Freyr$^+$: Harvesting Idle Resources in Serverless Computing via Deep Reinforcement Learning

Hanfei Yu, Hao Wang, *Member, IEEE,* Jian Li, *Member, IEEE,* Xu Yuan, *Senior Member, IEEE,* Seung-Jong Park, *Member, IEEE*

Abstract—Serverless computing has revolutionized online service development and deployment with ease-to-use operations, auto-scaling, fine-grained resource allocation, and pay-as-you-go pricing. However, a gap remains in configuring serverless functions—the actual resource consumption may vary due to function types, dependencies, and input data sizes, thus mismatching the static resource configuration by users. Dynamic resource consumption against static configuration may lead to either poor function execution performance or low utilization. This paper proposes *Freyr*$^+$, a novel resource manager (RM) that dynamically harvests idle resources from over-provisioned functions to accelerate under-provisioned functions for serverless platforms. *Freyr*$^+$ monitors each function's resource utilization in real-time and detects the mismatches between user configuration and actual resource consumption. We design deep reinforcement learning (DRL) algorithms with attention-enhanced embedding, incremental learning, and safeguard mechanism for *Freyr*$^+$ to harvest idle resources safely and accelerate functions efficiently. We have implemented and deployed a *Freyr*$^+$ prototype in a 13-node Apache OpenWhisk cluster using AWS EC2. *Freyr*$^+$ is evaluated on both large-scale simulation and real-world testbed. Experimental results show that *Freyr*$^+$ harvests 38% of function invocations' idle resources and accelerates 39% of invocations using harvested resources. *Freyr*$^+$ reduces the 99th-percentile function response latency by 26% compared to the baseline RMs.

Index Terms—Serverless computing, resource harvesting, reinforcement learning, attention, incremental learning

1 Introduction

Serverless computing, known as the next-generation cloud computing, has extensively simplified the way that developers access cloud resources. A wide spectrum of cloud applications, including web services [2], video processing [3, 4], data analytics [5, 6], and machine learning [7, 8] have been running on existing serverless computing platforms, such as AWS Lambda, Google Cloud Functions, and Azure Functions, with automated resource provisioning and management. By decoupling traditional monolithic cloud applications into inter-linked microservices executed by stateless *functions*, serverless computing frees developers from infrastructure management and administration with fine-grained resource provisioning, auto-scaling, and pay-as-you-go billing [9].

Existing serverless computing platforms enforce *static* resource provisioning for functions. For example, AWS Lambda allocates function CPU cores in a fixed proportion to the memory size configured by users, leading to either CPU over-provisioned or under-provisioned for the function execution. Therefore, serverless service providers are enduring poor resource utilization due to users' inappropriate function configuration—some functions are assigned with more resources than they need [10]. The high concurrency and fine-grained resource isolation of serverless computing further amplify such inefficient resource provisioning.

A few recent studies attempted to address the above issues. Some researchers proposed to maximize resource utilization and reduce the number of cold-starts by predicting the keep-alive windows of individual serverless functions [11, 12]. Fifer [10] incorporated the awareness of function dependencies into the design of a new resource manager to improve resource utilization. COSE [13] attempts to use Bayesian Optimization to seek for the optimal configuration for functions. Furthermore, several works [14, 15, 16] aimed to accelerate functions and improve resource efficiency by adjusting CPU core allocations for serverless functions in reaction to their performance degradation during function executions.

However, none of the existing studies has *directly* tackled the low resource efficiency issue raised by the inappropriate function configurations. There are three critical challenges from the perspective of serverless service providers to address this issue. *First*, a user function is secured as a black box that shares no information about its internal code and workloads, making it hardly possible for the serverless system to estimate the precise resource demands of user functions. *Second*, decoupling monolithic cloud applications to serverless computing architectures generates a variety

- *Hanfei Yu and Hao Wang are with the Department of Electrical and Computer Engineering, Stevens Institute of Technology, Hoboken, New Jersey, USA, 07030.*
  E-mail: hyu42@stevens.edu, hwang9@stevens.edu.
- *Jian Li is with the Departments of Applied Mathematics and Statistics & Computer Science in the College of Engineering and Applied Sciences at Stony Brook University, State University of New York (SUNY), Stony Brook, New York, USA, 11790.*
  E-mail: jian.li.3@stonybrook.edu.
- *Xu Yuan is with the Department of Computer and Information Sciences at the University of Deleware, Newark, Delaware, USA, 19716.*
  E-mail: xyuan@udel.edu.
- *Seung-Jong Jay Park is with the Computer Science Department of the College of Engineering and Computing at the Missouri University of Science & Technology, Rolla, Missouri, USA, 65409.*
  E-mail: seung-jong.park@mst.edu.

- *Preliminary results have been presented in the ACM WebConf'22 [1].*

of functions with diverse resource demands and dynamic input workloads. *Third*, the resource provisioning for serverless functions is fine-grained spatially (*i.e.*, small resource volumes) and temporally (*i.e.*, short available time).

In this paper, we address the aforementioned challenges by presenting *Freyr*$^+$, a new serverless resource manager (RM) that dynamically harvests idle resources to accelerate functions and maximize resource utilization. *Freyr*$^+$ estimates the CPU and memory saturation points respectively of each function and identifies whether a function is over-provisioned or under-provisioned. For those over-provisioned functions, *Freyr*$^+$ harvests the wasted resources according to their saturation points; for those under-provisioned functions, *Freyr*$^+$ tries to accelerate them by offering additional, and just-in-need allocations to approach saturation points.

We apply an experience-driven algorithm to identify functions over-supplied and under-supplied by monitoring a series of performance metrics and resource footprints, including CPU utilization, memory utilization, and function response latency to estimate the actual resource demands of running functions. Instead of inputting raw state features, *Freyr*$^+$ leverages an attention-enhanced embedding to automatically extract critical information and prioritize the features in the latent space using attention mechanism [17], which improves the training efficiency and generality of the reinforcement learning agent. To deal with the highly volatile environment of serverless computing and large numbers of concurrent functions, we propose to apply the Proximal Policy Optimization (PPO) algorithm [18] to learn from the realistic serverless system and make per-invocation resource adjustments. We equip *Freyr*$^+$ with incremental learning to quickly adapt to environmental changes, such as function update and new function deployment, which commonly occur in serverless computing. Besides, we design a safeguard mechanism for safely harvesting idle resources without introducing any performance degradation to function executions that have resource harvested.

We implement *Freyr*$^+$ based on Apache OpenWhisk [19], a popular open-source serverless computing platform. We develop a Deep Reinforcement Learning (DRL) model and training algorithm using PyTorch and enable multi-process support for concurrent function invocations. We evaluate *Freyr*$^+$ with the other three baselines on large-scale simulation and an OpenWhisk cluster using realistic serverless workloads. Compared to the default resource manager in OpenWhisk, *Freyr*$^+$ reduces the 99th-percentile function response latency of invocations[1] by 26%. Particularly, *Freyr*$^+$ harvests idle resources from 38% of function invocations while accelerating 39% on the OpenWhisk cluster. Notably, *Freyr*$^+$ only degrades a negligible percentage of function performance under the system performance variations of the OpenWhisk cluster.

2 Background and Motivation

This section first introduces the status quo of resource provisioning and allocation in serverless computing. Then, we use real-world experiments to demonstrate that serverless functions can easily become under-provisioned or over-provisioned, and motivate the necessity to accelerate under-provisioned functions and optimize resource utilization by harvesting idle resources at runtime.

2.1 Resource Provisioning and Allocation in Serverless Computing

Existing serverless computing platforms (*e.g.*, AWS Lambda, Google Cloud Functions, and Apache OpenWhisk) request users to define memory up limits for their functions and allocate CPU cores according to a fixed proportion of the memory limits [19, 20, 21, 22]. Obviously, the fixed proportion between CPU and memory allocations leaves serverless functions either under-provisioned or over-provisioned because functions' CPU and memory demands differ significantly.

Further it is non-trivial for users to accurately allocate appropriate amounts of resource for their functions [13, 15] due to various function types, dependencies, and input sizes. Users are prone to oversize their resource allocation to accommodate potential peak workloads and failures [15, 23]. Finally, users' inappropriate resource allocations and providers' fixed CPU and memory provisioning proportion jointly degrade the resource utilization in serverless computing as resources allocated to functions remain idle.

Integrating *Freyr*$^+$ to existing serverless computing systems, such as OpenWhisk, AWS Lambda, and Google Cloud Functions, leads to merits for both service providers and users. For service providers, *Freyr*$^+$ carefully harvests idle resources and reuses them to accelerate function invocations, which improves the overall serverless platform's resource utilization. For users who mistakenly configured insufficient resource allocation for their functions, *Freyr*$^+$ transparently brings potential performance improvement (*i.e.*, faster function executions) using harvested idle resources without significantly degrading other users' performance.

2.2 Resource Saturation Points

We further demonstrate how easily a serverless function becomes under-provisioned or over-provisioned by introducing a new notion of *saturation points*. Given a function and an input size, there exists a resource allocation *saturation point*—allocating resource beyond this point can no longer improve the function's performance, but allocating resource below this point severely degrades the performance.

We profile the saturation points of two applications: email generation (EG) and K-nearest neighbors (KNN), representing two popular serverless application categories: web applications and machine learning (ML), respectively. We identify the allocation saturation points of CPUs and memory separately by measuring the response latency of functions allocated with different number of CPU cores and different sizes of memory. When adjusting a function's CPU (memory) allocation, we fix its memory (CPU) allocation to 1,024 MB (8 cores).

Figure 1 shows that saturation points vary from functions and input sizes. It is non-trivial for users to identify the

1. In this paper, a function denotes an executable code package deployed on serverless platforms, and a function invocation is a running instance of the code package.

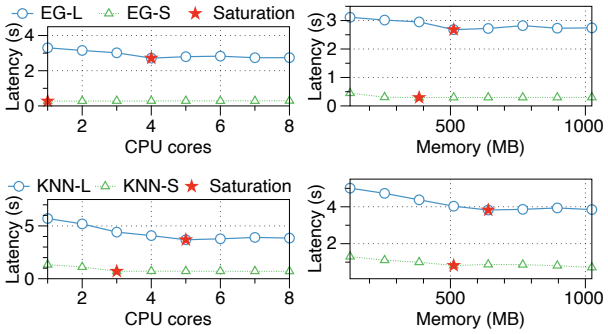
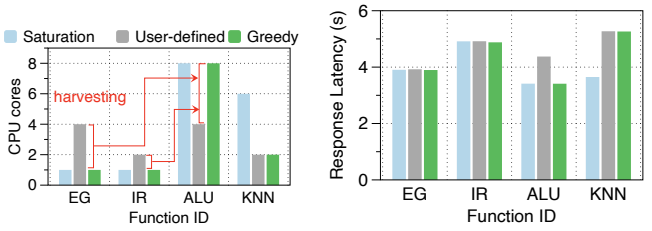


Fig. 1: Saturation points of EG and KNN with small (S) and large (L) workload sizes. EG-S (L) generates 1K (10K) emails, and KNN-S (L) inputs 2K (20K) data samples.



(a) CPU allocation (b) Function response latency

Fig. 2: The CPU allocation and response latency of four real-world functions: EG, image recognition (IR), arithmetic logic units (ALU), and KNN, where the EG generates 100K emails, the IR classifies ten images, the ALU calculates 20M loops, and the KNN inputs 20K data samples.

saturation points for every function with specific input sizes in their applications. Particularly, serverless functions are typically driven by events with varying input sizes. Without dynamic and careful resource allocations, functions tend to become either over-provisioned or under-provisioned.

2.3 The Need for Harvesting Idle Resources

Resource harvesting is a common methodology in virtual environments that increases resource utilization by reallocating idle resources to under-provisioned services without degrading the performance of services being harvested [24, 25, 26].

To motivate the need for dynamic resource harvesting in serverless computing, we compare the function response latency achieved by the default resource manager (Fixed RM) and greedy resource manager (Greedy RM) when executing four real-world serverless functions. The Fixed RM simply accepts and applies a fixed resource allocation pre-defined by users, such as the RM in OpenWhisk and AWS Lambda. The Greedy RM dynamically harvests CPU cores from functions over-provisioned and assigns the harvested CPU cores to functions under-provisioned in a first-come-first-serve manner based on the estimated function saturation points learned from functions' recent resource utilization (details in Section 6). In this experiment, we collect historical resource utilizations of four functions and profile their saturation points.

Figure 2(a) shows the Greedy RM accelerates the ALU by harvesting three CPU cores from the EG (*i.e.*, the EG function invocation) and one CPU core from the IR. Though the KNN is also under-provisioned, the Greedy RM assigns all harvested CPU cores to the ALU since the ALU is invoked before the KNN. As a comparison, Figure 2 also plots the saturation points of each function invocation and their response latency when allocated with saturated resources. Figure 2(b) shows the Greedy RM can increase resource utilization and accelerate under-provisioned functions without sacrificing over-provisioned functions' performance in the motivation scenario.

2.4 Deep Reinforcement Learning (DRL)

Due to the volatility and burstiness of serverless computing, it is non-trivial to accurately estimate the saturation points based on functions' recent resource utilization, and the greedy resource harvesting and re-assignment can hardly minimize the overall function response latency. Thus, we propose to utilize reinforcement learning (RL) algorithms to learn the optimal resource harvesting and re-assignment strategies.

At every timestep t, the agent is in a specific state s_t, and evolves to state s_{t+1} according to a Markov process with the transition probability $\mathbb{P}(s_t, a_t, s_{t+1})$ when action a_t is taken [27]. The immediate reward for the agent to take action a_t in state s_t is denoted as r_t. The goal of the agent is to find a policy π that makes decisions regarding what action to take at each timestep, $a_t \sim \pi(\cdot|s_t)$, so as to maximize the expected cumulative rewards, $\mathbb{E}_\pi[\sum_{t=1}^{\infty} \gamma^{t-1} r_t]$, where $\gamma \in (0, 1]$ is a discount factor.

To capture the patterns of real-world systems and address the curse-of-dimensionality, deep reinforcement learning (DRL) has been introduced to solve scheduling and resource provisioning problems in distributed systems [28, 29, 30, 31], where deep neural networks serve as the *function approximators* that describe the relationship between decisions, observations, and rewards. We design a lightweight score network as the approximator in DRL algorithms, which introduces negligible inference overheads in real-time serving (Section 6.7).

2.5 Incremental Learning

DRL unearths optimal resource harvesting and reassignment strategies for function invocations, though training a performant DRL agent is extremely time-consuming. The interaction between the DRL agent and realistic serverless computing systems contributes to most of the training time. For example, before the agent observes the next state, it has to wait until the completion of system operations (*e.g.*, container preparation and function execution).

Any environmental changes (*i.e.*, new functions deployment) may enforce retraining of the DRL agent, which makes DRL-based solutions inefficient for serving highly-volatile serverless functions. Thus, *Freyr*$^+$ should be carefully designed to adapt to dynamic changes in serverless platforms and avoid retraining from scratch. We introduce *incremental learning* for *Freyr*$^+$ to swiftly accommodate environmental changes. Incremental learning refers to a method that enables ML models to take input data continuously to extend the existing model's knowledge, *i.e.*, to further train the model. ML algorithms leverage incremental learning to adapt to new input data [32, 33, 34]. We design

a novel score network for *Freyr*$^+$ to share parameters of neural networks among different functions, thus supporting incremental learning inherently. With incremental learning, *Freyr*$^+$ is evaluated to achieve sub-optimal performance for harvesting and accelerating functions in a changing environment while converging to optimal performance in several training episodes (Section 6.5).

3 Overview

3.1 Design Challenges

Unlike long-running VMs with substantial historical traces for demand prediction and flexible time windows for resource harvesting, function executions in serverless computing are highly concurrent, event-driven, and short-lived with bursty input workloads [35], making it hardly practical to reuse the existing VM resource harvesting methods. To enable efficient and safe resource harvesting and performance acceleration in serverless computing, *Freyr*$^+$'s design tackles three key challenges:

Volatile and bursty serverless environments. The heterogeneity of serverless functions, the high concurrency of invocation events, and the burstiness of input workloads jointly make it non-trivial to accurately determine whether a function execution has idle resources to be harvested. Besides, serverless functions are sensitive to the latency introduced by resource harvesting and re-assignment due to their short lifetime and event-driven nature.

Huge space of harvesting and re-assignment decisions. Unlike the default resource managers that enforce a fixed proportion between the CPU and memory allocations, we decouple the resource provisioning for CPU and memory for more accurate resource harvesting and re-assignment, leading to a two-dimensional resource pool for *Freyr*$^+$ to seek for the optimal resource allocation. This is an immense action space for the DRL agent. For example, AWS Lambda allows any memory sizes between 128 MB and 10,240 MB and up to 6 CPU cores—60,672 choices in total. Such a huge action space complicates the DRL algorithm design and extensively increases the computation complexity to train the DRL agent.

Undeserved performance degradation. *Freyr*$^+$ harvests resources from invocations deemed as over-provisioned to improves the entire workload. One necessary requirement is to prevent the performance of those functions from degrading. In this paper, *undeserved performance degradation* is defined as an invocation's performance degrades when the resource it receives at runtime is below the user-defined allocation. It is vital to guarantee that harvested functions have no significant performance degradation.

3.2 *Freyr*$^+$'s Architecture

Freyr$^+$ is a resource manager in serverless platforms that dynamically harvests idle resources from over-provisioned function invocations and reassign the harvested resources to accelerate under-provisioned function invocations. It is located with the controller of a serverless computing framework and interacts with the container system (*e.g.*, Docker [36]) that executes function invocations.

Figure 3 shows an overview of *Freyr*$^+$'s architecture. First, concurrent function requests arrive at the frontend to

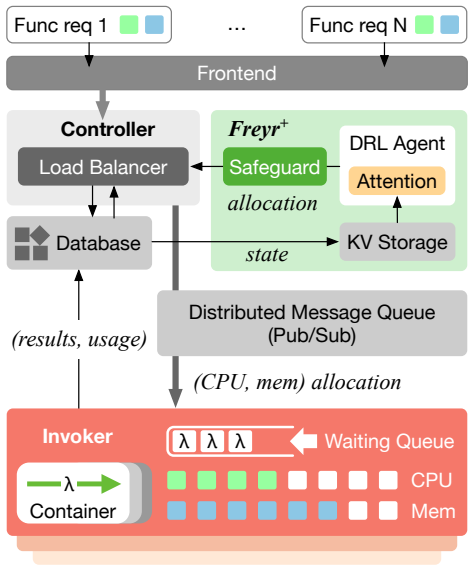


Fig. 3: *Freyr*$^+$'s architecture.

invoke specific functions with user-defined resource allocations. The controller admits the function requests, registers their configurations, and schedules them to the invokers. Before the function executions, *Freyr*$^+$ inputs observations from serverless platform database and makes resource harvesting and re-assignment decisions. The controller enforces invokers with the decisions when executing function invocations.

To handle the **volatile and bursty serverless environments**, *Freyr*$^+$ is designed to be event-driven with multi-process support that the arrival of a function request triggers *Freyr*$^+$ to make resource harvesting decisions. To shrink the **huge decision space**, *Freyr*$^+$ trains a score network to justify each allocation option of function invocations, converting the action space of all potential allocation options to a score for individual allocation option. *Freyr*$^+$ evaluates the score of each allocation option using this score network and enforces the allocation option with the highest score. To avoid **potential performance degradation** of functions with resources harvested, *Freyr*$^+$ applies a *safeguard mechanism* to prevent those potentially dangerous allocation options and guarantees the performance of every function invocation within a workload. The safeguard examines whether the allocation decision made by the DRL agent is below a function's historical resource usage peak. Besides, the safeguard monitors the function's runtime resource utilization and returns all harvested resources by calling a *safeguard invocation* when its resources appear to be fully utilized.

4 Design

4.1 Problem Formulation

We consider a serverless platform that handles a workload W with multiple concurrent function invocations. Let v denote a function invocation in W. We assume the response latency l of v is dominated by CPU and memory. Each function invocation v has a resource allocation $p = (p_c, p_m)$, where p_c and p_m denote a set of CPU and memory resources, respectively. We assume p is non-preemptive and fixed when the platform is executing v, *i.e.*, p is consistently

provisioned to v until the execution completes. Thus, we define the relationship between the response latency and the resource allocation as: $l = B(p)$. Section 2.2 demonstrates that a function invocation has a pair of saturation points for CPU and memory denoted by $p^{\Xi} = (p_c^{\Xi}, p_m^{\Xi})$, respectively.

The platform determines whether it can harvest or accelerate a function invocation v by comparing p with p^{Ξ}: if $p_c^{\Xi} < p_c$ ($p_m^{\Xi} < p_m$), v has idle CPU (memory), the platform can harvest at most $p_c - p_c^{\Xi}$ resources without increasing response latency l; if $p_c^{\Xi} > p_c$ ($p_m^{\Xi} > p_m$), the allocation of v hasn't saturated, the platform can provide v with at most $p_c^{\Xi} - p_c$ resources to improve the performance of v, *i.e.*, reduce response latency l. Thus for CPU or memory, function invocations in a workload W can be classified into three groups of invocations: $W = W_h + W_a + W_d$, where W_h denotes the set of invocations that can be harvested, W_a denotes the set of invocations that can be accelerated, and W_d denotes the set of invocations which have descent user configurations ($p^{\Xi} = p$).

We define a *speedup* value as the performance metric to avoid prioritizing long invocations while keeping short invocations starving. Recall that W denotes the workload, v denotes a function invocation in W. Function invocations arrive at the platform in a sequential order. At the first invocation of a function, the platform captures the response latency l^b with resources (p_c^b, p_m^b) configured by the user and employs it as a baseline denoted by b. When i-th invocation completes execution, the platform captures the response latency l^i of it. The speedup of the i-th invocation is calculated as

$$speedup := \frac{l^b - l^i}{l^b}. \tag{1}$$

We normalize the response latency of each invocation with baseline latency of user configuration. Intuitively, the speedup indicates how an invocation performs regardless of its duration length. A positive speedup value means the invocation is accelerated, otherwise the performance is degraded. We design the speedup value (Equation 1) as a unified metric for evaluation across different allocation schemes of functions in our experiments, and the speedup is embedded into the reward design (Section 4.5) of Freyr+ to guide the DRL decisions.

A function invocation may be accelerated while being harvested at the same time (*e.g.*, $p_c^{\Xi} < p_c$ while $p_m^{\Xi} > p_m$). In this case, the speedup is a mixed result. For individual invocations, we only focus on the performance regardless of details of resource allocation, *i.e.*, the invocation is good as long as it yields positive speedup. We use average speedup to measure how well a workload is handled by the platform with harvesting and acceleration. Hence, the goal is to find a set of resource allocation $p = (p^1, p^2, ..., p^{|W|})$ which maximizes the average speedup of a workload, defined as

$$\begin{aligned} avg\_speedup :=& \frac{1}{|W|}\sum_{i=1}^{|W|}\frac{l^b - l^i}{l^b} \\ =& \frac{1}{|W|}\sum_{i=1}^{|W|}\frac{B(p^b) - B(p^i)}{B(p^b)} \end{aligned}$$

However, as introduced in Section 2.2, estimating varying saturation points of sequential function invocations posts a challenging sequential decision problem. The complex mapping from set of p to objective average speedup can hardly be solved by existing deterministic algorithms. Hence, we opt for DRL and propose *Freyr*$^+$, which learns to optimize the problem by replaying experiences through training. *Freyr*$^+$ observes information from platform level and function level in real time. Figure 4 depicts how *Freyr*$^+$ estimates CPU/memory saturation points. Given a function invocation, we encode every possible CPU and memory option into a scalar value representing the choice.

TABLE 1: The observation state space of the DRL agent.

Platform State	`avail_cpu`, `avail_mem` `inflight_request_num`
Function State	`avg_cpu_peak`, `avg_mem_peak`, `avg_interval`, `avg_execution_time`, `baseline`

4.2 Attention-enhanced Information Pre-processing

When allocating resources for a function invocation, *Freyr*$^+$ collects information from two levels in Table 1: platform level and function level. Specifically, for the platform, *Freyr*$^+$ captures the number of invocations remaining in the system (*i.e.*, `inflight_request_num`), available CPU cores, and available memory. For the incoming function, *Freyr*$^+$ queries invocation history of the function which records average CPU peak, average memory peak, average inter-arrival time (IAT), average execution time, and baseline execution time (*i.e.*, `baseline`) with user-requested resources. We embed all information and the potential configuration options together into a flat state vector as input to *Freyr*$^+$ agent.

Once collecting such information and encapsulating the state vector, we enhance the plain state features using the *attention mechanism* in the latent space. *Freyr*$^+$ automatically learns the importance of individual state features by assigning a learnable weight using soft attention. Intuitively, the idea behind the attention mechanism is to learn the relative importance of each feature in the state space. To calculate the attention weight, we treat the feature as a query x and the rest of the features as a key matrix Y. The attention weight **Attn** is calculated by

$$\mathbf{Attn}(x, Y) = \texttt{softmax}(\frac{xY^{\mathsf{T}}}{d}), \tag{2}$$

where d is the dimension of the query x. Given a state $\boldsymbol{s} = \{h^1, \ldots, h^j, \ldots, h^J\}$, we denote h^j as a feature and J is the total number of features in $\boldsymbol{s}$, where $j \in J$. Referring to the Equation 2, *Freyr*$^+$ learns an attention weight $\mathbf{Attn}^j$ for each h^j by replacing x and Y^{T} with h^j and $\boldsymbol{s}^{\neg j} = \{h^1, \ldots, h^{j-1}, h^{j+1}, \ldots, h^J\}$, given by

$$\mathbf{Attn}^j = \mathbf{Attn}(Z_x^j h^j, Z_y^j \boldsymbol{s}^{\neg j}), \tag{3}$$

where Z_x^j and Z_y^j are learnable parameters for the query h^j and the key matrix $\boldsymbol{s}^{\neg j}$, respectively. Finally, according to the Equation 3, the attention-enhanced state $\tilde{\boldsymbol{s}}$ is obtained by weighting the original state $\boldsymbol{s}$ with the attention output $\mathbf{Attn}_{\boldsymbol{s}} = \{\mathbf{Attn}^1, \ldots, \mathbf{Attn}^j, \ldots, \mathbf{Attn}^J\}$, given by

$$\tilde{\boldsymbol{s}} = \boldsymbol{s} \times \mathbf{Attn}_{\boldsymbol{s}} = \{\mathbf{Attn}^1 h^1, \ldots, \mathbf{Attn}^j h^j, \ldots, \mathbf{Attn}^J h^J\}. \tag{4}$$

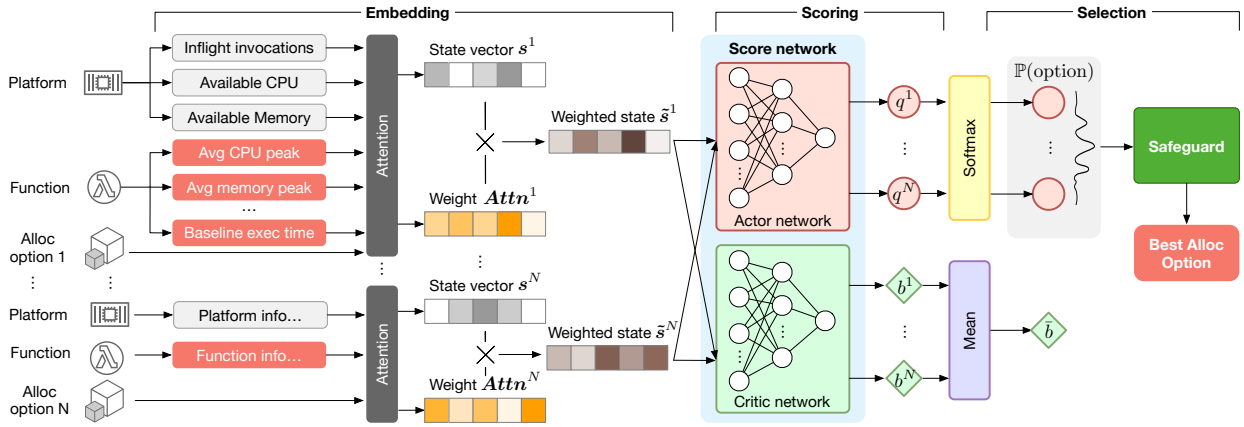


Fig. 4: The workflow of *Freyr*$^+$.

We feed the attention-enhanced state $\tilde{s}$ to *Freyr*$^+$ agent for choosing an allocation. We illustrate the information embedding process in Figure 4.

4.3 Score Network

Freyr$^+$ uses a *score network* to calculate the priority of selecting potential resource allocation options. Figure 4 visualizes the policy network of *Freyr*$^+$ agent, and illustrates the workflow of how the agent selects the best allocation option based on states. At time t, a function invocation arrives at the platform which has in total N potential resource configuration options. After embedding procedure, *Freyr*$^+$ collects a batch of state vectors $\tilde{\boldsymbol{s}}_t = (\tilde{\boldsymbol{s}}_t^1, \ldots, \tilde{\boldsymbol{s}}_t^n, \ldots, \tilde{\boldsymbol{s}}_t^N)$, where $\tilde{\boldsymbol{s}}_t^n$ maps the state to the n-th option. *Freyr*$^+$ inputs s_t to the score network. We implement the score network using two neural networks, an *actor network* and a *critic network*. The actor network computes a score q_t^n, which is a scalar value mapped from the state vector $\tilde{\boldsymbol{s}}_t^n$ representing the priority to select configuration option n. Then *Freyr*$^+$ applies a `softmax` operation to the scores $(q_t^1, \ldots, q_t^n, \ldots, q_t^N)$ to compute the probability of selecting option n based on the priority scores, given by

$$\mathbb{P}_t(option = n) = \frac{\exp(q_t^n)}{\sum_{n=1}^{N} \exp(q_t^n)},$$

at time t. The critic network outputs a baseline value b_t^n for option n, the average baseline value $\bar{b}_t$ is calculated as

$$\bar{b}_t = \frac{1}{N} \sum_{n=1}^{N} b_t^n, \tag{5}$$

which is used to reduce variance when training *Freyr*$^+$. The whole operation of policy network is end-to-end differentiable.

The score network itself contains no manual feature engineering. *Freyr*$^+$ agent automatically learns to compute accurate priority score of allocation options through training. More importantly, *Freyr*$^+$ uses the same score network for all function invocations and all potential resource allocation options. By embedding options into state vectors, *Freyr*$^+$ can distinguish between different options and use the score network to select the best option. Reusing the score network reduces the size of networks and limits the action space of *Freyr*$^+$ agent significantly.

Algorithm 1: Safeguard mechanism atop *Freyr*$^+$.

```
while request_queue.notEmpty do
    function_id ← request_queue.dequeue()
    calibrate_baseline ← False
    last_request ← QueryRequestHistory(function_id)
    if last_request == None then
        /* Trigger safeguard */
        range ← [user_defined]
        calibrate_baseline ← True
    else
        threshold ← 0.8
        last_alloc ← last_request.alloc
        last_peak ← last_request.peak
        recent_peak ← GetRecentPeak(function_id)
        if last_peak < user_defined then
            /* Over-provisioned */
            if last_peak / last_alloc ≥ threshold then
                /* Trigger safeguard */
                range ← [user_defined]
                calibrate_baseline ← True
            else
                range ← [recent_peak + 1,
                 user_defined]
            end
        else
            /* Under-provisioned */
            range ← [recent_peak + 1,
             max_per_function]
        end
    end
    alloc_option ← Freyr+(function_id, range)
    Invoke(function_id, alloc_option,
     calibrate_baseline)
end
```

4.4 Safeguard

We design *Freyr*$^+$ to improve both over-provisioned and under-provisioned functions. However, when harvesting resources from functions deemed as over-provisioned, it is possible that *Freyr*$^+$ under-predicts their resource demands.

The performance of functions degrades when being over-harvested. We devise a safeguard mechanism atop *Freyr*$^+$ to regulate decisions by avoiding decisions that may harm performance and returning harvested resources immediately when detecting a usage spike. We use this safeguard mechanism to mitigate obvious performance degradation of individual functions.

Algorithm 1 summarizes the safeguard mechanism built atop *Freyr*$^+$. We refer safeguard invocation as invoking the function with user-defined resources. When there are no previous invocations, *Freyr*$^+$ triggers the safeguard to obtain resource usage and calibrate the baseline mentioned in Equation 1 (lines 5–7). For further invocations, *Freyr*$^+$ queries the history of function and polls the usage peak, allocation of the last invocation, and the highest peak since last baseline calibration (lines 10–12). *Freyr*$^+$ first checks current status of the function, *i.e.*, over-provisioned or under-provisioned (line 13). We assume functions with resource usage below 80% of user-requested level is over-provisioned. For over-provisioned (harvested) functions, *Freyr*$^+$ then checks the usage peak of last invocation (line 14). If the usage peak approaches 80% of allocation, we suspect there may be a load spike, which could use more resources than current allocation. This triggers the safeguard invocation and baseline re-calibration, *Freyr*$^+$ immediately returns harvested resource to the function at the next invocation (lines 15–16). If there is no usage spike, *Freyr*$^+$ is allowed to select an allocation option from recent peak plus one unit to a user-requested level (line 18). For under-provisioned functions, *Freyr*$^+$ is allowed to select from recent peak plus one unit to the maximum available level (line 21). After an allocation option is selected, *Freyr*$^+$ invokes the function and forwards the invocation to invoker servers for execution.

Section 4.4 presents a sensitivity analysis of safeguard thresholds and shows that the safeguard mechanism effectively regulates decisions made by *Freyr*$^+$ and protects performance of functions that have resources harvested.

4.5 Training the DRL Agent

Freyr$^+$ training proceeds in *episodes*. In each episode, a series of function invocations arrive at the serverless platform, and each requires a two-dimensional action to configure CPU and memory resources. When the platform completes all function invocations, the current episode ends. Let T denote the total number of invocations in an episode, and t_i denote the arrival time of the i-th invocation. We continuously feed *Freyr*$^+$ with a reward r after it takes an action to handle an invocation. Concretely, we penalize *Freyr*$^+$ with

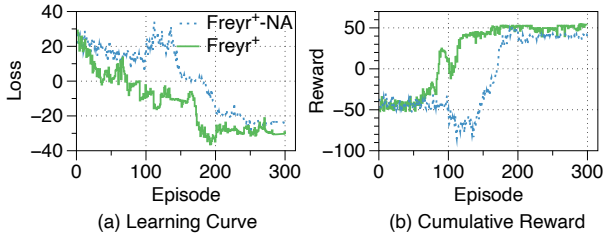
$$r_i = R_{(speedup>0)} - R_{(speedup<0)} + \sum_{k=1}^{|W|} \frac{l^b - l^k}{l^b},$$

after taking action on the i-th invocation, where W is the set of invocations that finish during the interval $[t_{i-1}, t_i)$, $\frac{l^b - l^k}{l^b}$ is the speedup of an invocation v^k finished during the interval $[t_{i-1}, t_i)$, and two constant summaries for awarding good and penalizing bad actions (*i.e.*, $R_{(speedup<0)}$ and $R_{(speedup>0)}$). The algorithm's goal is to maximize the expected cumulative rewards defined as

$$\mathbb{E}\left[\sum_{i=1}^{T} \gamma^{t-1}\left(R_{(speedup>0)} - R_{(speedup<0)} + \sum_{k=1}^{|W|} \frac{l^b - l^k}{l^b}\right)\right]. \quad (6)$$

We set the discount factor γ in Equation 6 to be 0.99 as commonly used [27]. Hence, *Freyr*$^+$ learns to maximize the overall *speedup* of the given workload.

We use the Algorithm 2 to train *Freyr*$^+$ with 4 epochs per surrogate optimization and a 0.2 clip threshold [18]. We update the policy network parameters using the AdamW optimizer [37] with a learning rate of 0.001. We train *Freyr*$^+$ with 300 episodes. The total training time is about 36 hours. Figure 5 shows the learning curve and cumulative rewards of *Freyr*$^+$ training on OpenWhisk testbed. In Figure 5(a), the descending loss trendline indicates that *Freyr*$^+$ gradually learns to make good resource management decisions. In Figure 5(b), the ascending trendline shows that *Freyr*$^+$ seeks to maximize the cumulative rewards through training.



(a) Learning Curve (b) Cumulative Reward

Fig. 5: The average loss and cumulative reward of *Freyr*$^+$ and *Freyr*$^+$-NA with 300-episode training on the OpenWhisk testbed, respectively. *Freyr*$^+$ converges faster and achieves higher cumulative reward with the attention-enhanced embedding.

4.6 The Training Algorithm

Freyr$^+$ uses a policy gradient algorithm for training. Policy gradient methods are a class of RL algorithms that learn policies by performing gradient ascent directly on the parameters of neural networks using the rewards received during training. When updating policies, large step sizes may collapse the performance, while small step sizes may decrease the sampling efficiency. We use the Proximal Policy Optimization (PPO) algorithms [18] to ensure that *Freyr*$^+$ takes appropriate step sizes during policy updates. More specifically, given a policy π_θ parameterized by θ, the PPO algorithm updates policies at the k-th episode via

$$\theta_{k+1} = \arg\max_{\theta} \mathop{\mathbb{E}}_{s,a\sim\pi_{\theta_k}} \Big[\mathbb{L}(s, a, \theta_k, \theta)\Big],$$

where $\mathbb{L}$ is the *surrogate advantage* [38], a measure of how policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy. We use the PPO-clip version of a PPO algorithm, where $\mathbb{L}$ is given by

$$\mathbb{L}(s, a, \theta_k, \theta) = \min\Big(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\ g(\epsilon, A^{\pi_{\theta_k}}(s, a))\Big),$$

and $g(\epsilon, A)$ is a clip operation defined as

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A, & \text{if } A \geq 0, \\ (1-\epsilon)A, & \text{otherwise,} \end{cases}$$

Algorithm 2: *Freyr*$^+$'s Training Algorithm.

Initial policy (actor network) parameters θ_0 and value function (critic network) parameters ϕ_0
for *episode* $k \leftarrow 0, 1, 2, \ldots$ **do**
  Run policy $\pi_k = \pi(\theta_k)$ in the environment until T-th invocation completes
  Collect set of trajectories $\mathbb{D}_k = \{\tau_i\}$, where $\tau_t = (\tilde{s}_t, a_t), t \in [0, T]$ and $\tilde{s}_t$ is computed via Equation 4
  Compute reward $\hat{r}_t$ via Equation 6
  Compute baseline value $\bar{b}_t$ via Equation 5
  Compute advantage $\hat{\mathbb{A}}_t = \hat{r}_t - \bar{b}_t$
  Update actor network by maximizing objective using stochastic gradient ascent:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathbb{D}_k|T} \sum_{\tau \in \mathbb{D}_k} \sum_{t=0}^{T} \mathbb{L}(\tilde{s}_t, a_t, \theta_k, \theta)$$

  Update critic network by regression on mean-squared error using stochastic gradient descent:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathbb{D}_k|T} \sum_{\tau \in \mathbb{D}_k} \sum_{t=0}^{T} (\bar{b}_t - \hat{r}_t)^2$$

end

where A is the advantage calculated as rewards r subtracted by baseline values b; ϵ is a hyperparameter that restricts how far the new policy is allowed to deviate from the old. Intuitively, the PPO algorithm sets a range for step sizes of policy updates, which prevents the new policy from deviating too much from the old (either positive or negative).

Algorithm 2 presents the training process of *Freyr*$^+$. For each episode, we record the whole set of trajectories including the states, actions, rewards, baseline values predicted by the critic network, and the logarithm probability of the actions for all invocations. After each training episode finishes, we use the collected trajectories to update the actor and critic networks.

4.7 Adapting to Environmental Changes

Functions on serverless platforms are highly volatile, where users may deploy new functions in any second. *Freyr*$^+$ can adapt to environmental changes promptly with the support of incremental learning. We design the score networks in Section 4.3 to enable *Freyr*$^+$ with incremental training. Upon new functions deployed by users, *Freyr*$^+$ restores the latest checkpointed agent model and incrementally trains the parameters of score networks by interacting with new functions. We treat changing function codebases or re-uploading existing functions as adding new functions for simplicity. Section 6.5 compares the performance and convergence of *Freyr*$^+$ with and without incremental learning. We show that *Freyr*$^+$ can promptly adapt to environmental changes with incremental learning, thus effectively avoiding the overhead of retraining.

5 Implementation Details

Apache OpenWhisk is an open-source, distributed serverless platform that powers IBM Cloud Functions [39]. Figure 3 illustrates the architecture of *Freyr*$^+$ based on OpenWhisk. OpenWhisk exposes an NGINX-based REST interface for users to interact with the platform. Users can create new functions, invoke functions, and query results of invocations via the frontend. The Frontend forwards function invocations to the Controller, which selects an Invoker (typically hosted using VMs) to execute invocations. The Load Balancer inside the Controller implements the scheduling logic by considering Invoker's health, available capacity, and infrastructure state. Once choosing an Invoker, the Controller sends the function invocation request to the selected Invoker via a Kafka-based distributed messaging component. The Invoker receives the request and executes the function using a Docker container. After finishing the function execution, the Invoker submits the result to a CouchDB-based Database and informs the Controller. Then the Controller returns the result of function executions to users synchronously or asynchronously. Here we focus on resource management for containers.

We modify the following modules of OpenWhisk to implement our resource manager:

Frontend: Initially, OpenWhisk only allows users to define the memory limit of their functions and allocates CPU power proportionally based on memory. To decouple CPU and memory, we add a CPU limit and enable the Frontend to take CPU and memory inputs from users. Users are allowed to specify CPU cores and memory of their functions. The Frontend forwards CPU and memory limits to the Controller.

Controller: The Load Balancer makes scheduling decisions for the Controller. When selecting an Invoker, the Load Balancer considers available memory of Invokers. We modify the Load Balancer also to check available CPU cores of Invokers—the Load Balancer selects Invokers with enough available CPU cores and memory to execute function invocations.

Invoker: The Invoker uses a semaphore-based mechanism to control containers' access to available memory. We apply the same mechanism to control access to available CPU cores independently.

Container: By default, OpenWhisk uses `cpu-shares` parameter to regulate CPU power of containers. When plenty of CPU cycles are available, all containers with `cpu-shares` use as much CPU as they need. While `cpu-shares` improves CPU utilization of Invokers, it can lead to performance variation of function executions. We change the CPU parameter to `cpus` which restricts how many CPU cores a container can use. This is aligned with the CPU allocation policy of AWS Lambda[2]. For each function invocation, we monitor the CPU cores and memory usage of its container using `cgroups`. We record the usage peak during function execution as history for *Freyr*$^+$ to query.

2. https://docs.aws.amazon.com/lambda/latest/dg/limits.html

TABLE 2: Characterization of training and testing invocation trace sets used in the simulation and OpenWhisk evaluation. Metrics include: total number of unique traces, total number of invocations (Invo.), average inter-arrival time (IAT), and requests per second.

Set	Traces	Invo.	Avg IAT (s)	Reqs/sec
SIM-train	15,427	83,521,859	0.71	1.39
SIM-test	1,000	85,470	0.69	1.42
OW-train	1,000	26,705	2.21	0.44
OW-test	10	268	2.20	0.45

TABLE 3: Characterizations of serverless applications used in OpenWhisk evaluation. (DH: Dynamic HTML, EG: Email Generation, IP: Image Processing, VP: Video Processing, IR: Image Recognition, KNN: K Nearest Neighbors, GD: Gradient Descent, ALU: Arithmetic Logic Units, MS: Merge Sorting, and DV: DNA Visualization.)

Function	Type	Dependency
DH	Web App	Jinja2, CouchDB
EG	Web App	CouchDB
IP	Multimedia	Pillow, CouchDB
VP	Multimedia	FFmpeg, CouchDB
IR	Machine Learning	Pillow, torch, CouchDB
KNN	Machine Learning	Scikit-learn, CouchDB
GD	Machine Learning	NumPy, CouchDB
ALU	Scientific	CouchDB
MS	Scientific	CouchDB
DV	Scientific	Squiggle, CouchDB

DRL agent: We implement the *Freyr*$^+$'s agent using two neural networks, each with an attention layer and two fully connected hidden layers. The attention layer has 32 neurons, and both two hidden layers have 64 neurons. Each neuron uses `Tanh` as its activation function. The agent is implemented in 2K lines of Python code using PyTorch [40]. *Freyr*$^+$ is lightweight because the policy network consists of only 1858 parameters (12 KB in total). Mapping a state to an action takes less than 20 ms, which is sufficiently negligible for serverless workloads.

6 Evaluation

We implement *Freyr*$^+$ with 6K lines of Scala code in Apache OpenWhisk [19] and deploy it to a realistic OpenWhisk cluster. We train and evaluate *Freyr*$^+$ using realistic workloads from public serverless benchmarks and invocation traces sampled from Azure Functions traces [12].

6.1 Methodology

We first describe our experimental methodology below.

Baselines. We compare *Freyr*$^+$ with three baseline RMs: 1) *OpenWhisk default RM*: the default RM of most existing serverless platforms (including) that allocates CPU cores in a fixed proportion to user-defined memory sizes. 2) *Greedy RM* detects a function's saturation points based on its historical resource usage by gradually decreasing (increasing) the allocation for an over-provisioned (under-provisioned) function in a fine-tuned and fixed step. Our implementation sets the detect step size one core and 128 MBs for CPU and memory, respectively. Besides, Greedy RM allocates resources to functions in a first-come-first-serve manner. 3) *ENSURE* [15] allocates memory resources as users request and adjusts the CPU cores for each function at runtime when detecting performance degradation.

Evaluation metrics. We use the *speedup* value defined in Section 4.1 to measure the performance of a function invocation. Function invocations with positive speedups have lower response latency. For resource harvesting, *Freyr*$^+$ aims to maximize the amount of harvested resources while having minimal impact on the performance of harvested functions. For resource re-assignment, *Freyr*$^+$ treats invocations with different lengths of response latency as the same by maximizing speedups, which improves the overall performance of the workload. We also report the details of undeserved performance degradation and 99th-percentile (P99) function response latency of the workload.

Testbeds. We evaluate *Freyr*$^+$ extensively using both large-scale simulation and a real OpenWhisk cluster. For simulation, we develop a simulative serverless computing environment based on OpenAI Gym [41], an open-source library for evaluating RL algorithms. We implement the simulator in Python with 3K lines of code, and various APIs for defining a customized cluster in the simulator. We configure our simulative cluster with 30 worker servers, each with eight CPU cores and 32 GBs of memory available for functions. Each function has maximum access to eight CPU cores and 1,024 MBs of memory. For OpenWhisk evaluation, we deploy and evaluate *Freyr*$^+$ on an OpenWhisk cluster with 13 physical servers using AWS EC2 [42]. Two of the servers host the OpenWhisk components, such as the frontend, the controller, the messaging queue, and database services. One deploys the *Freyr*$^+$ agent. The remaining ten servers serve as the invokers for executing functions. The server that hosts *Freyr*$^+$ agent is a c4.4xlarge instance, while the other 12 servers are c4.2xlarge instances. The server hosting *Freyr*$^+$ agent has 16 CPU cores and 64 GB memory, and each of the other 12 servers has eight CPU cores and 32 GB memory. Each function can be configured with eight CPU cores and 1,024 MB of RAM at most. Considering the serverless functions' short lifecycle, we monitor their CPU and memory usage per 100 ms and keep the historical resource usage in the Redis (*i.e.*, the KV store in Figure 3).

Invocation traces. We randomly sample four invocation trace sets consisting of over 16,000 unique functions from Azure Functions traces. Table 2 depicts the characteristics of four trace sets used in the simulation (SIM-train and SIM-test) and OpenWhisk evaluation (OW-train and OW-test) for training and testing, respectively. We develop a discrete event generator with Python to drive the experiments, which invokes functions according to the sampled traces.

Workload functions. We employ ten real-world functions from three serverless benchmark suites: SeBS [43], ServerlessBench [44], and ENSURE-workloads [15]. Table 3 describes the type and dependency of ten serverless applications from benchmark suites. DH downloads HTML templates, populates the templates based on input, and uploads them to CouchDB. EG generates emails based on the input and returns them to CouchDB. IP downloads images, resizes them, and uploads them to CouchDB. VP downloads videos, trims and tags them with a watermark,

and uploads them to CouchDB. IR downloads a batch of images, classifies them using ResNet-50, and uploads them to CouchDB. KNN downloads the dataset, performs the KNN algorithm on it, and uploads the result to CouchDB. GD performs three kinds of gradient descent based on input and uploads the result to CouchDB. ALU computes the arithmetic logic based on input and uploads the result to CouchDB. MS performs merge-sorting based on input and uploads the result to CouchDB. DV downloads a DNA sequence file, visualizes the sequence, and uploads the result to CouchDB. For DH, EG, IP, KNN, ALU, MS, and GD, each is initially configured with one CPU cores and 128 MB of memory; for VP, IR, and DV, each is initially configured with eight cores and 1,024 MB. We set the initial resource configuration of each function according to the default settings of the suites. We feed input data with dynamic patterns to functions for individual invocations. Specifically, IP and IR randomly sample 100 pictures from the CIFAR-100 dataset [45]. VP randomly samples 100 videos from the YouTube-8M dataset [46]. DV uses different genome sequences of Bacillus subtilis from the NCBI dataset [47]. ALU randomly samples two numbers ranging from 100 to 10,000 to perform computations. EG randomly generates 100 to 10,000 emails. GD performs a random number of optimization steps via SGD, RMSProp, and Adam. DH randomly renders HTML pages ranging from 10 to 10,000. KNN randomly generates a dataset size and the number of feature dimensions to perform predictions. MS sorts a list of numbers ranging from 10,000 to 1,000,000.

6.2 Simulation at Scale

We first trains *Freyr*$^+$ and evaluate *Freyr*$^+$ in the serverless computing simulator using SIM-train and SIM-test workloads from Table 2, respectively. We summarize the speedup and resource allocation of function invocations of the testing workload in Figure 6. In each subgraph, each point (*i.e.*, •, +, −, and ×) indicates a function invocation. The y-axis indicates the speedup values of function invocations, and the x-axis shows the CPU and memory allocation of function invocations relative to their user configurations. The negative CPU and memory values indicates that RMs harvest corresponding resources from those invocations, and the positive means that those invocations are provided with additional resources.

Overall performance. *Freyr*$^+$ outperforms baseline RMs with the best overall performance. For processing the same workload, *Freyr*$^+$ achieves a highest average speedup of 0.24, whereas Default RM, Greedy RM, ENSURE are 0, 0.10, and 0.13, respectively. Recall in Section 4.1, a higher speedup indicates a faster function response. Compared to the default RM in OpenWhisk, *Freyr*$^+$ provides an average of 24% faster function executions and 73% lower P99 response latency for the workload. *Freyr*$^+$ harvests idle resources from 42% of function invocations and accelerates 45% of invocations.

Harvesting and acceleration. Figure 6 shows the performance of 85,470 individual invocations processed by four RMs. Default RM has no resource adjustment during its workload processing. Greedy harvests an average of 2.2 cores and 284 MB from harvested invocations and

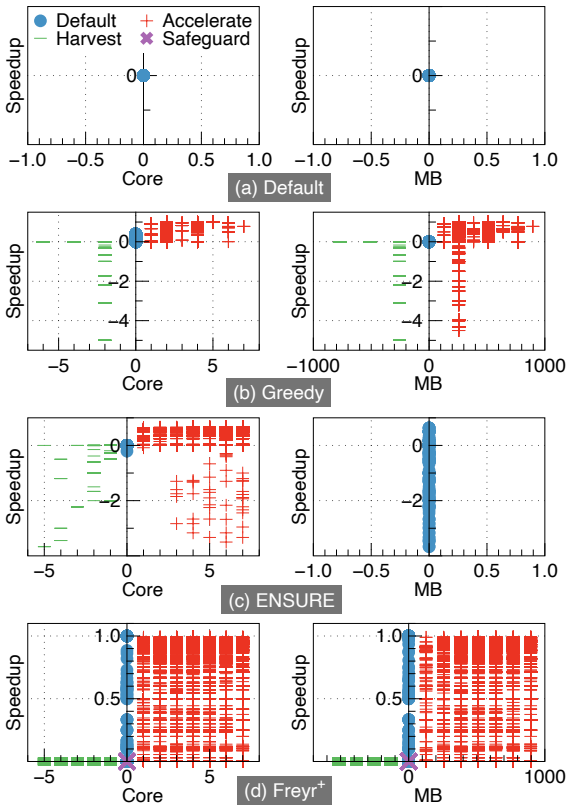


Fig. 6: Performance of individual invocations processed by four RMs in simulation. Default (•): invocations with user-requested allocation. Accelerate (+): invocations accelerated by supplementary allocation. Harvest (−): invocations with resource harvested. Safeguard (×): invocations protected by the safeguard.

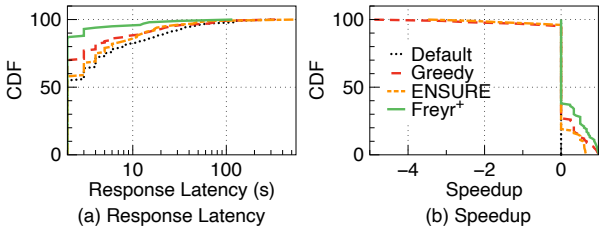


Fig. 7: The CDF of function response latency (left) and speedup (right) in simulation, respectively.

accelerates under-provisioned functions with an average of 2.0 cores and 318 MB. ENSURE's policy also harvests and accelerates invocations with CPU cores but makes no changes to memory resources. ENSURE harvests an average of 1.9 cores from over-provisioned functions and accelerates under-provisioned functions with an average of 2.3 cores. *Freyr*$^+$ harvests an average of 2.5 cores and 324 MB from over-provisioned functions and accelerates under-provisioned functions with an average of 3.7 cores and 523 MB. *Freyr*$^+$ re-assign harvested resources to accelerate under-provisioned invocations, which speeds up for under-provisioned function invocations up to 99%.

Undeserved performance degradation. Figure 6 shows that both Greedy RM and ENSURE severely degrade function execution performance since there are function invoca-

tions with large negative speedup values. Default RM has no performance degradation as it performs no harvesting or acceleration. Greedy RM degrades the performance of some harvested invocations over 5×. ENSURE degrades the performance of some harvested invocations over 3.5× when harvesting CPU cores. Compared to Greedy RM and ENSURE, *Freyr*$^+$ harvests idle resources perfectly from under-provisioned invocations without any performance degradation. When harvesting idle resources, *Freyr*$^+$ calls safeguard for 16.2% of invocations to avoid potential performance degradation due to usage spike.

P99 latency. *Freyr*$^+$ outperforms three baselines regarding both function response latency and speedup. Figure 7(a) shows the CDF of function response latency of the testing workload. *Freyr*$^+$ has a P99 function response latency of 32 seconds, whereas Default RM, Greedy RM and ENSURE are 121, 81, and 69 seconds, respectively. Figure 7(b) shows the CDF of the speedup of the testing workload. *Freyr*$^+$ maintains P99 speedups below 0 (*i.e.*, no undeserved performance degradation) for all invocations, whereas Greedy RM and ENSURE are -5.0 and -3.5, respectively. As the Default RM adjusts no resources, the speedup stays 0 for all percentile.

6.3 Evaluation on the OpenWhisk Testbed

We then trains and evaluate *Freyr*$^+$ in a distributed OpenWhisk cluster using OW-train and OW-test from Table 2, respectively. Same to the simulation, we also report the speedup and resource allocation of function invocations of the testing workload in Figure 8.

Overall performance. Aligned with the simulative results, *Freyr*$^+$ outperforms other baselines with the best overall performance. For processing the same workload, *Freyr*$^+$ achieves a highest average speedup of 0.21, whereas Default RM, Greedy RM, ENSURE are 0, 0.10, and 0.06, respectively. Compared to the default RM in OpenWhisk, *Freyr*$^+$ provides an average of 21% faster function executions and 27% lower P99 response latency for the same workload. *Freyr*$^+$ harvests idle resources from 38% of function invocations and accelerates 39% of invocations.

Harvesting and acceleration. Figure 8 shows the performance of 268 individual invocations processed by four RMs. Default RM has no resource adjustment during its workload processing. Greedy harvests an average of 2.7 cores and 368 MB from harvested invocations and accelerates under-provisioned functions with an average of 3 cores and 392 MB. ENSURE's policy also harvests and accelerates invocations with CPU cores but makes no changes to memory resources. ENSURE harvests an average of 2.4 cores from over-provisioned functions and accelerates under-provisioned functions with an average of 2.9 cores. *Freyr*$^+$ harvests an average of 3.1 cores and 480 MB from over-provisioned functions and accelerates under-provisioned functions with an average of 3.6 cores and 364 MB. *Freyr*$^+$ reassign harvested resources to accelerate under-provisioned invocations, which speeds up for under-provisioned function invocations up to 98%.

Undeserved performance degradation. Figure 8 shows that both Greedy RM and ENSURE severely degrade function execution performance as some function invocations

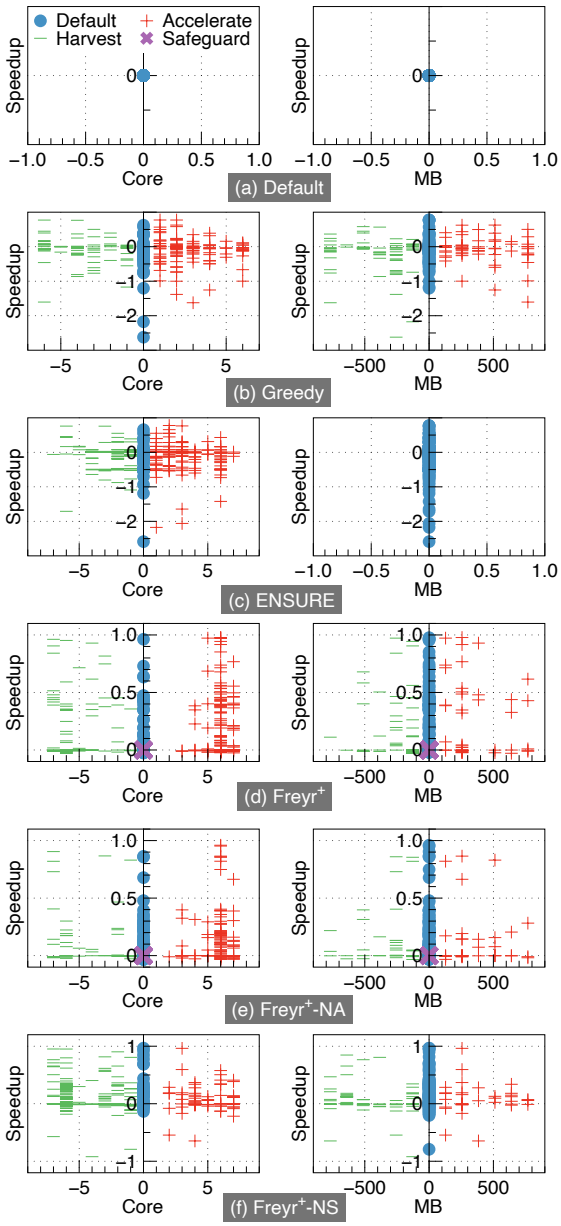


Fig. 8: Performance of individual invocations processed by six RMs in OpenWhisk evaluation.

have large negative speedup values. Default RM has no degradation as it performs no harvesting or acceleration. Greedy RM degrades the performance of some harvested invocations over 2.6×. ENSURE degrades the performance of some harvested invocations over 2.8× when harvesting CPU cores. Compared to Greedy RM and ENSURE, *Freyr*$^+$ rationally harvests idle resources from under-provisioned invocations, as the performance degradation of harvested invocations is limited within 3%. When harvesting idle resources, *Freyr*$^+$ calls safeguard for 21.8% of invocations to avoid potential performance degradation.

P99 latency. Figure 9(a) shows the CDF of function response latency of the testing workload. *Freyr*$^+$ has a P99 function response latency in less than 28 seconds, whereas Default RM, Greedy RM and ENSURE are 38, 34, and 36 seconds, respectively. Figure 9(b) shows the CDF of the speedup of the same workload. *Freyr*$^+$ maintains P99

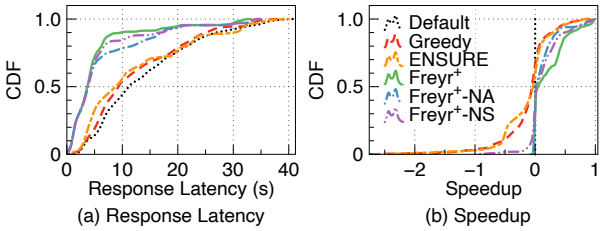


Fig. 9: The CDF of function response latency (left) and speedup (right) in OpenWhisk evaluation, respectively.

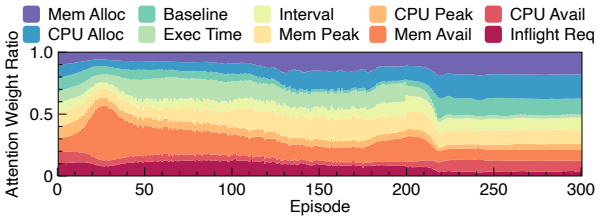


Fig. 10: Relative ratio of attention weights of state features learned by *Freyr*$^+$ through 300-episode training.

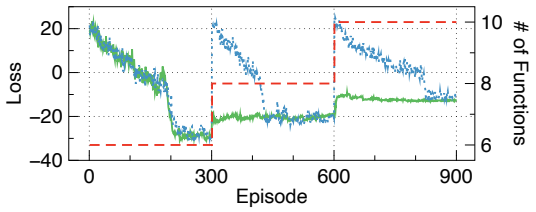


Fig. 11: The learning curve of *Freyr*$^+$ and *Freyr*$^+$-NI with 900-episode training on the OpenWhisk testbed, respectively.

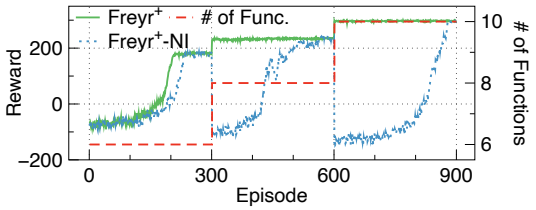


Fig. 12: The cumulative reward of *Freyr*$^+$ and *Freyr*$^+$-NI with 900-episode training on the OpenWhisk testbed, respectively.

speedups below 0.03 for all invocations, whereas Greedy RM and ENSURE are 2.0 and 2.4, respectively. As the Default RM adjusts no resources, the speedup stays 0.

6.4 Ablation Study

We perform an ablation study to examine the effectiveness of two key components, attention and safeguard, in *Freyr*$^+$. We compare *Freyr*$^+$ with two variants, *Freyr*$^+$-NA (No Attention) and *Freyr*$^+$-NS (No Safeguard), by evaluating the OW-test workload from Table 2 on our OpenWhisk testbed.

Response latency. Figure 9(a) shows the CDF of function response latency of three *Freyr*$^+$ variants. *Freyr*$^+$ outperforms the other two variants due to being fully equipped with attention and safeguard mechanism. *Freyr*$^+$ reduces the 99th percentile of the same workload by 15% and 34% compared to *Freyr*$^+$-NA and *Freyr*$^+$-NS, respectively.

Speedup and undeserved performance degradation. Figure 9(b) shows the CDF of execution speedup of three variants. *Freyr*$^+$ outperforms the other two variants by providing faster function invocation executions without significantly degrading the performance of harvested functions. *Freyr*$^+$ and *Freyr*$^+$-NA degrade execution performance 2.4% and 6.3% at worst regarding response latency. Compared to *Freyr*$^+$ and *Freyr*$^+$-NA, *Freyr*$^+$-NS suffers at worst 92% performance degradation.

Harvesting and acceleration. Figure 8 shows the performance of all invocations processed by four variants. *Freyr*$^+$ and *Freyr*$^+$-NS provides more precise harvesting and faster execution acceleration (higher speedups) with attention-enhanced predictions, whereas *Freyr*$^+$-NA accelerates function invocations less (lower speedups). *Freyr*$^+$ and *Freyr*$^+$-NA have some invocations protected by the safeguard daemon, resulting in limited performance degradation. In contrast, invocations handled by *Freyr*$^+$-NS can experience serious performance degradation without safeguarding.

Convergence and cumulative reward. Figure 5(a) shows the learning curve of *Freyr*$^+$ and *Freyr*$^+$-NA through 300 episodes of training, respectively. *Freyr*$^+$ converges faster than *Freyr*$^+$-NA with attention-enhanced observations from environmental interactions. Figure 5(b) presents the cumulative reward obtained by *Freyr*$^+$ and *Freyr*$^+$-NA through 300 episodes of training, respectively. *Freyr*$^+$ achieves maximum cumulative reward faster and higher final reward than *Freyr*$^+$-NA with attention enhancement. Figure 10 depicts the relative ratio of attention weights of ten state features assigned by *Freyr*$^+$. *Freyr*$^+$ gradually learns the unique importance of each feature through 300-episode training.

6.5 Effectiveness of Incremental Learning

To evaluate the effectiveness of *Freyr*$^+$'s incremental learning and *Freyr*$^+$'s robustness to environmental changes, we train *Freyr*$^+$ and a variant of *Freyr*$^+$ separately—*Freyr*$^+$-NI (No Incremental learning)—in a dynamic OpenWhisk environment using the ten functions and the OW-test workload in evaluation. We initially deploy six functions (DH, GD, MS, ALU, EG, and KNN) on the OpenWhisk cluster. After 300 training episodes, we deploy another two functions (VP and IP) to the cluster; after 600 training episodes, we deploy the rest two functions (IR and DV). All functions are configured in the same settings as in OpenWhisk testbed.

Figure 11 and Figure 12 show the learning curve and the cumulative reward of *Freyr*$^+$ and *Freyr*$^+$-NI through 900-episode training in the dynamic environment, respectively. For the first 300 episodes of training, both *Freyr*$^+$ and *Freyr*$^+$-NI converges and achieve the maximal cumulative reward after 250 episodes. At the 301-st episode, two new functions are deployed, *Freyr*$^+$ converges swiftly within a few episodes, while *Freyr*$^+$-NI has to retrain with more than 200 episodes to converge. At the 601-st episode, another two new functions are deployed, *Freyr*$^+$ consistently converge shortly after a few episodes, while *Freyr*$^+$-NI needs to retrain again with 200 episodes for convergence. The results demonstrate that *Freyr*$^+$ can adapt to environmental changes promptly with incremental learning, thus avoiding large retraining overhead. The trained DRL model contains important prior knowledge about function scheduling so that it is not useless when dealing with new functions.

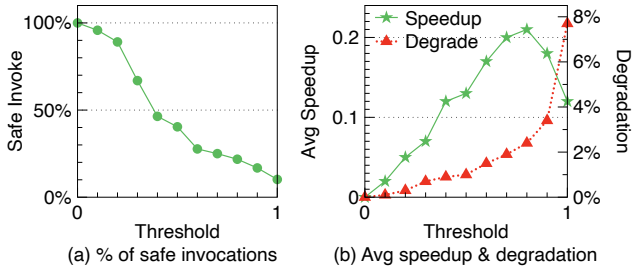


Fig. 13: Sensitivity analysis of safeguard thresholds.

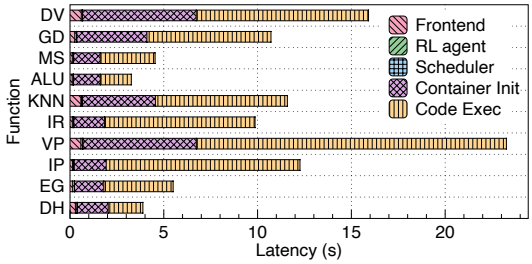


Fig. 14: Latency breakdown.

Figure 11 and Figure 12 show that *Freyr*$^+$ achieves near-optimal performance (over 96% of optimal rewards) when first serving unseen new functions with the old trained model (at episodes 301-st and 601-st) and then quickly convergences within a few episodes to optimal performance by continuously updating the model.

To incrementally update the DRL model, we only need to record the information of function invocations (such as user-requested resources and resource utilization), and the update can be offline. The resources for updating the lightweight DRL model offline only take a CPU core and one GB of memory. To implement incremental learning in a real-time environment, we monitor the frequency of triggering *Freyr*$^+$'s safeguard as an indicator. If the frequency drastically increases above a certain threshold, we deem the environment has new changes and begin the incremental learning process.

6.6 Sensitivity Analysis

We set the default threshold value in the safeguard algorithm to be 0.8, which allows *Freyr*$^+$ to trigger the safeguard just before detecting a full utilization. The threshold is tunable—a high threshold may allow *Freyr*$^+$ to presumptuously harvest idle resource and deteriorate performance, while a low threshold may too conservatively restrict the harvesting and under-utilize resources. We conduct a threshold analysis on our OpenWhisk testbed using the workload OW-test from Table 2 to evaluate the sensitivity of safeguard threshold in *Freyr*$^+$. We increase *Freyr*$^+$'s safeguard threshold from 0 to 1 with a step of 0.1 and run the same workload using *Freyr*$^+$. Figure 13(a) shows the percentage of safe invocations (invocations allocated with user-defined CPU/memory) under each threshold. Figure 13(b) shows the average speedup and percentage of degraded invocations under each threshold. When increasing the threshold, the rate of safe invocation drops down as *Freyr*$^+$ gradually harvests idle resources wildly. The percentage of degraded invocations gradually rises because *Freyr*$^+$'s harvesting policy becomes more and more unrestricted. For average speedup of the workload, *Freyr*$^+$ achieves better and better overall performance until its threshold reaching 0.8. Due to severe performance degradation, *Freyr*$^+$ yields a worse performance for thresholds 0.9 and 1.0.

To deploy *Freyr*$^+$ in a production environment, service providers can tune the safeguard threshold based on their own criteria, *i.e.*, tightening the threshold to conservatively or loosening the threshold to actively harvest idle resources.

6.7 Latency Breakdown

Figure 14 shows the latency breakdowns of ten functions used in evaluation. We run the ten functions in the same setting as on the OpenWhisk testbed. The results are averaged over five times of experiments. *Freyr*$^+$'s RL agent incurs negligible overhead (19 ms on average) compared to the container initialization and function execution time. The RL inference overhead takes only 0.2% of the total function time on average.

7 Related Work

Resource harvesting. Research has been conducted on VM resource management in traditional clouds for years. SmartHarvest [26] proposes a VM resource harvesting algorithm using online learning. MHVM [48] harvests idle memory from VMs and extends the techique to Hadoop. Unlike *Freyr*$^+$, which uses harvested resources to accelerate function executions, SmartHarvest and MHVM offer new low-priority VM services using harvested resources. Directly replacing *Freyr*$^+$ with SmartHarvest or MHVM is not feasible as SmartHarvest is not designed for serverless computing. Zhang *et al.* [25] proposed to harvest VMs for serverless computing, while *Freyr*$^+$ harvests idle resources of serverless functions directly.

Resource provisioning. Spock [49] proposes a serverless-based VM scaling system to improve SLOs and reduce costs. For resource management in serverless, [14] and [15] both aim to automatically adjust CPU resource when detecting performance degradation during function executions, which help mitigate the issue of resource over-provisioning. Unlike [14] and [15] that only focus on CPU, *Freyr*$^+$ manages CPU and memory resources independently. Kaffes *et al.* [16] propose a centralized scheduler for serverless platforms that assigns each CPU core of worker servers to CPU cores of scheduler servers for fine-grained core-to-core management. *Freyr*$^+$ focuses on resource allocation rather than scheduling or scaling. Fifer [10] tackles the resource under-utilization in serverless computing by packing requests to fewer containers for function chains. Instead of improving packing efficiency, *Freyr*$^+$ directly harvests idle resources from under-utilized functions. ServerMore [50] improves server utilization by carefully co-locating serverless and serverful workloads. Our *Freyr*$^+$ is orthogonal to ServerMore and can easily adopt ServerMore as a supplement.

Resource configuration. Recently, many works have been proposed to optimize serverless function costs by user-side resource configuration. COSE [13] uses Bayesian Optimization to find the optimal function configuration that reduces

cost. Sizeless [51] selects the optimal resource size of functions using offline profiling. GRAF [52] and StepConf [53] aim to predict the optimal configuration for serverless function chains. Unlike user-side research, *Freyr*$^+$ is a provider-side framework that dynamically harvests over-provisioned functions and accelerates under-provisioned functions using harvested resources.

Reinforcement learning. SIREN [8] adopts DRL techniques to dynamically invoke functions for distributed machine learning with a serverless architecture. Our work *Freyr*$^+$ leverages DRL to improve the platform itself rather than serverless applications. Decima [29] leverages DRL to schedule DAG jobs for data processing clusters. Metis [54] proposes a scheduler to schedule long-running applications in large container clusters. TVW-RL [30] proposes a DRL-based scheduler for time-varying workloads. George [31] uses DRL to place long-running containers in large computing clusters. Differ from the above works, *Freyr*$^+$ learns resource management in serverless computing using DRL.

8 Conclusion

This paper proposed a new resource manager, *Freyr*$^+$, which harvests idle resources from over-provisioned functions and accelerates under-provisioned functions with supplementary resources. Given realistic serverless workloads, *Freyr*$^+$ improved most function invocations while safely harvesting idle resources using reinforcement learning with attention-enhanced embedding, incremental learning, and safeguard mechanism. We evaluate *Freyr*$^+$ on both large-scale simulation and real-world testbed. Experimental results on the OpenWhisk cluster demonstrate that *Freyr*$^+$ outperforms other baseline RMs. *Freyr*$^+$ harvests idle resources from 38% of function invocations and accelerates 39% of invocations. Compared to the default RM in OpenWhisk, *Freyr*$^+$ reduces the 99th-percentile function response latency by 26% for the same testing workload.

9 Acknowledgements

The work of Hanfei Yu and Hao Wang was supported in part by NSF 2153502, 2403247, 2403398, and the AWS Cloud Credit for Research program. The work of Jian Li was supported in part by NSF 2148309 and 2337914. The work of X. Yuan was supported in part by the NSF grants 2019511, 2348452, and 2315613. The work of Seung-Jong Park was supported in part by NSF 2403248 and 2403399. Results presented in this paper were obtained using CloudBank [55], supported by the NSF award 1925001. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, "Accelerating Serverless Computing by Harvesting Idle Resources," in *Proceedings of the ACM Web Conference (WebConf)*, 2022.

[2] M. Tanna and H. Singh, *Serverless Web Applications with React and Firebase: Develop real-time applications for web and mobile platforms*. Packt Publishing Ltd, 2018.

[3] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," in *Proc. of USENIX NSDI*, 2017.

[4] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proc. of ACM SoCC*, 2018.

[5] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *Proc. of ACM SoCC*, 2017.

[6] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure," in *Proc. of ACM SIGMOD*, 2020.

[7] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: a Serverless Framework for End-to-end ML Workflows," in *Proc. of ACM SoCC*, 2019.

[8] H. Wang, D. Niu, and B. Li, "Distributed Machine Learning with a Serverless Architecture," in *Proc. of IEEE INFOCOM*, 2019.

[9] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," 2019.

[10] J. R. Gunasekaran, P. Thinakaran, N. Chidambaram, M. T. Kandemir, and C. R. Das, "Fifer: Tackling Underutilization in the Serverless Era," in *Proc. of ACM Middleware*, 2020.

[11] A. Fuerst and P. Sharma, "FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching," in *Proc. of ACM ASPLOS*, 2021.

[12] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proc. of USENIX ATC*, 2020.

[13] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "COSE: Configuring Serverless Functions using Statistical Learning," in *Proc. of IEEE INFOCOM*, 2020.

[14] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya, "Automated Fine-Grained CPU Cap Control in Serverless Computing Platform," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[15] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments," in *Proc. of ACSOS*, 2020.

[16] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized Core-Granular Scheduling for Serverless Functions," in *Proc. of ACM SoCC*, 2019.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," 2017.

[19] Apache, "Apache OpenWhisk Official Website," https://openwhisk.apache.org, 2018, [Online; accessed 1-May-2018].

[20] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the Curtains of Serverless Platforms," in *Proc. of USENIX ATC*, 2018.

[21] AWS, "AWS Lambda: Serverless Compute," https://aws.amazon.com/lambda/, 2018, [Online; accessed 1-May-2018].

[22] Google Cloud, "Google Cloud Function:Event-Driven Serverless Compute Platform," https://cloud.google.com/functions, 2018, [Online; accessed 1-May-2018].

[23] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang *et al.*, "PerfIso: Performance Isolation for Commercial Latency-Sensitive Services," in *Proc. of USENIX ATC*, 2018.

[24] P. Ambati, Í. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety *et al.*, "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms," in *Proc. of USENIX OSDI*, 2020.

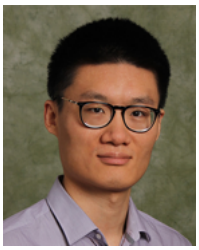
[25] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and Cheaper Serverless Computing on Harvested Resources," in *Proc. of ACM SOSP*, 2021.

[26] Y. Wang, K. Arya, M. Kogias, M. Vanga, A. Bhandari, N. J. Yadwadkar, S. Sen, S. Elnikety, C. Kozyrakis, and R. Bianchini,

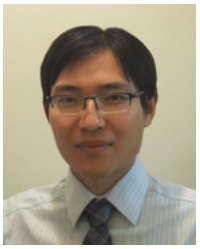
"SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud," in *Proc. of ACM EuroSys*, 2021.
[27] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.
[28] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource Management with Deep Reinforcement Learning," in *Proc. of ACM HotNets*, 2016.
[29] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning Scheduling Algorithms for Data Processing Clusters," in *Proc. of ACM SIGCOMM*, 2019.
[30] S. S. Mondal, N. Sheoran, and S. Mitra, "Scheduling of Time-Varying Workloads Using Reinforcement Learning," in *Proc. of AAAI*, 2021.
[31] S. Li, L. Wang, W. Wang, Y. Yu, and B. Li, "George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints," in *Proc. of ACM SoCC*, 2021.
[32] L. Bruzzone and D. F. Prieto, "An incremental-learning neural network for the classification of remote-sensing images," *Pattern Recognition Letters*, 1999.
[33] R. Polikar, L. Upda, S. S. Upda, and V. Honavar, "Learn++: An incremental learning algorithm for supervised neural networks," *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, 2001.
[34] G. A. Carpenter, S. Grossberg, N. Markuzon, J. H. Reynolds, D. B. Rosen *et al.*, "Fuzzy artmap: A neural network architecture for incremental supervised learning of analog multidimensional maps," *IEEE Transactions on neural networks*, 1992.
[35] DataDog, "The State of Serverless," https://www.datadoghq.com/state-of-serverless-2020/, 2020, [Online; accessed 1-July-2021].
[36] Docker, "Docker: Empowering App Development for Developers," https://www.docker.com, 2021, [Online; accessed 1-May-2021].
[37] L. Ilya and H. Frank, "Decoupled Weight Decay Regularization," in *Proc. of ICLR*, 2019.
[38] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," 2017.
[39] IBM, "IBM Cloud Functions," https://www.ibm.com/cloud/functions, 2021.
[40] PyTorch, "PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration," https://pytorch.org, 2018, [Online; accessed 1-May-2018].
[41] OpenAI, "Gym: A Toolkit for Developing and Comparing Reinforcement Learning Algorithms." https://gym.openai.com, 2018, [Online; accessed 1-May-2018].
[42] AWS, "AWS EC2: Secure and Resizable Compute Capacity in the Cloud," https://aws.amazon.com/ec2/, 2020, [Online; accessed 1-May-2020].
[43] M. Copik *et al.*, "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing," *arXiv preprint arXiv:2012.14132*, 2020.
[44] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in *Proc. of ACM SoCC*, 2020.
[45] A. Krizhevsky, G. Hinton *et al.*, "Learning Multiple Layers of Features from Tiny Images," 2009.
[46] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "Youtube-8m: A Large-scale Video Classification Benchmark," *arXiv preprint arXiv:1609.08675*, 2016.
[47] D. L. Wheeler, T. Barrett, D. A. Benson, S. H. Bryant, K. Canese, V. Chetvernin, D. M. Church, M. DiCuccio, R. Edgar, S. Federhen *et al.*, "Database Resources of the National Center for Biotechnology Information," *Nucleic Acids Research*, 2007.
[48] Fuerst, Alexander and Novaković, Stanko and Goiri, Íñigo and Chaudhry, Gohar Irfan and Sharma, Prateek and Arya, Kapil and Broas, Kevin and Bak, Eugene and Iyigun, Mehmet and Bianchini, Ricardo, "Memory-harvesting vms in cloud platforms," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.
[49] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud," in *Proc. of IEEE CLOUD*, 2019.
[50] A. Suresh and A. Gandhi, "Servermore: Opportunistic execution of serverless functions in the cloud," in *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, 2021.
[51] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proceedings of the 22nd International MIDDLEWARE Conference (MIDDLEWARE)*, 2021.
[52] J. Park, B. Choi, C. Lee, and D. Han, "GRAF: A Graph Neural Network Based Proactive Resource Allocation Framework for SLO-Oriented Microservices," in *Proc. of the 17th International Conference on emerging Networking EXperiments and Technologies (CONEXT)*, 2021.
[53] Z. Wen, Y. Wang, and F. Liu, "StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows," in *Proc. of the IEEE International Conference on Computer Communicationss (INFOCOM)*, 2022.
[54] L. Wang, Q. Weng, W. Wang, C. Chen, and B. Li, "Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale," in *Proc. of ACM SC*, 2020.
[55] M. Norman, V. Kellen, S. Smallen, B. DeMeulle, S. Strande, E. Lazowska, N. Alterman, R. Fatland, S. Stone, A. Tan *et al.*, "CloudBank: Managed Services to Simplify Cloud Access for Computer Science Research and Education," in *Practice and Experience in Advanced Research Computing (PEARC)*, 2021.

Hanfei Yu received his B.E. degree in Electronic Engineering from Shanghai Jiao Tong University, Shanghai, China, in 2019, and his M.S. degree in Computer Science from University of Washington, Tacoma, WA, USA, in 2021. He is currently working toward a Ph.D. degree in Computer Engineering at Stevens Institute of Technology, Hoboken, NJ, USA. His research interests include cloud computing, serverless computing, reinforcement learning, and AI systems.

Hao Wang (M) is an Assistant Professor in the Department Electrical and Computer Engineering at Stevens Institute of Technology, Hoboken, NJ, USA. He received both his B.E. degree in Information Security and M.E. degree in Software Engineering from Shanghai Jiao Tong University, Shanghai, China, in 2012 and 2015 respectively, and the Ph.D. degree in the Department of Electrical and Computer Engineering at the University of Toronto, Canada in 2020. His research interests include distributed ML systems, AI security and forensics, privacy-preserving data analytics, serverless computing, and high-performance computing. He is a recipient of the NSF CRII Award.



Jian Li (M) received the B.E. degree from Shanghai Jiao Tong University, Shanghai, China, in June 2012, and the Ph.D. degree in computer engineering from Texas A&M University at College Station, College Station, TX, USA, in December 2016. He is an Assistant Professor of Data Science with the Departments of Applied Mathematics and Statistics & Computer Science in the College of Engineering and Applied Sciences at Stony Brook University, State University of New York (SUNY), Stony Brook, NY, USA. Before that, he was an Assistant Professor at SUNY-Binghamton from 2019 to 2023. He was a Postdoctoral Fellow with the College of Information and Computer Sciences, University of Massachusetts Amherst, Amherst, MA, USA, from January 2017 to August 2019. His current research interests lie in the areas of reinforcement learning, online learning, network optimization, online algorithms, and their applications in large-scale networked systems.



Xu Yuan (M'16-SM'22) received the B.S. degree from the College of Information Technology, Nankai University, Tianjin, China, in 2009, and the Ph.D. degree from the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA, in 2016. From 2016 to 2017, he was a Post-Doctoral Fellow of Electrical and Computer Engineering with the University of Toronto, Toronto, ON, Canada. He is currently an Associate Professor in the Department of Computer and Information Sciences at the University of Deleware, Newark, DE, USA. Before that, he was a Hardy Edmiston Endowed Assistant Professor in the School of Computing and Informatics at the University of Louisiana at Lafayette. He was the receipt of NSF CRII Award and NSF CAREER Award. His research interest focuses on artificial intelligence, cybersecurity, networking and cyber-physical system.

Seung-Jong Jay Park is the Kummer Endowed Professor and the Chair of the Computer Science Department of the College of Engineering and Computing at the Missouri University of Science & Technology, Rolla, MO, USA. He was the Dr. Fred H. Fenn Memorial Professor of Computer Science and Engineering at Louisiana State University. He has worked in cyberinfrastructure development for large-scale scientific and engineering applications since 2004. He received his Ph.D. in the school of Electrical and Computer Engineering from the Georgia Institute of Technology (2004). He has performed interdisciplinary research projects including (1) big data & deep learning research including developing software frameworks for large-scale science applications and (2) cyberinfrastructure development using cloud computing and high-performance computing. Those projects have been supported by federal and state funding programs from NSF, NASA, NIH, etc. He received IBM faculty research awards between 2015-2017. Since 2021, he has served at the U.S. National Science Foundation (on leave from LSU) as a program director for research support programs, such as Cyberinfrastructure for Sustained Scientific Innovation (CSSI), Principles and Practice of Scalable Systems (PPoSS), Computational and Data-Enabled Science and Engineering (CDS&E), OAC Core, etc.