Trace-Driven Implementation and Evaluation of ESFF Scheduling on Azure Serverless Workloads

Maakhish Sai, Krishna Harsha, Mohith Sri Vaishnav
Department of Computer Science and Engineering, NIT Calicut
punnam_b221135cs@nitc.ac.in , krishna_b220042cs@nitc.ac.in, sikhakolli_b221224cs@nitc.ac.in

***Abstract*—Serverless computing on edge servers must operate under strict resource constraints and non-negligible cold-start overheads. The Efficient Serverless Function Scheduling in Edge Computing (ESFF) algorithm was proposed to minimize average response time by making instance creation and replacement decisions based on function-specific execution time and latencies. In this work, We implement ESFF as a discrete-event simulator and simulate it using a real Azure Functions trace dataset. We show the complete work starting from problem definition till the end. We also outline two possible extensions: a Priority ESFF (ESFF-P) and a Fairness ESFF (ESFF-F), which aim to incorporate priorities and per-function slowdown fairness into ESFF's weight-based decisions.**

***Index Terms*—Serverless computing, edge computing, scheduling, ESFF, Azure Functions, QoS, fairness.**

I. Introduction

Serverless computing, has become a popular execution model for cloud applications. Developers write stateless functions that are triggered by events, while the platform automatically manages provisioning, and other tasks. Public cloud providers such as AWS Lambda, Azure Functions, and Google Cloud Functions offer virtually elastic scaling, but this assumption does not hold at the edge.

Edge servers typically have limited CPU cores, memory, and storage. They cannot indefinitely scale the number of concurrent function instances, and must carefully decide which functions to keep warm, which to evict, and when to create new instances. Cold starts and evictions incur non-negligible delays, which directly impact user-experience.

The Efficient Serverless Function Scheduling in Edge Computing (ESFF) algorithm was recently proposed to address this challenge. ESFF models per-function execution time, cold-start time, and eviction cost, and uses this information in a Function Creation Policy (FCP) and a Function Replacement Policy (FRP). Together, these policies determine how many instances of each function should exist and which instances to replace when the edge server is full, with the aim of minimizing average response time. In this work, We :

- Implement a simulator for ESFF.
- Preprocess a real Azure Functions trace into a format suitable for simulation (arrival time, function id, execution time).
- Run experiments to evaluate ESFF under different edge server capacities and workload intensity ratios.
- Measure average response time, average slowdown, and average cold-start time per request.
- Propose two extensions on top of ESFF: Priority ESFF (ESFF-P) and Fairness ESFF (ESFF-F).

II. Related Work

A. Serverless Computing and Cold Starts

Serverless platforms simplify deployment and scaling for cloud applications, but the pay-per-use model introduces cold-start latency when new function instances are initialized. A large body of work focuses on how to decrease cold-start latencies in the cloud, including pre-warming strategies, resource prediction, and function pooling. These approaches generally assume large-scale backends and focus on reducing cold-start impact for individual functions rather than jointly optimizing scheduling across many functions.

B. Edge Computing and Resource Constraints

Edge computing moves computation closer to end users in order to reduce latency. However, edge nodes are resource-constrained: they can host only a small number of concurrent function instances. Traditional autoscaling techniques that work in data centers cannot be directly applied at the edge, because oversubscribing resources can lead to severe contention and unacceptable latencies. Scheduling and placement decisions at the edge must therefore carefully consider limited capacity.

C. Serverless Scheduling on Edge Servers

Recent work proposes specialized scheduling algorithms for serverless workloads on edge servers. ESFF (Efficient Serverless Function Scheduling) is one such algorithm. ESFF explicitly models:

- The average execution time of each function
- It's cold-start time
- It's eviction time

ESFF derives an optimal ordering for a simplified single-server scheduling model and then generalizes the idea into two online components: **FCP**, which decides when to create or replace instances on request arrivals, and **FRP**, which decides how to reassign instances on request completions.

Other approaches (e.g., OpenWhisk variants, simple LRU caching of function containers, or random eviction) do not exploit function-specific cost parameters as explicitly as ESFF, and thus may perform poorly when capacity is tight and workloads are skewed. My work focuses on implementing ESFF and evaluating it on real traces, then extending it with QoS and fairness considerations.

III. System Model

We consider a single edge server that executes serverless functions. The server has a capacity of C instances, meaning at most C function instances can be active (warm or cold) at any point in time.

A. Requests and Functions

Let R denote the set of all requests in the workload. Each request $r_i \in R$ is characterized by an arrival time t_i^a, a completion time t_i^c, an execution time t_i^e, an associated function f_j. The response time of a request is $t_i^c - t_i^a$. The set of functions is denoted by $\mathcal{F} = \{f_1, f_2, \dots\}$. Each function f_j has an average execution time t_j^e, a cold-start time t_j^l, an eviction time t_j^v.

B. Function Instances and Queues

For each function f_j, the edge server maintains a queue q_j of waiting requests, a set K_j of instances of that function. Each instance $k_j^o \in K_j$ is either *idle*, waiting for a request, or *busy*, currently executing a request.

An idle instance can be evicted to free capacity (paying eviction time t_j^v). Creating a new instance of f_j incurs a cold-start delay t_j^l before it can execute requests.

C. Capacity Constraint

At any time, the total number of instances across all functions must respect:

$$\sum_j |K_j| \le C. \tag{1}$$

IV. Problem Formulation

The key performance objective is to minimize the *average response time* across all requests:

$$\bar{T} = \frac{1}{|R|} \sum_{i \in R} \big(t_i^c - t_i^a\big). \tag{2}$$

In addition, it is useful to define the *slowdown* of a request:

$$\text{slowdown}_i = \frac{t_i^c - t_i^a}{t_i^e}. \tag{3}$$

Slowdown compares the observed response time to the ideal execution time with no waiting, and helps capture how badly a request is delayed.

For a given workload, capacity C, and function parameters (t_j^e, t_j^l, t_j^v), the scheduling problem is to decide:

- When to create new instances of each function.
- Which idle instances to evict when capacity is full.
- How to reuse instances upon request completions.

so as to minimize $\bar{T}$ (and, implicitly, to keep slowdowns low) while respecting the capacity constraint. ESFF provides one such scheduling policy.

V. ESFF Algorithm

This section summarizes the ESFF scheduling algorithm as implemented in my simulator.

A. Simplified Single-Server Insight

In a simplified setting also called OSSFS with a single server, known workloads, and identical execution times per function, ESFF derives an optimal ordering based on a function-level weight:

$$w_j = t_j^e + \frac{t_j^l + t_j^v}{n_j}, \tag{4}$$

where n_j is the number of requests belonging to function f_j. Executing functions in order of increasing w_j (and processing all requests of a function together) minimizes average response time in that simplified model.

This idea motivates the more general online ESFF policies for multi-instance, trace-driven scenarios.

B. Function Creation Policy (FCP)

When a request r_i for function f_j arrives at the edge server, ESFF applies the Function Creation Policy.

1) **Immediate use of idle instance**: If the queue q_j is empty and there exists an idle instance of f_j in K_j, r_i is directly dispatched to that idle instance without additional delay.
2) **Capacity not full**: If the total number of instances $\sum_j |K_j| < C$, ESFF estimates whether creating a new instance of f_j is beneficial using

$$n_j^e = n_j^w + 1 - \frac{\bar{t_j^l} \cdot |K_j|}{\bar{t_j^e}}, \tag{5}$$

where n_j^w is the current number of waiting requests of f_j. If $n_j^e > 0$, this indicates that one or more requests will still remain waiting even after existing instances work during the cold-start period, so ESFF creates a new instance (cold start). Otherwise, r_i is enqueued in q_j.
3) **Capacity full**: If there is no idle function instance of f_j and the resources are insufficient to initialize such instance, it assesses whether the total response time can benefit from replacing an idle function instance.The number of f_j's waiting requests after f_j's eviction time and f_j's cold start latency is estimated by:

$$n_{j,j'}^e = n_j^w + 1 - \frac{(\overline{t_j^l} + \overline{t_j^v}) \left|\mathbf{K}^j\right|}{\overline{t_j^e}} \tag{6}$$

The candidate set of functions with enough eviction time that makes $n_{j',j}^e$ is computed by:

$$\mathbf{S} = \left\{ f_{j'} \,|\, n_{j,j'}^e > 0 \text{ and } \sum_{k_o^{j'} \in K^{j'}} \text{state}(k_o^{j'}) > 0 \right\} \tag{7}$$

FCP chooses $f_j \in \mathbf{S}$ with the largest t_j^e, and replaces an idle instance of f_j with a new one. If $\mathbf{S}$ is empty, no function instance is initialized.

C. Function Replacement Policy (FRP)

When an instance of f_j completes a request, ESFF invokes the Function Replacement Policy to decide whether to keep this instance as f_j or reassign it to another function.

1) **Compute weight** w_j: For f_j, if $n_j^w > 0$ and $|K_j| > 0$, ESFF computes

$$w_j = t_j^e + \frac{\bar{t_j^v}|K_j|}{n_j^w}, \tag{8}$$

Otherwise, w_j is treated as a large value.
2) **Collect candidates**: ESFF builds a set of candidate functions $S = \{f_{j'} \mid n_{j'}^w > 0\}$ that have non-empty queues.
3) **Estimate benefit for each candidate**: For each $f_{j'} \in S$, ESFF estimates how many of its requests remain after evicting k_o^j and initializing a new instance of f_j, yielding $n_{j',j}^e$, and computes a candidate weight

$$w_{j'} = \bar{t_{j'}^e} + \frac{(\bar{t_{j'}^l} + \bar{t_{j'}^v})(|K_j| + 1)}{n_{j',j}^e}. \tag{9}$$

The candidate set of functions with enough eviction time that makes $n_{j',j}^e > 0$ is computed by:
4) **Choose minimum weight**: ESFF compares w_j with all $w_{j'}$ and selects the function with minimum weight.
   - If some $w_{j'} < w_j$, ESFF replaces this instance with a new instance of $f_{j'}$ (incurring eviction and cold-start times).
   - Otherwise, ESFF keeps the instance as f_j: if q_j is non-empty, the next request in q_j is served; if q_j is empty, the instance becomes idle.

This combination of FCP (on arrivals) and FRP (on completions) allows ESFF to dynamically adapt the instance mix based on observed queues and function-specific costs.

VI. Results

A. Experimental Setup

We use the Azure trace requests as input to the simulator.

- Function parameters are estimated from the trace (for t_j^e) and sampled for t_j^l and t_j^v,
- The default capacity is set to $C = 16$,
- The simulator processes all requests in chronological order.

We evaluate ESFF under two axes:

1) Varying edge server capacity C with fixed workload,
2) Varying workload intensity via scaling of arrival times.

B. Varying Edge Server Capacity

To study the impact of capacity, We fix the workload (same arrival and execution times from the trace) and run ESFF with $C \in \{4, 8, 12, 16, 20, 24, 28, 32\}$.

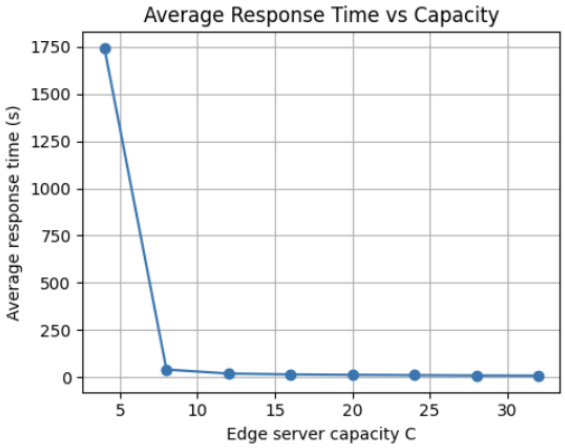

Fig. 1. Impact of capacity (C) on Avg Response Time.

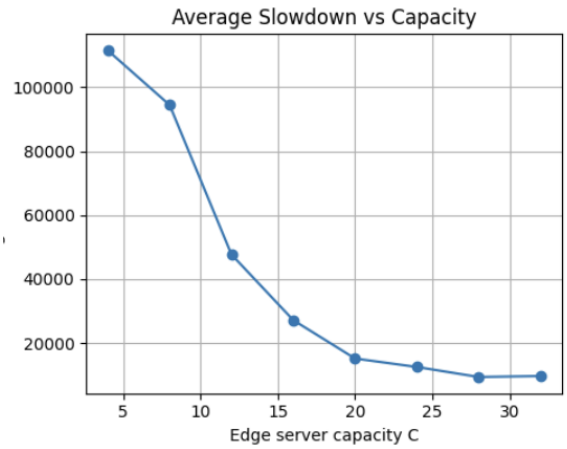

Fig. 2. Impact of capacity (C) on Avg Slowdown.

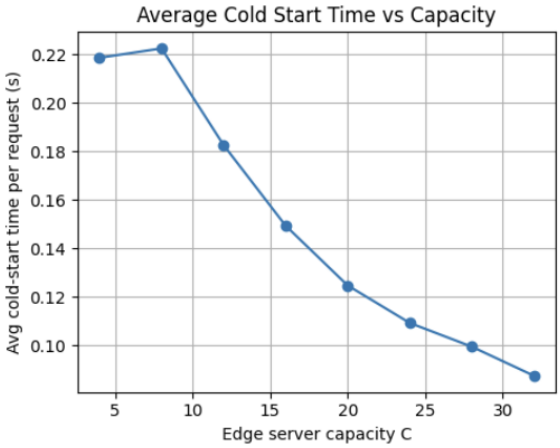

Fig. 3. Impact of capacity (C) on Avg Cold Start Time.

As shown in Fig. 1, as C increases, average response time decreases because more instances reduce queuing. Similarly, Fig. 2 shows that average slowdown decreases with higher capacity. And Fig. 3 shows that the average cold start time decreases with higher capacity.

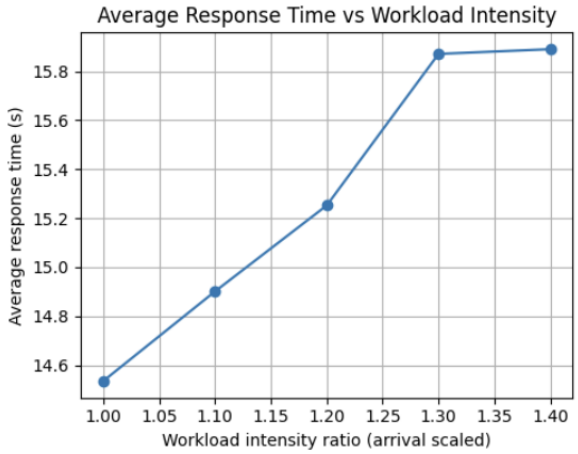

Fig. 4. Impact of Workload Intensity on Avg Response Time.

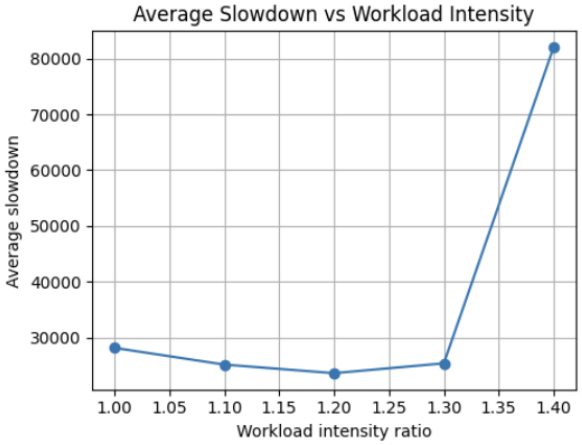

Fig. 5. Impact of Workload Intensity on Avg Slowdown.

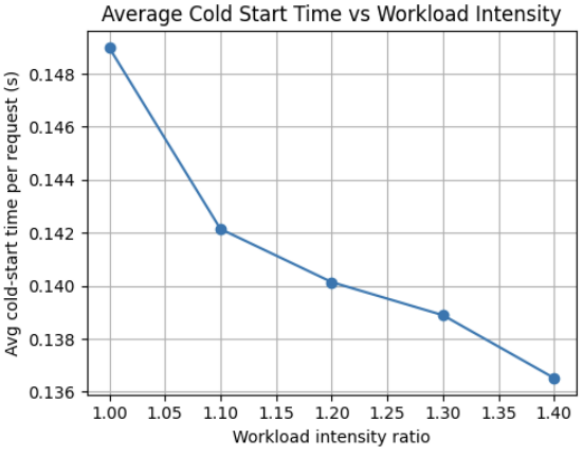

Fig. 6. Impact of Workload Intensity on Avg Cold Start Time.

Workload Intensity is the number of requests arriving per second. As more requests arrive, the scheduler and serverless functions become busy. So tasks likely have to wait longer in queues and increases the response time as shown in Fig. 4. The intensity of 1.3-1.4 is considered a high workload so for this amount of load, the scheduler hits a saturation point, which resulted in spike in slowdown as showin in Fig. 5. When workload intensity is high, this means that more requests are in queue. So the scheduler in order to reduce the average response time (which is affected by cold start time, eviction time) and to keep the containers warm, requests from same function which are present in queue will be executed. This results in decrease of average cold start time as shown in Fig. 5.

C. Discussion

The results from these experiments show that the ESFF implementation behaves sensibly under realistic trace-driven workloads. The capacity sweep reveals how sensitive ESFF is to the number of available instances, while the workload intensity sweep shows how ESFF scales under varying load. These baselines also serve as reference points for evaluating extended versions of ESFF that incorporate additional objectives such as priority/QoS and fairness.

VII. Future Work

Based on the baseline ESFF implementation and evaluation, We propose two main directions for future work: a Priority/QoS-aware ESFF (ESFF-P) and a Fairness-aware ESFF (ESFF-F). Both can be implemented as modifications of ESFF's weight computations, without completely changing the discrete-event simulation framework.

A. Priority / QoS–Aware ESFF (ESFF-P)

In the current ESFF formulation, all functions are treated equally from the perspective of scheduling. In practice, however, some functions are more important than others. For example, user-facing APIs may be latency-sensitive, whereas background analytics or batch processing can tolerate higher delays. ESFF-P extends ESFF by associating a *priority* with each function and using it into the weight.

1) Priority Model: Each function f_j is assigned a priority $p_j \in (0, 1]$ or a discrete priority class that can be normalized into this range. For instance:

- interactive APIs: $p_j = 1.0$,
- batch/background functions: $p_j = 0.5$.

Higher p_j means higher importance.

2) Priority-Aware Weight: Recall that, in the simplified single-server analysis, ESFF uses:

$$w_j = t_j^e + \frac{t_j^l + t_j^v}{n_j}, \tag{10}$$

where n_j is the number of requests of function f_j. More generally, in the online ESFF policies, function weights w_j and $w_{j'}$ are computed in FRP and, implicitly, in FCP decisions.

In ESFF-P, We define a priority-aware weight:

$$w_j^{(P)} = \frac{w_j}{p_j}. \tag{11}$$

This has the following effect:

- if p_j is large (high-priority function), $w_j^{(P)}$ decreases, so the scheduler is more likely to favor f_j;

- if p_j is small (low-priority function), $w_j^{(P)}$ increases, so f_j is delayed relative to high-priority functions.

In the implementation, wherever ESFF computes a weight w_j or $w_{j'}$ (e.g., in FRP), We can wrap it as:

$$w_{\text{eff}} = \frac{w_{\text{plain}}}{p_j}, \tag{12}$$

and use w_{eff} for comparisons and minimum-weight selection.

3) Evaluation Plan for ESFF-P: To evaluate ESFF-P, We plan to

- group functions into priority classes (e.g., high and low),
- measure average response time and slowdown *per priority class* under baseline ESFF and ESFF-P,
- quantify how much high-priority functions benefit in terms of latency,
- quantify the corresponding impact on low-priority functions.

This extension is novel compared to the original ESFF paper and is relatively straightforward to implement in the simulator.

B. Fairness-Aware ESFF (ESFF-F)

While ESFF focuses on minimizing global average response time, it may cause some functions to experience high slowdowns if they are consistently deprioritized by weight-based decisions. ESFF-F introduces a fairness component by tracking the per-function slowdown and adjusting weights to reduce extreme imbalances.

1) Per-Function Slowdown Tracking: For each function f_j, We can track a running average slowdown over its completed requests:

$$\overline{S}_j = \frac{1}{|R_j|} \sum_{i \in R_j} \frac{t_i^c - t_i^a}{t_i^e}, \tag{13}$$

where R_j is the set of requests belonging to f_j. We also maintain the global average slowdown $\overline{S}_{\text{global}}$ across all functions.

2) Fairness Factor: We then define a fairness factor F_j:

$$F_j = 1 + \alpha \left(\frac{\overline{S}_j}{\overline{S}_{\text{global}}} - 1 \right), \tag{14}$$

where $\alpha \geq 0$ is a tunable fairness parameter.

Intuitively:

- if $\overline{S}_j > \overline{S}_{\text{global}}$, function f_j is having more slowdown than average; thus $F_j > 1$, and we should effectively reduce its weight to give it more chance to acquire instances,
- if $\overline{S}_j < \overline{S}_{\text{global}}$, function f_j is doing better than average; thus $F_j < 1$, and its weight can be slightly increased, allowing other functions to catch up.

3) Fairness-Aware Weight: In ESFF-F, the effective weight becomes:

$$w_j^{(F)} = w_j \cdot F_j. \tag{15}$$

In implementation, after computing the baseline weight w_j in FRP, We apply:

$$w_{\text{eff}} = w_{\text{plain}} \cdot F_j, \tag{16}$$

and use w_{eff} in all comparisons. This will show bias in favor of functions that have experienced high slowdown.

4) Evaluation Plan for ESFF-F: For ESFF-F, We intend to:

- measure the distribution of per-function slowdowns under ESFF and ESFF-F,
- examine reduction in extreme slowdown outliers,
- quantify the impact on global average response time and slowdown.

The goal is to show that ESFF-F can achieve a better balance (fairness) across functions with a modest increase in global latency, which is not directly addressed in the original ESFF algorithm.

VIII. Conclusion

In this work, We implemented the ESFF scheduling algorithm as a discrete-event simulator and applied it to a real Azure Functions trace. We described the system model, the problem formulation, and the ESFF Function Creation and Function Replacement Policies. We then detailed the implementation, trace preprocessing, and metrics computation, and sketched results for varying edge server capacity and workload intensity ratios, focusing on average response time, slowdown, and cold-start behavior.

Beyond implementing ESFF, We proposed two extensions: Priority/QoS-aware ESFF (ESFF-P), which incorporates per-function priorities into ESFF's weight-based decisions, and Fairness-aware ESFF (ESFF-F), which uses historical per-function slowdown to balance performance across functions. Implementing and evaluating these extensions on top of the existing simulator is a clear next step and can provide novel insights into QoS and fairness in serverless scheduling at the edge.