UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms

Ghazal Sadeghian Mohamed Elsakhawy Mohanna Shahrad[†] Joe Hattori[‡] Mohammad Shahrad

University of British Columbia *McGill University*[†] *University of Tokyo*[‡]

Abstract

We present UnFaaSener, a lightweight framework that enables serverless users to reduce their bills by harvesting non-serverless compute resources such as their VMs, on-premise servers, or personal computers. UnFaaSener is not a new serverless platform, nor does it require any support from today's production serverless platforms. It uses existing pub/sub services as the glue between the serverless application and offloading hosts. UnFaaSener's asynchronous scheduler takes into consideration the projected resource availability of the offloading hosts, various latency and cost components of serverless versus offloaded execution, the structure of the serverless application, and the developer's QoS expectations to find the most optimal offloading decisions. These decisions are then stored to be retrieved and propagated through the execution flow of the serverless application. The system supports partial offloading at the resolution of each function and utilizes several design choices to establish confidence and adaptiveness. We evaluate the effectiveness of UnFaaSener for serverless applications with various structures. UnFaaSener was able to deliver cost savings of up to 89.8% based on the invocation pattern and the structure of the application, when we limited the offloading cap to 90% in our experiments.

1 Introduction

Serverless computing [54] allows developers to quickly build scalable, event-driven applications and pay for only what they use. It also removes the provisioning and maintenance burdens of the traditional cloud system. Developers have identified and embraced this game-changing cloud paradigm. According to Datadog's June 2022 analysis of cloud user telemetry [33], more than 70% of organizations using AWS and over 50% of Azure and Google Cloud users have adopted serverless offerings. A year prior, the serverless adoption numbers for these three leading cloud providers were just above 50%, 35%, and 20%, respectively [32].

In addition to increased adoption, serverless applications are also becoming increasingly complex. In 2019, the majority of serverless applications were composed of just one function and 80% had three or fewer functions [72]. Today, complex serverless workflows are no more rare. A recent study of open-source serverless projects has identified 31% of studied applications to have workflow structures [35]. From 2019 to 2022 the popularity of serverless DAGs has grown by 6× at Azure [59]. The increased complexity of serverless applications can be attributed to the maturation of serverless offerings and the increased proficiency of developers.

With a developer-focused perspective, this work is motivated by a relatively simple question: *why should serverless functions be bound to be executed within the serverless platforms?* If serverless functions are designed to be primarily stateless, the serverless model disaggregates storage from compute, the serverless isolation/virtualization mechanisms are lightweight [17], and the model is event-driven, it begs asking whether offloading execution of serverless functions off the serverless platform can make economic sense.

Many organizations and development teams use serverless offerings in conjunction with other cloud service types, such as VMs or microservices' containers [24, 36]. Reports from different cloud providers indicate that the majority of VMs in public clouds are heavily under-utilized [30, 44]. Despite low VM utilization, public cloud providers have been able to improve data center efficiency using advanced resource oversubscription to co-locate many underutilized VMs with predictable guarantees [30, 48, 55], or through dynamic capacity harvesting from those underutilized VMs to sell to others [39, 74, 80]. Such strategies help the provider operate at higher efficiency, but cloud users still have to pay for their full static allocations. If a team is already paying an hourly rate for renting a VM and that VM is not fully utilized, it could be harnessed to run their own serverless functions. Additionally, an organization may have on-premise computational capacity that already incurs capital and operating costs, which can similarly be leveraged to execute migratable serverless functions.

The merits of the proposed serverless function offloading are clear, but determining when to migrate functions depends on various factors. For example, offloading one or more functions in a latency-sensitive chain of short functions could lead to QoS violations due to added network latency. In contrast, for a serverless DAG with imbalanced branches, offloading those executions off the critical path may be worthwhile if the cost of data movement and added latency are acceptable. While these examples focus on latency, some serverless applications, such as nightly builds, may have little to no latency requirements, making offloading more viable. Ultimately, how and when functions of a serverless application can be offloaded depends on a long list of factors. Serverless providers do not have an incentive to enable such functionality as it would negatively impact their profitability, and developers would rather not deal with the complexity as it goes against the serverless philosophy of freeing them from provisioning concerns. We believe that there is real value to be delivered in this junction by providing developers with a system that adaptively and transparently offloads their serverless functions to their own alternative execution hosts.

We design and build UnFaaSener, **the first holistic serverless offloading system without any change to today's serverless platforms**. This system performs adaptive offloading of a developer's functions to their own alternative hosts. UnFaaSener does this by dynamically considering latency and cost implications of offloading as well as resource availability predictions on hosts against goals conveyed by the developer (e.g., saving maximum cost, or respecting a certain latency QoS). We build UnFaaSener to use existing services on a popular serverless platform, and run various serverless applications on it. It is available at https://github.com/ubc-cirrus-lab/unfaasener.

2 Background

2.1 The Status Quo

Execution of serverless functions. Your serverless functions run within the serverless platform you operate on. Depending on the provider, your functions might be allocated to run in lightweight VMs, containers, or other isolation abstractions that themselves use dedicated allocations or internally harvested resources [86]. Interestingly, your functions will not be allocated to any underutilized VMs that you already pay for. If you have computational resources on a different cloud or on-premise, those are not used to host your functions either. If a developer decides to tap into these capacities to reduce their serverless bill, they need to build their applications differently and effectively do resource provisioning. This defeats the purpose of using serverless in the first place.

Offloading to and from serverless. Serverless's unparalleled horizontal scaling and pay-per-use pricing model enables cheap acceleration of bursty, massively parallel workloads. Researchers have developed general purpose (e.g., gg [37]) and domain-specific frameworks (e.g., ExCamera [38] and NumPyWren [73]) for this purpose. Offloading to serverless is popular for edge [26, 52, 82] and network function virtualization (NFV) [16, 70, 85] applications. Researchers have also proposed offloading from serverless to the edge [41].

The idea of offloading from VMs to serverless has also been proposed. Most of these works utilize serverless as a backup when scaling out VMs [43, 47, 63, 84, 87], however, some others simultaneously offload a small portion of traffic to serverless [67]. As VMs are the primary deployment in these works, the benefits of serverless functions, such as high scalability can not be fully exploited, and the scope of the applications is limited to the capacity of the VMs. To fully exploit serverless advantages, researchers have suggested hybrid VM-serverless deployment of applications [56, 75]. In these systems, however, a secondary custom scheduler is added before the serverless scheduler, which limits the scalability of the system and comes with security concerns.

The systems mentioned earlier are not designed for resource harvesting. However, a number of works have proposed modifying the serverless platform to offload serverless on the harvested resources [78, 83, 86]. We identify this as a limitation, as one cannot expect serverless providers to change their platforms and reduce their profitability. Besides that, the offloading will be limited to hosts located only within the scope of the platform scheduler (same cluster, region, or zone of the same cloud, depending on the provider's architecture). UnFaaSener is designed to work with existing serverless platforms, without requiring any change. By harvesting the idle resources of any host within or outside the cloud hosting serverless functions, UnFaaSener opportunistically achieves cost reductions for a wide range of general-purpose applications, from single functions to applications in a form of complex DAGs (consisting of multiple branches, merging points, and dynamic fan-out patterns).

2.2 The Serverless Cost Model

Understanding the serverless cost model is imperative for building a mental model of how UnFaaSener offers cost savings. We provide a summary here and refer the reader to related work for more detailed descriptions [57].

Capacity cost: Developers are required to set the memory size of their serverless functions. This indirectly sets the CPU share, too, as the CPU-to-memory allocation ratio is fixed in current serverless systems [21]. Multiplying the execution time by the configured memory size determines the GB-seconds usage. There is a cost charged per GB-seconds; e.g., $\$1.67 \times 10^{-5}$ for AWS Lambda on x86 [3]. Some providers enforce a minimum execution time per invocation (e.g., 100 ms for Azure Functions). Execution times are also rounded-up; e.g., to the nearest 1 ms for Azure Functions and to the nearest 100 ms for Google Cloud Functions.

Invocation cost: Each invocation also incurs a cost; e.g., $0.2 and $0.4 per million requests for Azure Functions and Google Cloud Functions, respectively.

Free tier: Typically, serverless providers offer monthly free tiers: AWS, Azure, and Google Cloud provide 400,000 GB-s per month. The free tier also includes no invocation cost for the first 1 million requests in AWS Lambda and Azure functions, and the first 2 million for Google Cloud Functions. **Saving cost by offloading:** If a developer's serverless usage is low enough to fit within the monthly free tier offered by cloud providers, they should not run UnFaaSener. Beyond that, offloading functions to alternative hosts will eliminate the capacity and invocation costs associated with it.

3 UnFaaSener Design Challenges

Let us decompose the sub-problems that need to be solved to enable adaptive offloading of serverless functions to achieve maximum cost reduction with minimum impact on latency, and with no provider support:

1. ***Enabling flexible offloading*** (§4): The very first requirement is to enable partial offloading of requests for each function of a serverless application to arbitrary hosts. The solution should support various serverless applications, ranging from those with a single function to complex workflows (typically DAGs).
2. ***Asynchronous Scheduler*** (§5): To deliver a practical solution, UnFaaSener's design should satisfy the following:

(a) The system should work with no scheduling or load balancing support from the provider.

(b) As offloading is opportunistic, serverless execution should be the default case to ensure high scalability and low latency for the common case.

(c) The serverless platform should remain the end-point for the incoming traffic to the application to enforce network-based access control rules.

(d) Given that invocation patterns can be sporadic [72], continuously performing scheduling tasks on a dedicated VM/container is not an option, as the incurred cost can easily exceed the cost savings of function offloading.

We need a scheduling mechanism that resides outside the serverless platform, is activated when necessary, invokes the solver if needed for new decisions, and reflects the offloading decisions to be used by functions.

3. ***Determining optimal offloading*** (§6): Optimal offloading decisions depend on various factors such as application structure, function resource requirements, execution times, invocation patterns, host resource availability, communication latencies and costs. Additionally, the diverse range of developers using public clouds and the variety of serverless applications lead to different notions of optimality. Optimizing for latency vs. cost, mean vs. tail, etc., should all be expressible. A solver is required to take these inputs and determine the host(s) for function execution.
4. ***Monitoring and prediction of hosts' resource availabilities*** (§7.1, §7.2): Considering the delays associated with the distributed nature of UnFaaSener's multi-node, or even multi-cloud deployment, each host's resource availability should be forecasted with a high degree of confidence.

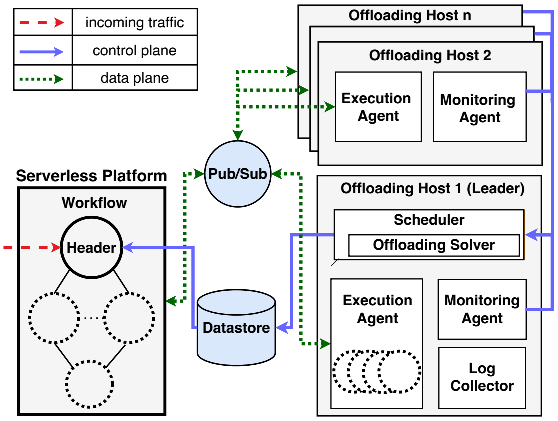


Figure 1: UnFaaSener's system diagram.

5. ***Managing execution-related tasks on the host side*** (§7.3): Each host is responsible for various mechanical tasks, such as pulling the function on a cold start, setting up an appropriate execution environment, enforcing resource limits, executing the function, implementing keep-alive policies, queuing incoming requests, and invoking the next function on the appropriate host. While this aspect of the system may not be particularly novel in terms of research, it is crucial for the overall performance of the system.

Figure 1 provides an overview of UnFaaSener's different system components. In the next sections, we describe how the design challenges stated earlier shaped our design decisions. In total, UnFaaSener includes ~6.3 K-SLOC: 5.4K lines of Python, 0.8K lines of C++, and less than 100 lines of Shell.

4 Enabling Flexible Offloading

There is no universal approach to build and deploy a serverless application with more than one function. One can either use different messaging systems (e.g., AWS SNS), or rely on higher level abstractions delivered by services such as AWS Step Functions [5] and Google Workflows [10]. For us, it is critical to compose the application in a way that facilitates dynamically offloading arbitrary functions to user-specified hosts. Additionally, complexities of offloading functions to varying end-points, supporting dynamic DAG structures, or rate limiting should be kept hidden from the developers as much as possible. In this section, we explain how UnFaaSener achieves these goals. We motivate and explain design decisions that let us inject offloading decisions, propagate future decisions, and merge branches for DAGs. Throughout the section, we will use a simple example (shown in Figure 2) to demonstrate each implementation aspect.
Pub/Sub as the glue. Using a topic-based publisher/subscriber (pub/sub) messaging pattern enables us to control where each function has to be executed without changing

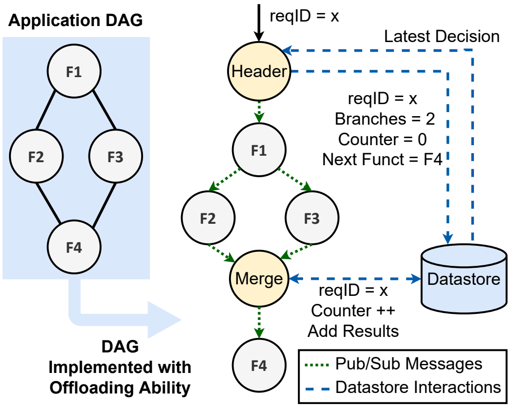


Figure 2: DAG conversion to support dynamic offloading.

the serverless platform. To do so, UnFaaSener first assigns a unique pub/sub topic to each offloading host (e.g., user's VMs) as a unique subscriber. Pub/sub's flexibility enables adding, removing, or hot-swapping hosts. This means an offloading host can be substituted by subscribing to the same topic without modifying pub/sub topics. Second, a few lines of code are inserted at the end of any non-terminal function to enable publishing to serverless or to any specific offloading host. We call this the *routing epilogue*. Finally, the routing decision for each function needs to be delivered to its parent function's routing epilogue. To do this, UnFaaSener adds a lightweight header function as the entry point of the DAG. Its role is to communicate the routing decisions made by the asynchronous scheduler to all functions by piggybacking the decisions to the incoming invocation.

The availability of pub/sub services across major clouds (e.g., AWS SNS and Google Pub/Sub) and strong support for it in various programming languages was a major driving factor for us to facilitate portability of UnFaaSener. Furthermore, the use of pub/sub provides a degree of fault tolerance by requiring subscriber acknowledgment. If no acknowledgment is received within a set timeframe, pub/sub automatically tries to resend the message. We evaluate pub/sub latency in §9.8.

Routing epilogue. The routing epilogue is a general code snippet added to the end of all non-terminal functions in the application. The code snippet in Figure 3 shows the routing epilogue for a Python function. The routing epilogue adds only a conditional statement in the critical path of execution, resulting in negligible added latency compared to the regular call to the subsequent function. Based on the routing character sent to this function by the header function, the role of this code piece is to route the invocation to a serverless endpoint or to any offloading host. The former is encoded by a "0" character, and the latter is determined by the host ID embedded in the character. This works as UnFaaSener names pub/sub topics for offloading hosts to follow the same convention: e.g., `hostTopic1`, `hostTopic2`, etc.

Header function. As illustrated in Figure 2, UnFaaSener adds a header function to the head of the DAG. The role of the header is to generate the routing decisions to be used by the routing epilogue of non-terminal functions. It also takes care of creating entities for the merging points, as described next.

```
...
if (routing == "0"):     # run next function in serverless
    topic = publisher.topic_path(projectID, "F3")
    publish_future = publisher.publish(topic, data=message, reqID=reqID, routing=routingData.encode('utf-8'))
    publish_future.result()
else:   # offload next function to a host
    hostNumber = ord(routing) - 64
    hostTopic = "hostTopic" + str(hostNumber)
    topic = publisher.topic_path(projectID, hostTopic)
    publish_future = publisher.publish(topic, data=message, reqID=reqID, invokedFunction="F3",routing=routingData.encode('utf-8'))
    publish_future.result()
```

Figure 3: The routing epilogue appended to a function.

Merge function. In a generic serverless DAG, a function may have several predecessors. A challenge is dealing with predecessors finishing at different times. Since serverless functions are primarily stateless, we need to persist information sent by predecessors. One way to solve this is using stateful functions, such as AWS Step Functions [5] and Azure Durable Functions [22], to join on predecessors. To provide a merging mechanism that works for predecessors executed on different hosts UnFaaSener uses a database; specifically, a NoSQL database (Google Datastore) in our current implementation.

As shown in Figure 2, the header function creates a new entity (data object) for each merging point in the Datastore by passing the metadata containing the request ID, the number of branches, and the subsequent function to be invoked after that merging point. The merge function, which is added between the predecessors and the child function, is invoked by each predecessor. It keeps track of the content and number of responses from the predecessors using the Datastore entity. It determines the completion of the last predecessor by comparing the number of times it was invoked by the same request ID to the predecessor count stored for it by the header function in Datastore. The merge function then triggers the next function and removes that entity from the Datastore.

DAG description. The developer provides UnFaaSener with a DAG description JSON file, which defines the structure of the DAG. The code snippet in §A.2 shows the workflow description for one of our benchmark applications.

5 Asynchronous Scheduler

In this section, we describe UnFaaSener's scheduler and how we address the challenges stated earlier in §3.

An asynchronous scheduler off the critical path. To address challenges 2(a), 2(b), and 2(c), we design UnFaaSener's scheduler to only activate when making offloading decisions.

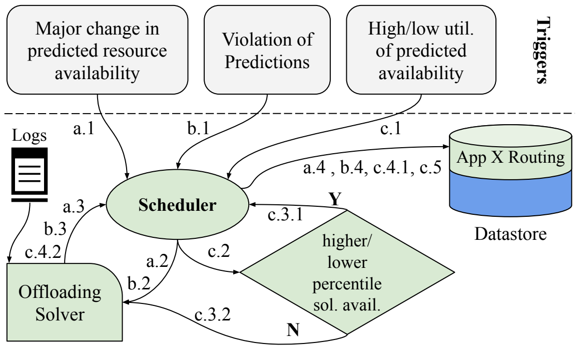


Figure 4: Different events triggering the asynchronous scheduler alongside the decision-making flow for each.

New offloading decisions do not need to be determined for every single invocation, and the header function (§4) can rely on the most recent decisions available in the Datastore.

Run in an offloading host. To run UnFaaSener, one or more offloading hosts are needed. Otherwise, there is no offloading and subsequently no cost reduction. Now, if there are already offloading host(s) available, we can leverage them to host the asynchronous scheduler as well. This addresses challenge 2(d) stated in §3. When there are more than one offloading hosts, the developer can tag one of them as the leader at the setup stage. If the leader host is not specified by the developer or if the leader fails, we use the Raft [64] consensus algorithm to elect the next leader in the order of the highest average available resources since deployment.

Triggering rules. The offloading solver, which will be presented in the next section, determines the most optimal offloading decisions theoretically. However, serverless workloads can be highly dynamic. The request inter-arrival times are shown to be highly variable [72], function execution times for certain applications can be heavily input-dependent [62], and the relatively short median execution time for serverless functions [32, 72] amplifies the relative performance jitter in the presence of third-party API calls. The resource utilization of offloading hosts can change frequently, as well. To prevent the solver from being constantly re-invoked, we pick minimal triggering rules and rely on pre-solved scenarios. Figure 4 shows how system events trigger the scheduler.

A common case requiring new offloading decisions is variations in the invocation rate. UnFaaSener's Log Collector (§8) maintains an updated view of each application's invocation rate distribution on the leader host. Whenever the offloading solver is run, it solves the same problem for different percentiles of the invocation rate distribution: 25^{th}, 50^{th}, 75^{th}, and 95^{th} percentiles. The scheduler uses the 50^{th} percentile solutions for determining offloading decisions. Later, if the utilization of predicted available resources was too high (85%) or too low (20%), the scheduler can quickly pick the solution for a higher or lower invocation percentile, respectively. We found this simple mechanism effective for rapid flow control without re-running the solver. In the high load case, if the offloading decisions assuming the 95^{th} percentile rate is still not low enough, the solver is triggered.

6 Offloading Solver

In §3, we discussed the offloading solver that sits at the core of UnFaaSener and is responsible for determining which functions to offload, to where, and to what degree. At a high level, the solver is a non-linear optimizer that considers execution times, latencies, costs, resource demands, and availabilities to determine the optimal offloading decisions. We describe various design choices for UnFaaSener's solver in this section.

Supporting different optimization goals: Any optimization has an objective function. As one can imagine, the wide range of service-level objectives (SLOs) in public cloud offerings and the broad spectrum of serverless applications (e.g., from latency-sensitive speech recognition [46] to cost-oriented nightly builds [58, 60]) prevents using a one-size-fits-all objective function. As a result, we built UnFaaSener's solver to support two different modes of operation: the **cost mode** and the **latency mode**. In the cost mode, the goal is to get maximum cost reduction without considering any constraints on the added latency. In the latency mode, the solver aims to optimize for maximum cost reduction in the face of a specific tolerance window for the added latency. By default, the tolerance window is set to the median latency of the workflow (the start of the first function to the end of the last function) when executed fully on serverless. This value is easily modifiable by the user, and we evaluate the impact of changing it in §9.3. Users have the flexibility to modify existing optimization modes or introduce their own.

Considering locality: The solver can be re-invoked based on changes in the resource availability of the hosts, change in traffic patterns, etc. Each invocation of the solver should not result in vastly different offloading decisions. Instead, the solver should consider the current offloading host(s) of each function and minimize migrations as much as possible. This is because offloading a function to a new host would incur additional latency to pull the code and build a container image. The solver has a locality parameter, α, to control the degree of emphasis placed on the locality.

Robustness to performance variations: The solver consumes the data gathered by the Log Collector (§8) to determine the most optimal offloading. These logs contain observations for execution times on the serverless platform and various hosts, as well as communication latencies. Execution time for serverless functions has been shown to be highly variable [42, 66]. The latency of pub/sub, which we use to build our flexible offloading framework, can be highly variable too (we evaluate this in §9.8). In this context, using only the mean or median of limited observations can potentially misguide the solver, specially in the latency mode, where a latency QoS should be respected. Thus, the solver first checks the similarity between serverless and host execution time distributions using the Kolmogorov–Smirnov (KS) test, a non-parametric method that makes no assumptions about the normality of the data distributions. If the KS test reveals distinct distributions for more than 70% of offloaded workflow functions, the solver uses mean statistics. However, if the distributions are not distinguishable for more than 30% of offloaded functions, the solver solves for the following three scenarios using non-parametric confidence intervals (95% confidence level [61]) and reports the average of three solutions as the final answer:

1. **Best case:** This is to model the situation where everything goes well for offloading to hosts. The lower confidence interval bound for each host's execution records, and upper confidence interval bound for serverless execution records are used by the solver.
2. **Worst case:** Here, confidence intervals are chosen so that we account for the low-end cost reduction and high-end added latency for offloading each function. Thus, the choices for confidence interval bounds are opposite to the best case mode.
3. **Mean case:** Here, the mean of system logs for each unique function-host pair is used.

The total offloading decision percentage for each function is limited to 90%, whether using a single or triple solver. Keeping some traffic on the serverless platform allows for 1) continuously observing execution times and latencies in spite of a varying workload, and 2) keeping some serverless function images warm in case the host(s) becomes unavailable.

Multi-host offloading: When multiple hosts are available, maximum cost reduction may require offloading the same function to more than one host. In such a case, each host will be in charge of a portion of the incoming traffic to that function. Here, the solver considers the resource availability of multiple hosts and recommends partial (as opposed to binary) offloading decisions.

Optimization formulation: We have reviewed various design aspects of the solver. Let us go over the formulation of the optimization problem, shown in Algorithm 1. The latency mode optimization has an additional constraint (Constraint 4) for comparing the added latency with the latency tolerance.

Intuitively, the solver aims to find offloading decisions that lead to maximum cost reduction, without violating resource or QoS constraints. Offloading decisions are partial (as opposed to binary) to ensure delivering cost savings even with limited host capacity. Cost is defined as the serverless execution cost, which the solver can try to minimize; unlike the host cost, which is already paid for. The predicted serverless cost is based on the invocation rate and execution time history, as well as the capacity required by the function. Therefore, the solver will prioritize offloading the function with higher resource usage, higher execution time, or higher invocation rate, as it will result in a higher cost reduction.

Implementation: We implemented the solver in Python to leverage its rich optimization and data manipulation packages. We used the GEKKO [19] package for mixed integer nonlinear programming (MINLP), used the CriticalPath [76] package for slack analysis, and used the Pandas [1] package for storing logs and efficiently performing complex statistical operations.

Algorithm 1 Optimization algorithm used by the Solver.

```
$\alpha$: Locality Weight
$\mu$: Adjusted Average CPU Utilization
$d^t_{n,i}$: $\text{Func}_i$ offloading percent on $\text{host}_n$ at $\text{time}_t$
$rps_i$: Request per second for $\text{Func}_i$
$\text{ExecTime}_{n,i}$: Execution time of $\text{Func}_i$ on $\text{host}_n$
Scenarios: All offloading scenarios based on partial decisions.
$\text{host}_n$ or serverless $\leftarrow$ scenario$[i]$ ▷ The assigned placement for Func$_i$ in a scenario.
Directed Paths: All paths starting from an initial node and ending at a terminal node in a DAG
$\text{CommLatency}_{\text{scenario}[m],\text{scenario}[n]}$: Communication latency between $\text{Func}_m$ on scenario$[m]$ and $\text{Func}_n$ on scenario$[n]$.
$\text{Slack}_{\text{path}} = \text{Duration}_{\text{critical path}} - \text{Duration}_{\text{path}}$
$\text{ExecLatency}_{n,i} = \text{ExecTime}_{n,i} - \text{ServerlessExec}_i$    ▷ Added latency by executing $\text{Func}_i$ on $\text{host}_n$ versus serverless execution.
procedure CALCCOST($d^t_{n,i}$)
  Cost = 0    ▷ Assumes hosts with fixed cost regardless of utilization, e.g., a VM billed at an hourly rate.
  for $\text{Func}_i \in$ functions do
    $\text{offloading}_i = 0$
    for $n \in$ offloadingHosts do
      $\text{offloading}_i \leftarrow \text{offloading}_i + \frac{d^t_{n,i}}{100}$
      Cost $\leftarrow$ Cost+ $\alpha \times \left|\min(d^t_{n,i}, 1) - \min({d_{n,i}}^{t-1}, 1)\right|$
    Cost $\leftarrow$ Cost+$(1-\alpha) \times \text{cost}_{\text{Func}_i} \times \text{rps}_i \times$ (0.9- $\text{offloading}_i$)
Constraint 1: $\sum_{\text{Func}_i \in functions} \mu \times \text{ExecTime}_{n,i} \times \text{rps}_i \times \frac{d^t_{n,i}}{100} \leq \text{AvailableCPU}_{\text{host}_n}$
Constraint 2: $\sum_{\text{Func}_i \in functions} \text{Mem}_i \times \text{ExecTime}_{n,i} \times \text{rps}_i \times \frac{d^t_{n,i}}{100} \leq \text{AvailableMem}_{\text{host}_n}$
Constraint 3: $\sum_{n \in \text{offloading hosts}} d^t_{n,i} \leq 90$
Constraint 4: (only for the latency mode)
for scenario $\in$ Scenarios do
  for path $\in$ Directed Paths do
    latency = 0
    for $\text{Func}_i \in$ path do
      if scenario$[i]$ != serverless then
        $\text{host}_n \leftarrow$ scenario$[i]$
        latency $\leftarrow$ latency $+ \min(d^t_{n,i}, 1) \times \text{ExecLatency}_{n,i}$
      for $\text{Func}_j \in$path and $\text{Func}_j \in$predecessors$_i$ do
        latency$\leftarrow$ latency $+ \text{CommLatency}_{\text{scenario}[i],\text{scenario}[j]}$
    latency$\leq \text{Slack}_{path}$+LatencyTolerance
Decision: $d^{*t}_{n,i}$ (OptimalValue) $\leftarrow \operatorname{argmin}_{d^t_{n,i}}$ CALCCOST($d^t_{n,i}$)
```

7 Host Agents

Let us explain UnFaaSener's different host agents.

7.1 Resource Monitor Agent

Implemented in C++, this lightweight agent tracks the CPU and memory usage at the host. It distinguishes between UnFaaSener-related processes (including containers running offloaded functions) and host processes to create differentiated scheduler triggers, described in §5. The monitoring period is 100 ms in our implementation

7.2 Resource Predictor Agent

The predictor agent, also written in C++, is tightly coupled with the monitor agent. It periodically (every 1 s in our implementation) predicts the maximum resource utilization in the next time window. UnFaaSener's effectiveness in cost reduction is as good as the accuracy of the predictions made by this agent. If predictions are too conservative, the host's available capacity will not be well utilized, limiting cost reductions. Conversely, aggressive predictions risk causing resource contention between offloaded functions and host processes. We formulate this trade-off using the following metrics:

1. Reclamation Efficiency (RE): *RE* represents how much of the available resources could be reclaimed based on the predicted usage. If predictions always match the peak resource usage, *RE* would be 100%.
2. Violation Rate (V): If the predicted usage in a window is lower than the materialized peak usage, the *RE* is capped at 100% and a prediction violation event is logged. *V* denotes the percentage of predictions leading to a violation.

An ideal predictor would yield $RE = 100\%$ and $V = 0\%$. For each resource dimension (e.g., CPU and memory), the objective function combining the two looks like this:

$$PredictionScore = w \times RE + (1-w) \times (100-V) \tag{1}$$

Here, w denotes the resource reclamation weight. By default, UnFaaSener gives equal importance to reclamation efficiency maximization and violation minimization ($w = 0.5$) as it does not make any assumptions about the host workloads. Further knowledge about the workload or user's tolerance of slowdown for it can change this. We do not explore this angle in this work and only use $w = 0.5$. We evaluate the prediction quality and performance of various prediction policies in §9.9 and derive the one best suited to UnFaaSener.

7.3 Execution Agent

The execution agent subscribes to the pub/sub topic of its host and is notified on incoming invocations. If that function's code is not present on the host, the execution agent proceeds to download the function's code and metadata (runtime and memory limit) with a call to the Google Cloud Functions' API. Once the function's code and metadata are retrieved, a new docker container image is built using the skeleton container for the specified runtime (e.g., Python 3.10). We use Docker Hub [8] to host skeleton images. The agent leverages Docker's build utility to generate a local image of the runtime that contains the function's code and dependencies. As the image generation step is computationally expensive, the generated images are kept on the host for future use. Prior work has shown that even employing simple keep-alive policies can notably reduce cold starts [40, 69, 72]. The execution agent stops an idle container when the time since the end of the last execution exceeds the keep-alive window (10 minutes), or when capacity is needed to execute a different function.

UnFaaSener's execution agent supports concurrent execution of multiple instances of the same or different functions. However, the agent queues incoming requests on the host if the predicted resource availability is more than 90% utilized, or if the current degree of concurrency is at or beyond the concurrency limit. The execution agent has a feedback mechanism to set the concurrency limit dynamically based on observed CPU utilization and performance degradation of offloaded functions. It starts with assuming one CPU thread per function, the concurrency limit is thus set to the number of cores available in the host. Over time, the agent has access to average container CPU utilization reported by the monitoring agent. It also has access to execution time trends for functions allowing it to measure any slowdown. Combining the two, it calculates the adjusted average CPU utilization as:

$$\mu = \min(1, \text{AvgFuncCPUUtil} + 0.03 \times (e^{\Delta \text{ExecTime}} - 1)) \tag{2}$$

From this, the concurrency limit is calculated as $\frac{\text{Core Count}}{\mu}$. The insight for this asymmetric feedback mechanism is as follows: if no performance degradation is sensed, μ directly reflects the average utilization of docker containers hosting offloaded functions. However, with slight degradation, μ is increased super linearly to reduce the concurrency limit. More details on this process is provided in §A.1.

UnFaaSener's approach of using pre-solved offloading decisions accelerates the scheduling path. This approach also makes the scheduler early binding [50]. The distributed nature of the system introduces some delay from when a burst hits to the time that the host scheduler updates the offloading decisions. This can create a request build-up on the hosts' queues. To prevent QoS violations, the execution agent monitors requests in the queue and if their wait time exceeds a specified window (2 s by default), that request and all its descendant function calls are redirected to the serverless platform.

In §4 we described how using the subscription retry policy of pub/sub can provide a degree of fault tolerance. If no acknowledgment is received within the specified number of retries (default 5), the message is redirected to the subscriber's *dead-letter* topic, provided by pub/sub. We have designated the topic of $host_{(i+1)\%num_{hosts}}$ as the *dead-letter* topic for $host_i$. Additionally, to prevent duplicate execution on the host side, we have made use of the *exactly-once delivery* feature. However, our current implementation lacks complete handling of a host failure scenario. Although no further offloading occurs to the failed host, any invocations previously acknowledged and queued on that host will be lost.

Offloaded function execution logs are stored locally and periodically published to a Datastore entity every one thousand logs for backup. This enables a smooth transition to a new leader host in case of primary host failure.

8 Log Collection

The log collector runs periodically (every 1 minute in our implementation) on the leader host, collecting execution and latency logs for serverless as well as host executions logs. The logs for serverless executions are collected using the Google Cloud CLI (gcloud [27]). Aside from execution time information, serverless logs contain timestamps for the invocation of the header function that runs exclusively on serverless. These timestamps are used to calculate the invocation rate distribution for the workflow, which is used by the solver.

Host execution logs are needed by the solver to enforce the latency tolerance QoS (discussed in §6). As mentioned in §7.3, the execution agent writes execution records (start time, end time, invocation ID, and function ID) to local log caches, and asynchronously backs them up on the execution log Datastore entity. In the event of leader failure, the log collector retrieves execution records from the Datastore entity and stores them in a pandas dataframe, similar to how local log caches store host logs. This stored data is readily accessible for the offloading solver upon activation. To accelerate statistical operations, the dataframe stores a maximum of N most recent data points per function-host, with $N = 50$ yielding favorable results in our experiments.

9 Evaluation

9.1 Setup, Methodology, and Benchmarks

UnFaaSener has many components and serverless settings are complex. We try to carefully distill various design and performance aspects of UnFaaSener without confusing ourselves or the reader with unnecessary complexity. Each experiment conveys specific points to help readers build a mental model. The majority of our evaluations are from real deployment, but we also conduct some simulations to stress certain parts of the system with more usage scenarios. Any reported cost normalization is based on the scenario where all the functions of the workflow run on serverless.

9.1.1 Benchmark Applications

We used five real-world serverless applications to evaluate UnFaaSener; 1) DNA Visualization [28], a script that performs visualization of the input DNA sequence file, 2) Image Processing [51], an application that performs a sequence of operations on input images, 3) Text2Speech Censoring [34] turns short text segments into speech and censors any profanities within the text segment, 4) Regression Tuning [12], which solves a regression problem using Keras [25], and 5) Video Analytics [13], an application that performs object recognition on images generated from a video stream. Table 1 shows the diverse range of structures covered by these applications. The dynamic fanout column in the table captures whether the application has a sub-graph with parametrized fanout.

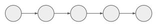
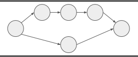
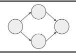
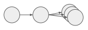
Benchmark	Branch	Dyn. Fanout	DAG Structure
DNA Visualization	✗	✗	
Image Processing	✗	✗	
Text2Speech	✓	✗	
Regression Tuning	✓	✗	
Video Analytics	✗	✓	

Table 1: Benchmark applications were chosen to represent various structures present in today's serverless.

9.1.2 Workload Invocation and Traffic

We use FaaSProfiler [9, 71], a serverless testing tool used by prior work [50, 66, 72], to invoke traffic patterns precisely. Depending on the nature of the experiment, we use different invocation patterns. For evaluating high-level trade-offs or assessing the extremes, using a uniform invocation rate suffices. For those with co-location scenarios, we use the 2021 Azure Functions Invocation Trace [6, 86].

9.1.3 Ensuring fair comparisons

We take the following steps to make sure that our reported gains are not inflated: 1) All cost numbers include the cost of functions added by our system; 2) all latency numbers are end-to-end and include the latency overhead introduced by UnFaaSener; 3) each function is tuned [2] for the most cost-optimal memory configuration. By making the baseline serverless functions as cost-efficient as it gets, we ensure that UnFaaSener's cost savings are not an artifact of comparing to bloated functions; 4) when comparing UnFaaSener to alternative solutions (§9.5), we ensure that each implementation is minimal and tuned to the specific offering; 5) we invoke each application with a set of random inputs; 6) we ignore the bootstrapping phase measurements, as UnFaaSener tends to offload more during this phase, leading to more cost savings.

9.2 Latency Mode vs. Cost Mode

Here, we use a VM with 4 vCPUs and 16 GB of memory to demonstrate how even a relatively small host can be used by UnFaaSener to offer cost savings. Figure 5 shows the normalized execution cost and end-to-end latency values associated with running three applications in latency and cost modes. For each application, we chose the invocation frequencies such that the maximum offloading (with a low rate) and minimum offloading (with a high enough rate) are stressed.

As expected, using the cost mode leads to more savings than the latency mode. It comes at the expense of potentially increased latency since the solver considers no latency constraint when making offloading decisions in the cost mode.

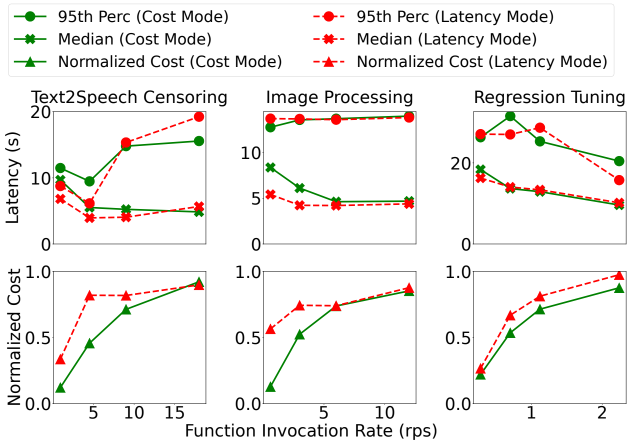


Figure 5: UnFaaSener's cost savings and end-to-end latency in two optimization modes.

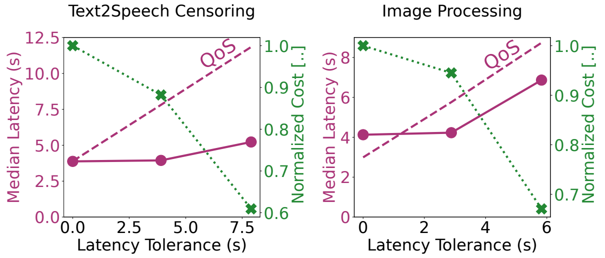


Figure 6: Increasing the latency tolerance in the latency mode leads to more offloading, and thus more savings.

Even in the latency mode, UnFaaSener can reduce the cost, but is limited by the default latency tolerance. As the invocation rates increase, the cost savings achieved by UnFaaSener decrease. This is because UnFaaSener is designed to harvest the unused computing resources of already-allocated hosts. Therefore, when the invocation rates are high, the system is fully utilizing the resources, leaving little or no unused capacity to be harvested.

Later in §9.3, we show how changing the latency tolerance affects the results in the latency mode.

We measured the added latency of the header function to be ~50ms. The end-to-end latency values include this overhead.

9.3 Latency Mode and Tolerance Window

In the latency mode, the solver tries respecting a tolerance on the added latency when making offloading decisions. Figure 6 shows the effect of changing the tolerance window on the cost (green cross markers) and median latency (purple circle markers). We studied its effect on two of our benchmarks: Image Processing and Text2Speech Censoring. The tolerance windows are set to zero, the median, and twice the median latency of each workflow when solely run on the serverless platform (baseline latency). The QoS (baseline + tolerance) for these applications is depicted with purple dashed lines.

Overall, UnFaaSener manages to offload in a controlled fashion and complies with the set tolerance window; except for when latency tolerance of 0 is set for Image Processing. Image Processing is a chain and thus every offloading is on the critical path (see Table 1). This is unlike Text2Speech that has asymmetric branches, because of which it enjoys higher cost reductions and a higher safety margin from the QoS line.

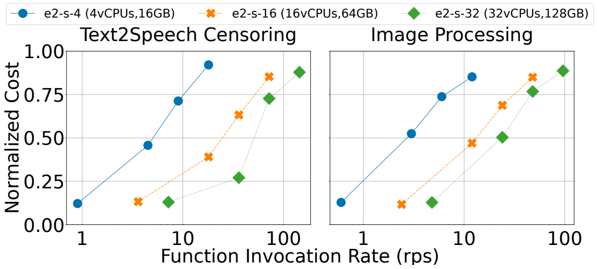


Figure 7: UnFaaSener leverages larger hosts effectively to increase offloading (reduce cost).

9.4 Impact of Host Size

The results presented so far were gathered on a relatively small host. This was to show that even with a small host, or equivalently, with a small leftover capacity, UnFaaSener can offer cost savings. It is worth asking how those results scale if we use a larger host. To answer this, we scale up the cost-mode experiments presented in §9.2 on two larger hosts. To prevent factors other than the resource capacity, we picked larger hosts from the same VM family as the small host (Google Cloud's *e2-standard* family). For brevity, we only show the cost saving results for two benchmark applications in Figure 7. The X-axis is logarithmic, and multiplying invocation rates appears as a shift to the right. As seen, with increased host size, the cost saving curve is shifted to the right consistently.

9.5 Comparison to Alternative Solutions

The primary objective of UnFaaSener is reducing the cost by offloading to pre-paid hosts. To evaluate whether we accomplished this goal, we compare the maximum cost savings, which is also the worst case latency, offered in the cost mode with two popular serverless workflow platforms: AWS Step Functions and Google Workflows. We also compare it to using AWS Lambda functions glued with AWS SNS, and Google Cloud functions glued with Google Pub/Sub. The latter is the underlying setup for UnFaaSener. Furthermore, we present results for UnFaaSener's latency mode using the default tolerance window. Figure 8 compares the average latency and cost per invocation for each benchmark and platform. We could not port Text2Speech Censoring benchmark to AWS as this benchmark utilizes Google Translate's text-to-speech API [11]. The cost mode is all about cost reduction. We see that UnFaaSener is able to trade latency with cost effectively, lowering the cost by about an order of magnitude. Using the latency mode limits this cost reduction depending on the latency tolerance expressed by developers (§9.2 and §9.3), and helps cut down on average latency compared to the cost mode, but the cost savings are not as significant. It is worth noting that DNA Visualization, a single-function application, experiences no added latency as it has low coordination overhead.

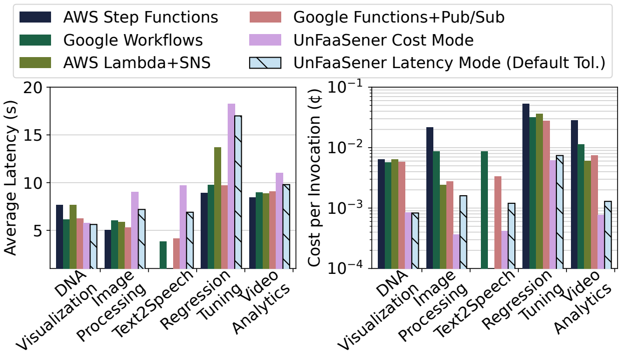


Figure 8: Comparison of latency and cost.

Finally, we see that using workflow coordination services comes with major cost implications. This is because, in addition to the capacity and invocation cost elements mentioned in §2.2, the developer has to pay for state transitions in these offerings [4, 15].

9.6 Adaptive Cost Saving

We are interested in assessing the responsiveness of UnFaaSener to host processes. We use the host with 16 vCPUs from earlier in this experiment. We replay a sample trace from the 2021 Azure Functions traces [6] using FaaSProfiler [9] to invoke the Image Processing application. At second 110, the Graph Analytics workload from CloudSuite 3.0 [7, 65] is run on the host for about 80 seconds. The workload uses Apache Spark to perform graph analytics on a large-scale Twitter dataset and uses all 16 vCPUs as well as 15 GB of memory. Before and after this window, the host is mostly idle and fully available for offloading.

Figure 9 shows the cost for each execution of the Image Processing workflow over time. The timespan for the execution of the heavy host workload is marked with dashed lines. Soon after 110 s, the execution agent slows down admitting new requests and the monitoring agent triggers the scheduler with a prediction failure trigger due to a sudden host load spike. New predictions are made, and the solver makes new offloading decisions for minimal offloading. This is reflected in increased cost during that period. After the VM workload ends, the predictor remains cautious briefly before declaring the majority of the host's resources as available. Offloading to the VM resumes as before. The two orders of magnitude difference in cost is attributed to the non-offloadable header function taking less than 100 ms to execute, while the rest of the application takes 3-4 seconds.

9.7 Host Interference

To measure the impact of UnFaaSener on host processes, we run Text2Speech and Image Processing applications on the 16-core host described in §9.4. We invoke them with the similar rates to those used in that section, and adhere to using the cost mode which guarantees maximal offloading, and equivalently, maximum host-side interference. We use three standard benchmark applications on the host to measure the impact on CPU, memory bandwidth, and network performance. For the first two, we use Sysbench [53], in CPU and memory test modes, respectively, each with 15 active threads. For network, we use the Iperf [79] benchmark. Figure 10 shows the performance of these benchmarks when normalized to that of an idle host without UnFaaSener running. Across experiments, the maximum average degradation at maximum offloading is less than 15%. We find this degradation reasonable considering that 1) benchmarks used are highly sensitive and 2) the developer is in the loop and aware of the operation of UnFaaSener resource harvesting. We observe that sysbench-mem benchmark experiences significantly higher degradation for the Text2Speech serverless application. We pinpointed this to Text2Speech having a function with a 2 GB memory configuration, whereas the largest function for Image Processing requires 256 MB of memory.

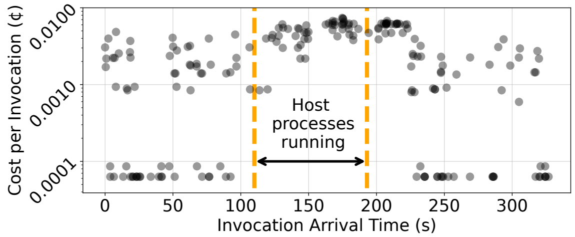


Figure 9: UnFaaSener adaptively adjusts offloading to respect host processes.

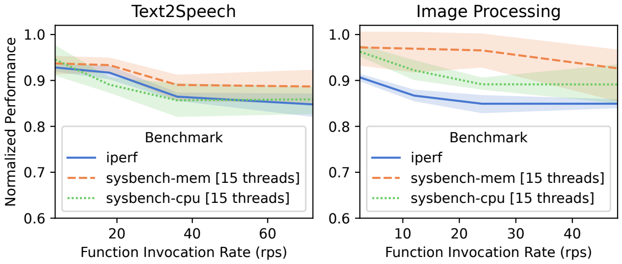


Figure 10: Degradation of host processes is a function of traffic rate and the resource requirements of the functions.

9.8 Pub/Sub Latency Overhead

Using pub/sub allows us to invoke functions on and from offloading hosts. Here, we characterize the latency overhead of the Google Pub/Sub [14], used by UnFaaSener.

In our experiment, the publisher is a Google Cloud function in the East Coast sending payloads to three subscribers: 1) another Google Cloud function in the same region, 2) a Google Cloud VM in the same region, and 3) a private VM in the West Coast. We test each publisher-subscriber pair with two message sizes (10 KB and 1 MB) and two invocation rates (0.1 rps and 5 rps). These values were driven by a recent characterization of production serverless DAGs [59]. We chose 10 KB and 1 MB message sizes to estimate high-end values for regular and high-fanout DAGs [59], respectively. Similarly, 5 rps and 0.1 rps rates were chosen as high and medium average invocation rates based on that characterization. We collected 250 samples per scenario (3,000 samples in total).

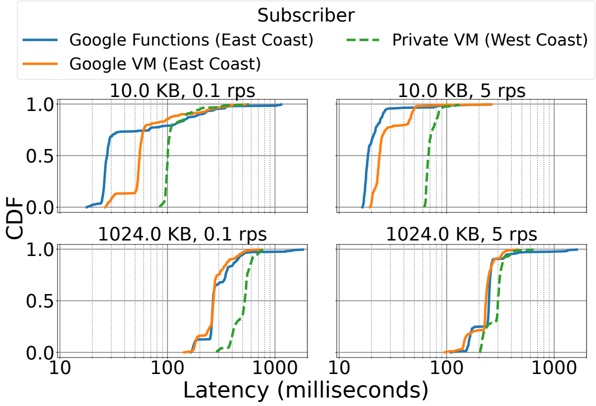


Figure 11: Pub/sub latency depends on message size, subscriber type, distance, and messaging rate.

Figure 11 compares the distribution of latency depending on the subscriber. Higher message size (1 MB vs. 10 KB), longer distance (West vs. East Coast), and very low invocation rate (0.1 rps vs 5 rps) increase the latency. The latter is likely caused by higher chances of lost connections between the publisher, forwarder, and subscriber during longer invocation periods. The wide distributions in Figure 11 reveal Pub/Sub's high performance variability. Even for two Google Cloud functions in the same region, the latency varies from less than 20 ms to more than a second. Such non-deterministic variations in Pub/Sub latency alongside high performance variations of serverless functions motivated us to use confidence intervals statistics for the solver.

9.9 VM Resource Prediction Policy

The predictor agent runs on each host to forecast the maximum resource utilization anticipated for host processes. It has two predictors: one for CPU and another for memory.

We dedicate this section to compare various prediction policies and determine their fitness to be used by the Predictor agent. To assess prediction policies for a wide range of VM usage scenarios we rely on simulations. We use the Azure Public Dataset [29, 30] to simulate different prediction policies for 1 million VMs over a 30-day period. The dataset includes 5-minute VM CPU utilization readings (min, average, and max utilization per reading) and has no memory readings. CPU utilization is inherently more variable than memory utilization. Besides, CPU utilization percentages can experience a wider variation range due to CPU being the typical resource bottleneck in public clouds [45]. These factors make forecasting CPU utilization a harder task.

We implement eight common time series forecasting methods and use traces from 100,000 VMs to train the best policy parameters for them. These trained policy parameters are used as the initial parameters for test VMs and during the lifetime of each test VM the parameters are periodically retuned per VM. A brief description of explored methods and their corresponding parameters (shown in curly brackets) is listed below. For all policies, we also consider a safeguard margin (m) to allow exploring conservative predictions. To derive this margin despite inherent differences between various prediction methods, we also included m as a training parameter.

Simple Exponential Smoothing (SES) {pars: m, α}:

$$s_{n+1} = \alpha x_n + (1-\alpha)s_n,\ x_{n+1} = s_{n+1} \times (1+m) \qquad (3)$$

Simple Moving Average (SMA) {pars: m, N}:

$$x_{n+1} = \frac{1+m}{N} \sum_{n-N+1}^{n} x_i \qquad (4)$$

The averaging window is limited to the existing number of observation if there are less than N observations.

Histogram (Hist) {pars: m, p, d}: Recent work [68, 72, 81] has demonstrated the superior performance of histogram-based time series forecasting. We implemented a histogram with a 1% utilization resolution. At each prediction window, the max observed utilization observed gets rounded to determine the histogram bin to be incremented. A certain percentile of the histogram (p) is then used to determine the prediction for the next window. A high percentile reduces violations, but reduces reclamation efficiency. Old observations in the histogram are depreciated using the decay factor (d).

Markov Chain (MC) {pars: m, r, o}: Markov Chains have been used for time series forecasting in various problem domains [18, 23, 49]. We build a simple MC predictor for CPU prediction. A state transition matrix (STM) captures the history of state transitions. Each state is a range of CPU utilization percentages; with state resolution (r) of 5%, there are a total of $\frac{100}{5} = 20$ states. The current state is determined based on the latest utilization observation: $i = \lceil 100\%/x_n \rceil$, and the forecasted utilization is:

$$x_{n+1} = \frac{\sum_{j=1}^{\lceil 100\%/r \rceil} STM_{i,j} \times (j+o)}{\sum_{j=1}^{\lceil 100\%/r \rceil} STM_{i,j}} \times (1+m) \qquad (5)$$

Here, o is an offset to compensate quantization of values.

Other predictors: We also implemented Double Exponential Smoothing, Autoregressive, Passive Aggressive Regression, and ARIMA predictors. We do not present their description and results for brevity due to their mediocre performance compared to SES and SMA despite more complexity.

Figure 12 compares prediction scores for these four policies. It also shows an Oracle policy where future is known, in which case RE is always 100% and V is always 0%. We only include VMs with a minimum lifetime of 30 minutes (65% of all VMs), as 5-minute readings mean that shorter lifetimes require 4 or fewer predictions; too few to draw meaningful statistical conclusions from. We see that any active prediction policy is significantly better than relying on the latest observation of peak utilization. While we picked MC for delivering slightly better scores, we do not observe considerable differences among these policies. To better understand what limits prediction scores, we look at resource reclamation efficiency and prediction violations of the MC prediction policy separately, as a function of VM lifetime (Figure 13). Results here include any VM with lifetime of at least 15 minutes (91%

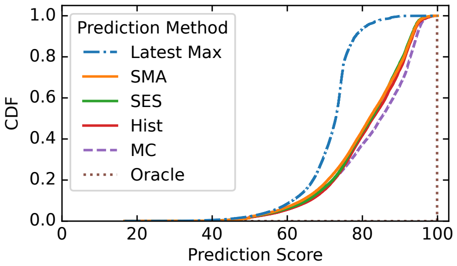


Figure 12: Prediction scores for simulated policies.

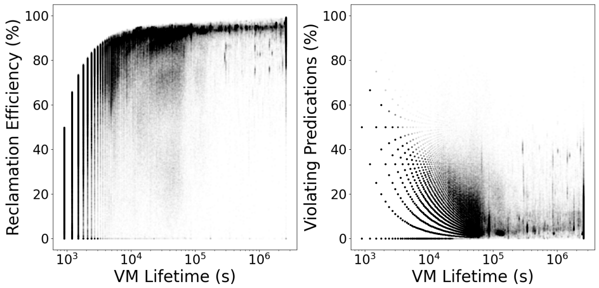


Figure 13: Resource reclamation efficiency and prediction violations as a function of VM lifetime for 1 million VMs.

of all 1 million VMs). We observe that with increased VM lifetime, the resource efficiency is increased and violations are reduced. This is not surprising, as with longer history, policy parameters can be better trained for each VM.

10 Related Work

Table 2 compares UnFaaSener with the related works targeting offloading in the serverless domain. For a category of related work, the application is primarily deployed on VMs and serverless plays the backup role at scale-out, while new VMs are being provisioned [43, 47, 63, 84, 87]. LIBRA [67] extends these works by simultaneously utilizing FaaS for the low-rate bursty portion of the traffic. Reliance on VMs as the primary infrastructure limits the scope of such systems to specific domains (as seen in Table 2), and in effect distances them from serverless's high scalability and pay-per-use features.

Another category of work proposes building hybrid IaaS-FaaS deployments [56, 75]. The main drawback of these systems is adding a second scheduling/control layer on top of that of serverless platforms. Moving away from the scheduler of public serverless offerings as the primary scheduler and adding a new layer limits the scalability of these approaches, and comes with reliability and security implications. UnFaaSener relies on the serverless scheduler to ensure a secure and scalable gateway to external events, and by using pre-solved decisions, eliminates added scheduling overheads.

Lastly, a category of work proposes modifying the serverless platform to enable serverless functions to use resource-harvesting VMs [86], idle resources from over-allocated serverless functions [83], and users' VMs on the same platform [78]. These proposals modify cloud providers extensively, which is out of reach of end users, and limit offloading to the scheduler's scope, usually within the same cluster. UnFaaSener works on top of existing serverless systems, requires no change to the platform, and puts no limit on the location of hosts it is harvesting resources from.

Related Work	**Supports Complex DAGs**	**Scheduling/ Control Path**	**General Purpose**	**Resource Harvesting**	**Primary Deployment**	**Partial Offloading Decisions**
Splice [75]	✗	FaaS scheduler + Custom Scheduler	✓	✗	Hybrid	✗
Spock [43]	✗	FaaS scheduler + Custom Scheduler	✗ (ML Inference)	✗	IaaS	NA
SplitServe [47]	✗	FaaS scheduler + Custom Scheduler	✗ (Spark Jobs)	✗	IaaS	NA
MArk [84]	✗	FaaS scheduler + Greedy Instance Plan	✗ (ML Inference)	✗	IaaS	NA
Amoeba [56]	✗	FaaS scheduler + Custom Controller	✗ (Microservices)	✗	Hybrid	✗
FEAT [63]	✗	FaaS scheduler + Custom Controller	✓	✗	IaaS	NA
LIBRA [67]	✗	FaaS scheduler + Custom Controller	✓	✗	IaaS	✓ (offloads excess traffic)
ServerMore [78]	✗	FaaS scheduler + Custom Controller	✓	✓	FaaS	✗
Skedulix [31]	✓	FaaS scheduler + Custom Scheduler	✓	✗	Private FaaS	✗
Kraken [20]	✓	Kraken Scheduler	✓	✗	FaaS	NA
Freyr [83]	✗	Serverless Controller + Resource Manager	✓	✓	FaaS	NA
Zhang et al. [86]	✗	FaaS scheduler	✓	✓	FaaS	NA
BeeHive [87]	✗	FaaS scheduler + Custom Runtime	✗ (Web Apps)	✗	IaaS	✓ (sets offloading ratio)
UnFaaSener	✓	FaaS scheduler + lightweight LUT	✓	✓	FaaS	✓

Table 2: The taxonomy of related work.

11 Discussion

Threat model. UnFaaSener offloads users' functions to their own offloading hosts. This simplifies the threat model by eliminating co-location of different teams' applications on the same host. Developers have full control: they specify offloading hosts, install host agents, and can unsubscribe hosts at any time. UnFaaSener's execution agent runs offloaded functions in separate Docker containers for isolation, but this also brings security implications [77]. In this context, we assume that 1) the host, which belongs to the same development team, is not malicious and 2) the developer is aware of the data protection implications of offloading functions.

12 Conclusion

UnFaaSener enables serverless developers to leverage the unused capacity of their VMs or on-premise servers, yielding substantial cost savings without modifying existing serverless platforms. UnFaaSener lays a foundation for researchers to explore the potential of the proposed serverless offloading mechanism for diverse purposes beyond cost optimization.

13 Acknowledgements

We thank members of the UBC CIRRUS Lab for their feedback on this work, and thank Ila Nimgaonkar for suggesting the name UnFaaSener. We also thank the anonymous reviewers and our shepherd, Chia-Che Tsai, for helping us improve the paper. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) grants RGPIN-2021-03714 and DGECR-2021-00462. This work was made possible by cloud resources from Digital Research Alliance of Canada, Google Cloud Research Credits program, and AWS Cloud Credit for Research program.

References

[1] pandas, Accessed: 2023-01-05. https://pandas.pydata.org/.

[2] AWS Lambda Power Tuning, Accessed on 2023-01-05. https://github.com/alexcasalboni/aws-lambda-power-tuning.

[3] AWS Lambda Pricing, Accessed on 2023-01-05. https://aws.amazon.com/lambda/pricing/.

[4] AWS Step Functions Pricing, Accessed on 2023-01-05. https://aws.amazon.com/step-functions/pricing/.

[5] AWS Step Functions: Visual workflows for distributed applications, Accessed on 2023-01-05. https://aws.amazon.com/step-functions/.

[6] Azure Functions Invocation Trace 2021, Accessed on 2023-01-05. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsInvocationTrace2021.md.

[7] CloudSuite: Graph Analytics, Accessed on 2023-01-05. https://github.com/parsa-epfl/cloudsuite/blob/CSv3/docs/benchmarks/graph-analytics.md.

[8] Docker Hub: Build and ship any application anywhere, Accessed on 2023-01-05. https://hub.docker.com/.

[9] FaaSProfiler, Accessed on 2023-01-05. https://github.com/PrincetonUniversity/faas-profiler.

[10] Google Cloud Workflows, Accessed on 2023-01-05. https://cloud.google.com/workflows.

[11] Google text-to-speech API, Accessed on 2023-01-05. https://cloud.google.com/text-to-speech.

[12] A PaaS end-to-end ML setup with Metaflow, serverless and SageMaker., Accessed on 2023-01-05. https://github.com/jacopotagliabue/no-ops-machine-learning.

[13] vSwarm: A suite of representative serverless cloud-agnostic (i.e., dockerized) benchmarks, Accessed on 2023-01-05. https://github.com/vhive-serverless/vSwarm.

[14] What is Pub/Sub?, Accessed on 2023-01-05. https://cloud.google.com/pubsub/docs/overview.

[15] Workflows pricing, Accessed on 2023-01-05. https://cloud.google.com/workflows/pricing.

[16] Paarijaat Aditya, Istemi Ekin Akkus, Andre Beck, Ruichuan Chen, Volker Hilt, Ivica Rimac, Klaus Satzke, and Manuel Stein. Will serverless computing revolutionize NFV? *Proceedings of the IEEE*, 107(4):667–678, 2019.

[17] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[18] R. Allard. Use of time-series analysis in infectious disease surveillance. *Bulletin of the World Health Organization*, 76(4):327, 1998.

[19] Logan Beal, Daniel Hill, R Martin, and John Hedengren. GEKKO optimization suite. *Processes*, 6(8):106, 2018.

[20] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 153–167. ACM, 2021.

[21] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 381–397. ACM, 2023.

[22] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

[23] A. Carpinone, M. Giorgio, R. Langella, and A. Testa. Markov chain modeling for very-short-term wind power forecasting. *Electric Power Systems Research*, 122:152–158, 2015.

[24] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[25] Francois Chollet et al. Keras, Accessed: 2022-10-13. https://github.com/fchollet/keras.

[26] Claudio Cicconetti, Marco Conti, Andrea Passarella, and Dario Sabella. Toward distributed computing environments with serverless solutions in edge systems. *IEEE Communications Magazine*, 58(3):40–46, 2020.

[27] Google Cloud. gcloud CLI overview, Accessed on 2023-01-05. https://cloud.google.com/sdk/gcloud.

[28] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78. ACM, 2021.

[29] Eli Cortez. Azure public dataset v1, Accessed on 2023-01-05. https://github.com/Azure/AzurePublicDataset/blob/master/AzurePublicDatasetV1.md.

[30] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167. ACM, 2017.

[31] Anirban Das, Andrew Leaf, Carlos A. Varela, and Stacy Patterson. Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 609–618, 2020.

[32] Datadog. The state of serverless, May 2021. https://www.datadoghq.com/state-of-serverless-2021/.

[33] Datadog. The state of serverless, June 2022. https://www.datadoghq.com/state-of-serverless/.

[34] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 265–276, April 2020.

[35] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering*, 2021.

[36] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2020.

[37] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[38] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[39] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting VMs in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 583–594. ACM, 2022.

[40] Alexander Fuerst and Prateek Sharma. FaasCache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400. ACM, 2021.

[41] Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149, 2022.

[42] Samuel Ginzburg and Michael J Freedman. Serverless isn't server-less: Measuring and exploiting resource variability on cloud FaaS platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 43–48, 2020.

[43] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for SLO and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208, 2019.

[44] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of

Alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19. ACM, 2019.

[45] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861. USENIX Association, November 2020.

[46] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262, 2018.

[47] Aman Jain, Ata F. Baarzi, George Kesidis, Bhuvan Urgaonkar, Nader Alfares, and Mahmut Kandemir. Split-Serve: Efficiently splitting apache Spark jobs across faas and iaas. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 236–250. ACM, 2020.

[48] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cerin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing co-located workloads in Alibaba cloud datacenters. *IEEE Transactions on Cloud Computing*, 2020.

[49] Yuxuan Jiang, Mohammad Shahrad, David Wentzlaff, Danny HK Tsang, and Carlee Joe-Wong. Burstable instances for clouds: Performance modeling, equilibrium analysis, and revenue maximization. *IEEE/ACM Transactions on Networking*, 28(6):2489–2502, 2020.

[50] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: Principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 289–305. ACM, 2022.

[51] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[52] Haneul Ko, Sangheon Pack, and Victor C. M. Leung. Performance optimization of serverless computing for latency-guaranteed and energy-efficient task offloading in energy harvesting industrial IoT. *IEEE Internet of Things Journal*, 2021.

[53] Alexey Kopytov. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/*, 2004.

[54] Samuel Kounev, Cristina Abad, Ian T. Foster, Nikolas Herbst, Alexandru Iosup, Samer Al-Kiswany, Ahmed Ali-Eldin Hassan, Bartosz Balis, André Bauer, André B. Bondi, Kyle Chard, Ryan L. Chard, Robert Chatley, Andrew A. Chien, A. Jesse Jiryu Davis, Jesse Donkervliet, Simon Eismann, Erik Elmroth, Nicola Ferrier, Hans-Arno Jacobsen, Pooyan Jamshidi, Georgios Kousiouris, Philipp Leitner, Pedro Garcia Lopez, Martina Maggio, Maciej Malawski, Bernard Metzler, Vinod Muthusamy, Alessandro V. Papadopoulos, Panos Patros, Guillaume Pierre, Omer F. Rana, Robert P. Ricci, Joel Scheuner, Mina Sedaghat, Mohammad Shahrad, Prashant Shenoy, Josef Spillner, Davide Taibi, Douglas Thain, Animesh Trivedi, Alexandru Uta, Vincent van Beek, Erwin van Eyk, André van Hoorn, Soam Vasani, Florian Wamser, Guido Wirtz, and Vladimir Yussupov. Toward a Definition for Serverless Computing. In *Serverless Computing (Dagstuhl Seminar 21201)*, volume 11, pages 34–93. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021.

[55] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Frujeri, Nithish Mahalingam, Pulkit A. Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. Prediction-Based power oversubscription in cloud platforms. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 473–487. USENIX Association, July 2021.

[56] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. Amoeba: QoS-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408, 2020.

[57] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Comput. Surv.*, 54(10s), September 2022.

[58] Álvaro López García, Jesús Marco De Lucas, Marica Antonacci, Wolfgang Zu Castell, Mario David, Marcus Hardt, Lara Lloret Iglesias, Germán Moltó, Marcin Plociennik, Viet Tran, Andy S. Alic, Miguel Caballer, Isabel Campos Plasencia, Alessandro Costantini, Stefan Dlugolinsky, Doina Cristina Duma, Giacinto Donvito, Jorge Gomes, Ignacio Heredia Cacha, Keiichi Ito, Valentin Y. Kozlov, Giang Nguyen, Pablo Orviz Fernández, Zdeněk Šustr, and Pawel Wolniewicz. A cloud-based framework for machine learning workloads and applications. *IEEE Access*, 8:18681–18692, 2020.

[59] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh

Bagchi, and Somali Chaterji. WiseFuse: Workload characterization and DAG transformation for serverless workflows. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

[60] Stefan Majiros. Nightly and serverless builds in MS appcenter for React native - async mobile DevOps example, 2021. https://stefan-majiros.com/blog/nightly-serverless-builds-with-app-center-for-react-native-async-mobile-devops/.

[61] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA, October 2018. USENIX Association.

[62] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: An opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 228–244. ACM, 2021.

[63] Joe H. Novak, Sneha Kumar Kasera, and Ryan Stutsman. Cloud functions for fast and robust resource auto-scaling. In *2019 11th International Conference on Communication Systems Networks (COMSNETS)*, pages 133–140, 2019.

[64] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[65] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *2016 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 122–132. IEEE, 2016.

[66] Haoran Qiu, Saurabh Jha, Subho S. Banerjee, Archit Patke, Chen Wang, Franke Hubertus, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Is function-as-a-service a good fit for latency-critical services? In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, WoSC '21, page 1–8. ACM, 2021.

[67] Ali Raza, Zongshun Zhang, Nabeel Akhtar, Vatche Isahagian, and Ibrahim Matta. LIBRA: An economical hybrid approach for cloud applications with strict SLAs. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 136–146, 2021.

[68] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 122–137. ACM, 2021.

[69] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 753–767. ACM, 2022.

[70] Marco Savi, Alessandro Banfi, Alessandro Tundo, and Michele Ciavotta. Serverless computing for NFV: Is it worth it? a performance comparison analysis. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 680–685, 2022.

[71] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1063–1075. ACM, 2019.

[72] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[73] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295. ACM, 2020.

[74] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. ACM, 2019.

[75] Myungjun Son, Shruti Mohanty, Jashwant Raj Gunasekaran, Aman Jain, Mahmut Taylan Kandemir, George Kesidis, and Bhuvan Urgaonkar. Splice: An automated framework for cost-and performance-aware blending of cloud services. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 119–128, 2022.

[76] Chris Spencer. CriticalPath Python package, Accessed: 2023-01-05. https://pypi.org/project/criticalpath/.

[77] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2019.

[78] Amoghavarsha Suresh and Anshul Gandhi. Servermore: Opportunistic execution of serverless functions in the cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 570–584. ACM, 2021.

[79] Ajay Tirumala. Iperf: The TCP/UDP bandwidth measurement tool. *http://dast. nlanr. net/Projects/Iperf/*, 1999.

[80] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. SmartHarvest: Harvesting idle CPUs safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 1–16. ACM, 2021.

[81] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu 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*, ASPLOS '22, page 768–781. ACM, 2022.

[82] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. LAVEA: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17. ACM, 2017.

[83] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. Accelerating serverless computing by harvesting idle resources. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 1741–1751. ACM, 2022.

[84] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[85] Lu Zhang, Weiqi Feng, Chao Li, Xiaofeng Hou, Pengyu Wang, Jing Wang, and Minyi Guo. Tapping into NFV environment for opportunistic serverless edge function deployment. *IEEE Transactions on Computers*, 71(10):2698–2704, 2022.

[86] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739. ACM, 2021.

[87] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. BeeHive: Sub-second elasticity for web services with semi-FaaS execution. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 74–87. ACM, 2023.

A Appendix

A.1 Dynamic Concurrency Feedback Details

As mentioned in §7.3, the execution agent uses a feedback mechanism to dynamically set the concurrency limit. Execution time changes during the time (ΔExecTime), which is used as feedback in Equation 2, is calculated as:

$$\overline{E_n}^t \leftarrow \text{Current Average Execution Time for Function}_n$$
$$E_n{}^t \leftarrow \text{Current Execution Time for Function}_n$$
$$\overline{E_n}^{t+1} = 0.8 \times \overline{E_n}^t + 0.2 \times E_n{}^t$$
$$\Delta E_n{}^t = \overline{E_n}^t - E_n{}^{t-1}$$
$$\Delta\text{ExecTime} = \sum_{n \in \text{functions}} \Delta E_n{}^t$$

In order to prevent frequent changes in the concurrency limit, we quantized the value of the μ calculated by Equation 2 into $[0.33, 0.66, 1]$. The weight of the exponential term in Equation 2 (0.03) was determined empirically, considering this quantization. Figure 14 shows the feedback mechanism used for setting the concurrency limit.

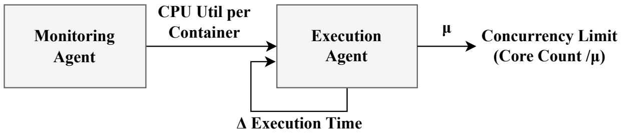


Figure 14: The feedback mechanism for dynamic concurrency.

A.2 Sample DAG JSON description file

```
{
"workflow": "Text2SpeechWorkflow",
"workflowFunctions": [
  "Text2SpeechWorkflow_GetInput",
  "Text2SpeechWorkflow_TransferInput",
  "Text2SpeechWorkflow_Profanity",
  "Text2SpeechWorkflow_Text2Speech",
  "Text2SpeechWorkflow_Conversion",
  "Text2SpeechWorkflow_Compression",
  "Text2SpeechWorkflow_MergeFunction",
  "Text2SpeechWorkflow_Censor"
],
"initFunc":"Text2SpeechWorkflow_GetInput",
"successors": [
  ["Text2SpeechWorkflow_TransferInput"],
  ["Text2SpeechWorkflow_Profanity",
  "Text2SpeechWorkflow_Text2Speech"],
  ["Text2SpeechWorkflow_MergeFunction"],
  ["Text2SpeechWorkflow_Conversion"],
  ["Text2SpeechWorkflow_Compression"],
  ["Text2SpeechWorkflow_MergeFunction"],
  ["Text2SpeechWorkflow_Censor"],
  []
],
"predecessors": [
  [],
  ["Text2SpeechWorkflow_GetInput"],
  ["Text2SpeechWorkflow_TransferInput"],
  ["Text2SpeechWorkflow_TransferInput"],
  ["Text2SpeechWorkflow_Text2Speech"],
  ["Text2SpeechWorkflow_Conversion"],
  ["Text2SpeechWorkflow_Compression",
  "Text2SpeechWorkflow_Profanity"],
  ["Text2SpeechWorkflow_MergeFunction"]
]
}
```

Figure 15: A sample DAG JSON description file.