (Keahey et al., 2020), polling EC2 and Lambda metrics captured during function calls. The implementation of these modules is in Python and can be integrated with ML inference frameworks such as Ray Serve (Anyscale, Inc. and Ray Project contributors, 2025) on Kubernetes, treating each VM and serverless function as a Ray actor for elastic, large-scale deployment. We reserve production-grade integration for future work.

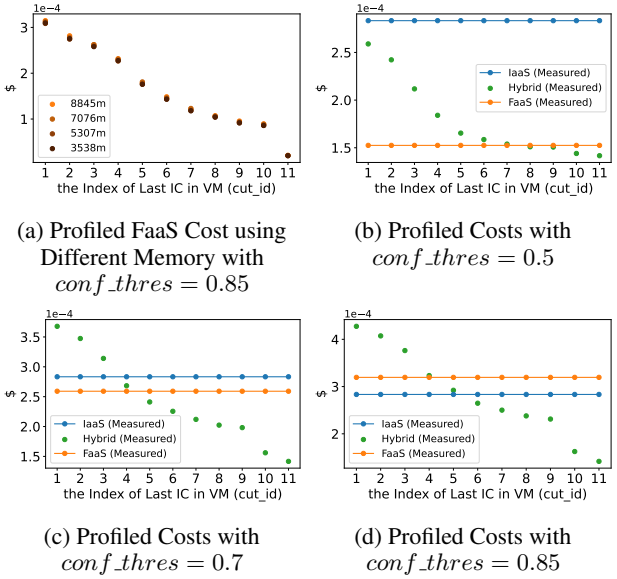
4.2 Offline Profiling and S_CIP Configuration

This section presents the offline profiling results. We first identify resource configurations that satisfy the 6-second SLO when all requests traverse every stage. As we discussed in the motivation example (Fig. 7 in Sec. 3.1), for $r_{\max} = \frac{100}{6}$ (100-requests mini-batches every 6 s), IaaS-only setup with a `c6i.xlarge` under $conf\_thres = 1$ completes each batch in 5.5 s, which is within the SLO of 6 seconds. In addition, we found that using a `c6i.large` takes 10.3 s, which would violate the SLO. FaaS-only setup using AWS Lambda with 8845 MB also meets the SLO in 4.3 s. A hybrid offloading setup that runs partitions 1–cut_id (with $cut\_id \leq 5$) on `c6i.large` and offloads the remainder to Lambda with 8845 MB also satisfies the 6-second SLO.

We profile cost and top-1 accuracy across confidence thresholds $conf\_thres \in \{0.5, 0.7, 0.85\}$. Figures 12b–12d plot cost with accuracies of 80%, 85%, and 86%, respectively. For reference, at $conf\_thres = 1.0$, our evaluation shows that accuracy reaches 87% (plots are not shown), indicating a minimal loss in accuracy. VM costs are flat since VMs run for the full SLO duration to avoid scaling overhead, while FaaS cost varies with execution time.

Each $conf\_thres$ induces a $\{\beta_{pid}\}$ distribution. Lower thresholds increase early exits at shallow ICs, whereas higher thresholds shift exits toward deeper ICs. When $conf\_thres = 0.5$, FaaS-only is the cheapest for any $cut\_id \leq 5$, so the online system only provisions {FaaS}. At $conf\_thres = 0.85$, IaaS-only is the cheapest for any $cut\_id \leq 5$, so the online system uses IaaS-only setup which provisions {IaaS, FaaS}, matching LIBRA's setting for stable runtimes. At $conf\_thres = 0.7$, the hybrid scheme is the cheapest at $cut\_id = 5$, so the pool of candidates is {Hybrid, FaaS}. We note that the Hybrid Offloading setup can only satisfy SLO when $cut\_id \leq 5$.

In general, increasing cut_id (i.e., assigning more partitions to VMs) reduces VM idle time and hybrid cost. Selecting the largest cut_id that still meets the SLO maximizes the cost efficiency. Empirical profiles closely match our estimates in Sec. 2.2.2, suggesting that profiling provides an accurate estimation for configurations. In the next section, we evaluate the online cost performance, focusing on $conf\_thres = 0.7$ and comparing SERFLOW to LIBRA.



(a) Profiled FaaS Cost using Different Memory with $conf\_thres = 0.85$

(b) Profiled Costs with $conf\_thres = 0.5$

(c) Profiled Costs with $conf\_thres = 0.7$

(d) Profiled Costs with $conf\_thres = 0.85$

Figure 12. Fig. 12a shows memory has little influence on FaaS resource costs. Thus, we set 8845 MB for all test cases. In Fig. 12b, 12c and 12d, we show the costs under different $conf\_thres$ of 0.5, 0.7 and 0.85, respectively. IaaS uses `c6i.xlarge`. Hybrid Offloading uses `c6i.large`. Lower thresholds shift exits to shallow ICs, making FaaS more cost-efficient. At $conf\_thres = 0.7$ and $cut\_id = 5$, hybrid beats FaaS by 8% and IaaS by 15%.

4.3 Online Load Balancing

We evaluate load balancing with $conf\_thres = 0.7$, comparing (i) FaaS only, (ii) IaaS+FaaS (Raza et al., 2021), and (iii) Hybrid Offloading+FaaS. We replay the first 400 epochs of the WITS CIFAR-10 classification trace (University of Waikato, 2020), sending requests every SLO interval (6 s).

Setup: Figure 13 shows, per epoch, the number of mini-batches in total (green line) and handled by FaaS (red bars) versus Hybrid Offloading (blue bars). For example, at epoch 50, there are 336 requests (3.36 mini-batches). With two healthy VMs, two full batches run on VMs and the remaining one full plus one 36-request batch run on Lambda.

We scale VMs every 25 epochs (150s) using $\mu + 1 \cdot \sigma$, where μ, σ are the EWMA and moving standard deviation of past load with weights $W^{\mu} = W^{\sigma} = 0.5$. To avoid overlapping scaling actions, and given VM cold starts may take up to 22

epochs (132 s), we choose a 25-epoch interval.

SERFLOW adapts to traffic dynamics. At epoch 25, it scales to 2 Hybrid Offloading instances, ready by epoch 44. It scales down to 1 instance at epoch 250 (ready by 252). And it scales to 2 instances at epoch 275 (ready by 293).

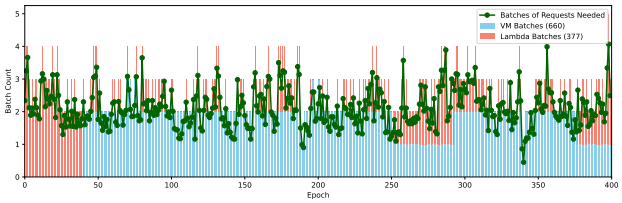


Figure 13. The number of mini-batches and how many go to Hybrid Offloading (blue) versus FaaS (red), for $conf\_thres = 0.7$.

Cost Analysis: Figure 14 compares the total cost of each resource setup over the 400-epoch trace. FaaS-only is 5% more expensive than Hybrid Offloading+FaaS and cannot adapt to sparsely activated models. And LIBRA's IaaS+FaaS setup is 20% more expensive than Hybrid Offloading+FaaS. All configurations meet the 6s SLO shown by profiling in the prior section. We evaluate cost with $conf\_thres = 0.7$. We use $T\_CIP = 90$ for Hybrid Offloading (Fig. 11), and $T\_CIP = 43$ for LIBRA (when $conf\_thres = 1.0$)[3].

To show VM under-utilization, we plot mean per-layer runtime for each setup in Fig. 15. As deeper layers see fewer requests and shorter runtimes, VMs sized for full-model throughput waste capacity. Thus, in Hybrid Offloading (Fig. 15a), we place partitions up to $cut\_id = 5$ onto VMs and offload the rest to FaaS. This stage-aware matching minimizes idle VM time and lowers overall cost, despite higher per-unit compute cost for deep partitions on FaaS.

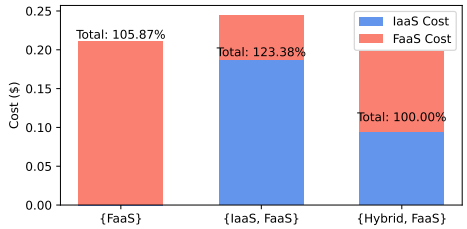
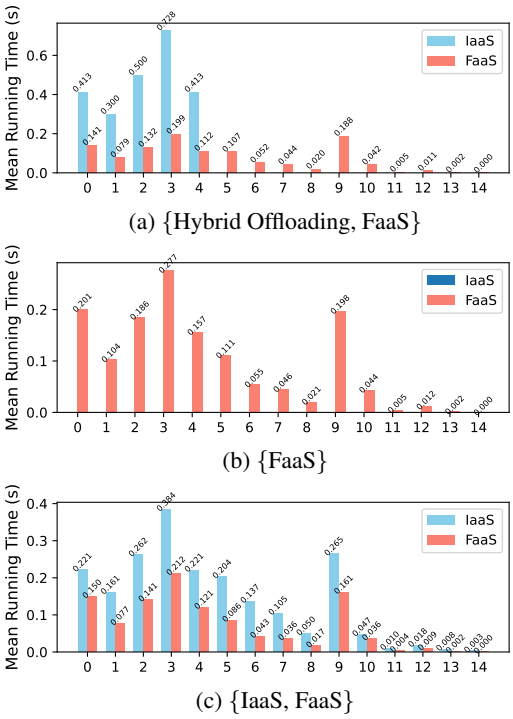


Figure 14. Total cost over 400 epochs for $\{\text{FaaS}\}$, $\{\text{IaaS}, \text{FaaS}\}$ (LIBRA), and $\{\text{Hybrid}, \text{FaaS}\}$, with $conf\_thres = 0.7$.

[3]The profiling steps are omitted as they are the same steps as shown in Fig. 11 with $conf\_thres = 0.7$.



(a) {Hybrid Offloading, FaaS}

(b) {FaaS}

(c) {IaaS, FaaS}

Figure 15. Average batch runtime per layer under each resource pool ($conf\_thres = 0.7$).

5 Conclusion and Future Work

SERFLOW provides an end-to-end optimization that allocates resources across multiple inference stages in deep neural networks. Rather than provisioning for the longest inference chain, it leverages serverless functions for transient workloads and virtual machines for stages with stable demand. This hybrid approach reduces cloud costs by over 20% compared to prior work that assumes uniform runtime.

We envision integrating SERFLOW into next-generation machine learning-as-a-service platforms, where it acts as a broker to provision cloud services (e.g., AWS Lambda and AWS EC2). The online system modules (Load Balancer, Traffic Monitor and Scaling Manager) can be wrapped in ML inference systems like Ray Serve (Anyscale, Inc. and Ray Project contributors, 2025) on Kubernetes to enable elastic large-scale deployment.

As for future work, SERFLOW can be extended for other machine learning workloads. In this work, the resource provisioning is based on the placements of internal classifiers in sequential networks. For models with high compute complexity (e.g., large language models), serverless functions may lack sufficient capacity to host even partial models, limiting cost efficiency. Future directions include exploring alternative compute platforms with different cost structures or adapting large-scale models for serverless execution.

REFERENCES

Amazon Auto Scaling. https://aws.amazon.com/ec2\/autoscaling/, 2018.

Google Auto Scaling. https://cloud.google.com/compute/\docs/autoscaler, 2018.

Amazon EC2. https://aws.amazon.com/ec2/, 2020.

Google Compute. https://cloud.google.com/compute, 2020.

Adobe. Cloud infrastructure overview. https://experienceleague.adobe.com/en/docs/commerce-operations/implementation-playbook/infrastructure/cloud/overview, 2023. Accessed: 2024-9-11.

Amazon Web Services. Adobe and aws partnership. https://aws.amazon.com/partners/adobe/, 2024. Accessed: 2024-9-11.

Amazon Web Services. Lambda Runtime Environment: Cold Start Latency. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html#cold-start-latency, 2025. Accessed: 2025-05-01.

Anyscale, Inc. and Ray Project contributors. Ray Serve: Scalable and Programmable Model Serving. https://docs.ray.io/en/latest/serve/index.html, 2025. Accessed: 2025-07-10.

AWS. Configure Lambda function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html, 2024. Accessed: 2024-10-23.

Behera, A. P., Champati, J. P., Morabito, R., Tarkoma, S., and Gross, J. Towards Efficient Multi-LLM Inference: Characterization and Analysis of LLM Routing and Hierarchical Techniques. *arXiv preprint arXiv:2506.06579*, 2025. URL https://arxiv.org/abs/2506.06579. Accessed: 2025-07-10.

Campbell Webb. Unleashing the power of innovation with the public cloud. https://blog.workday.com/en-us/unleashing-the-power-innovation-with-public-cloud.html, 2024. Accessed: 2024-9-11.

Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. The rise of serverless computing. *Communications of the ACM*, 2019.

Feng, S., Wang, Z., Goyal, P., Wang, Y., Shi, W., Xia, H., Palangi, H., Zettlemoyer, L., Tsvetkov, Y., Lee, C.-Y., and Pfister, T. Heterogeneous Swarms: Jointly Optimizing Model Roles and Weights for Multi-LLM Systems. *arXiv preprint arXiv:2502.04510*, 2025. URL https://arxiv.org/abs/2502.04510. Accessed: 2025-07-10.

Gross, S., Ranzato, M., and Szlam, A. Hard Mixtures of Experts for Large Scale Weakly Supervised Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 6865–6873, 2017.

Gunasekaran, J. R., Thinakaran, P., Kandemir, M. T., Urgaonkar, B., Kesidis, G., and Das, C. Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 199–208, 2019. doi: 10.1109/CLOUD.2019.00043.

Kaya, Y., Hong, S., and Dumitras, T. Shallow-deep networks: Understanding and mitigating network overthinking. In *International conference on machine learning*, pp. 3301–3310. PMLR, 2019.

Keahey, K., Anderson, J., Zhen, Z., Riteau, P., Ruth, P., Stanzione, D., Cevik, M., Colleran, J., Gunawi, H. S., Hammock, C., Mambretti, J., Barnes, A., Halbach, F., Rocha, A., and Stubbs, J. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

Krizhevsky, A. and Hinton, G. Learning Multiple Layers of Features from Tiny Images. Technical Report 0, University of Toronto, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Lewis, M., Bhosale, S., Dettmers, T., Goyal, N., and Zettlemoyer, L. Base layers: Simplifying training of large, sparse models. In *International Conference on Machine Learning*, pp. 6265–6274. PMLR, 2021.

Little, J. D. A proof for the queuing formula: L= λ w. *Operations research*, 9(3):383–387, 1961.

Mudvari, A., Jiang, Y., and Tassiulas, L. SplitLLM: Collaborative Inference of LLMs for Model Placement and Throughput Optimization. *arXiv preprint arXiv:2410.10759*, 2024. URL https://arxiv.org/abs/2410.10759. Accessed: 2025-07-10.

Novak, J. H., Kasera, S. K., and Stutsman, R. Cloud Functions for Fast and Robust Resource Auto-Scaling. In *COMSNETS*, 2019.

Purandare, S., Wasay, A., and Idreos, S. μ-two: 3$\times$ faster multi-model training with orchestration and memory optimization. *Proceedings of Machine Learning and Systems*, 5:541–562, 2023.

Raza, A., Zhang, Z., Akhtar, N., Isahagian, V., and Matta, I. LIBRA: An Economical Hybrid Approach for Cloud Applications with Strict SLAs. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 136–146, 2021. doi: 10.1109/IC2E52221.2021.00028.

Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. 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. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shahrad.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

Shen, Z., Lang, H., Wang, B., Kim, Y., and Sontag, D. Learning to decode collaboratively with multiple language models. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12974–12990, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.701. URL https://aclanthology.org/2024.acl-long.701/.

Teerapittayanon, S., McDanel, B., and Kung, H. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469, 2016. doi: 10.1109/ICPR.2016.7900006.

University of Waikato. WITS: Waikato Internet Traffic Storage HTTP Dataset. https://wand.net.nz/wits/catalogue.php, 2020. Accessed: 2025-07-10.

Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, Boston, MA, 2018. USENIX Association.

Zhang, C., Yu, M., Wang, W., and Yan, F. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *USENIX ATC*, Renton, WA, 2019.