Towards Optimizing Performance-Resource Trade-Off for Serverless Functions

Zhouruixing Zhu, *222010522,The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen)*

***Abstract*—Serverless computing, a rapidly emerging cloud paradigm, offers significant benefits in terms of resource efficiency and on-demand scalability. However, it faces a critical challenge in the form of cold starts, which introduce latency and degrade the performance of function invocations. Addressing this issue requires a delicate balance between minimizing cold start latency and optimizing resource usage. This survey explores three prominent approaches that aim to mitigate cold starts while optimizing the performance-resource trade-off in serverless environments: SPES, Hybrid, and Defuse. SPES proposes a differentiated scheduling strategy that leverages predictable invocation patterns to optimize function provisioning and significantly reduces both cold start latency and wasted memory time. Hybrid focuses on characterizing the FaaS workload and introduces a resource management policy that reduces cold starts while minimizing resource consumption. Defuse, on the other hand, takes a dependency-aware approach by mining function invocation histories to identify dependencies between serverless functions, thereby reducing cold starts through smarter scheduling. By comparing and synthesizing these approaches, we highlight the strengths and limitations of current solutions and provide insights into potential future directions for optimizing performance and resource utilization in serverless systems.**

***Index Terms*—Cloud computing, serverless computing, cold start, function categorization.**

I. Introduction

Function-as-a-service (FaaS), the most prominent implementation pattern of serverless computing, has extensively simplified developers' access to cloud resources. Major cloud vendors such as AWS Lambda, Google Cloud Functions, and Azure Functions have supported FaaS-based web services [1], [2], machine learning [3], [4], and other cloud applications [5], [6]. FaaS shifts the burden of infrastructure management from developers to cloud vendors, allowing developers to focus on their application functionality [7]. The statelessness and event-driven architecture of FaaS facilitate seamless updates and flexible application deployment. Plus, FaaS applies a pay-as-you-go billing model and dynamically allocates resources based on demand [8], appealing to users via cost-saving benefits. Figure 1 depicts a serverless application example.

Despite their benefits, cloud users on serverless platforms suffer from the infamous cold-start problem, which causes them to trade off performance latency and memory costs. Serverless functions are spawned on instances and typically have very short execution duration (orders of milliseconds to seconds) [9]. However, booting a function from scratch (*i.e., cold start*) incurs expensive latency in preparing the execution environment [10], compared to running an *warm* function whose instance is already loaded in the memory. Cold-start latency can account for 80% of the total response

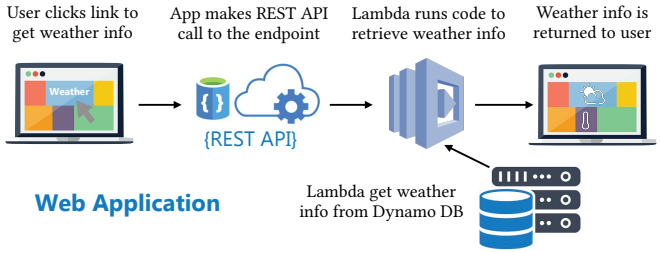


Fig. 1. An weather inquiry website based on a serverless web application.

latency [11], leading to user dissatisfaction and eventually causing user churn and economic loss. The pay-per-use model of FaaS also motivates cloud vendors to avoid unbillable set-up latency [12]. Meanwhile, keeping all functions warm is infeasible due to two reasons. First, most functions are invoked infrequently [13]. It is unacceptable to keep idle (also unbillable) and infrequently used function instances in memory, as it leads to unnecessary costs and wastes valuable resources. Second, because of the infrequent invocation and short execution, FaaS providers colocate thousands of function instances on a single server to achieve full utilization [14]. Loading all functions can occupy hundreds of GBs of memory, far beyond a server's capacity [14]. Thus, reducing cold start latency while minimizing memory waste has been a key challenge [15], [3], [16].

Existing solutions generally fall into two categories [17], [18]: accelerating the setup [19], [20], [21] or reducing cold start occurrences [22], [13], [23]. This paper focuses on the reduction of runtime cold start occurrence without renovating the underlying system-layer infrastructure. Previous such efforts [22], [13], [23] either keep instances loaded for a fixed period post-execution or manage instances based on rudimentary inter-invocation intervals. However, they overlook the intrinsic influence of triggering events on invocation patterns, thus falling short in addressing infrequently invoked functions. Hence, we intend to develop a more effective, non-intrusive method that can load functions right before incoming invocations and recycle idle instances with no potential near-future invocations. The key is to predict invocations accurately.

We identify four challenges in developing such a method:

- *Efficiency*: A FaaS platform can receive thousands of function invocations every minute from millions of potential functions [24]. Most functions have a very short execution time, so the method must decide the provision within a limited and unbillable time.
- *Scalability*: The method should be highly elastic, quickly scaling up and down to meet invocation fluctuations promptly, as the requests of FaaS functions are usually bursty and dynamic [2].

- *Imbalance*: The distribution of function invocations can be highly imbalanced, and most functions are rarely invoked [13]. This poses a challenge to approaches requiring sufficient training data.
- *Evolution*: The lightweight nature of FaaS makes applications evolve faster than traditional software. Both short-term factors (e.g., user activity variations) and long-term factors (e.g., software updates) can cause concept shifts in function invocations, thereby hindering predictive models.

In this article, we provide a comprehensive survey of existing techniques for optimizing cold starts in serverless functions. We present a detailed examination of three state-of-the-art methods: Hybrid [13], Defuse [22], and SPES [25], along with an analysis of their experimental results. We believe that this survey will offer valuable insights for both practitioners and researchers working on serverless function optimization.

The article is structured as follows: Sections 2 and 3 review related work and provide the necessary background, respectively. Sections 4 and 5 focus on surveying current approaches for cold start optimization in serverless functions and evaluating their performance. Finally, Section 6 concludes the article by highlighting potential research directions based on the findings of our survey.

II. RELATED WORK

Numerous efforts [26], [27], [28], [13], [22], [29], [19], [15], [30] have focused on minimizing cold starts and memory overhead for FaaS platforms. Related studies [17], [21] classify current methods into two categories: speeding up function warming (involving renovations in the system layer) and reducing occurrences of runtime cold starts (at the application layer). Methods of these two categories are orthogonal, so they can be combined for further cold start optimization.

This first category presents new systems to optimize infrastructure and resource management, employing techniques like load balancing among worker nodes [26], [27], snapshotting [31], [14], sandbox (e.g., container) scheduling and management [23], [28], [19], [32], [33], [34], [35]. One of the most popular research directions is sandbox sharing, i.e., sandboxes of the previous execution are reused for new invocations since reusing the idle sandbox incurs less delay than allocating a new one [32]. A recent study, Pagurus [33], proposed a container management scheme that allows one function's idle warm container to be forked by another function to alleviate cold starts, showing promising results. However, implementing these methods requires significant engineering efforts, system expertise, and ongoing maintenance due to the underlying platform and sandbox modifications, which pose challenges in diverse infrastructures and platforms.

The second category mitigates cold-start occurrences [13], [22], [15], [29], [23], [30] through a non-intrusive way. A primary step of function scheduling is to decide when and whether to pre-load/evict a function instance to reduce cold starts. [13] utilizes a small histogram to monitor function inter-invocation times, benefiting workloads with clear invocation patterns by optimizing keep-alive and pre-warming. However, it is fully data-driven and ignores the underlying invocation patterns, leaving much room for domain-knowledge-involved invocation prediction. Defuse [22] employs dependency mining from function invocation histories to optimize the keep-alive time and pre-warming. Yet, it relies on the statistical histogram and turns to a fixed keep-alive policy for more than 32% of the functions, delivering inadequate perfection of effectiveness. LCS [30] selected the least recently warm container to reduce cold starts by keeping the containers alive for a longer period. FaaSCache [15] innovatively establishes an equivalence between keeping functions alive and keeping objects in a cache, thereby implementing function provision based on Greedy-Dual-Size Frequency object caching. However, its fundamental idea is to use up the given resources as much as possible until they are insufficient, then evict containers to minimize cold starts. Thus, its way of optimizing resources is rigid without any predictions, failing to smoothly optimize cold starts and resource usage simultaneously. This leads to a more significant waste of memory resources.

There is improvement space for these function provisioning methods as they lack an understanding of invocation patterns and fine-grained invocation prediction. Instead, SPES adopts horses-for-courses rule-based strategies to enhance existing approaches via accurate invocation prediction. SPES also proposes adaptive designs and builds up inter-function correlations, delivering better memory-economic cold-start reduction. Other efforts target a downstream task to determine a suitable node for scheduling an individual function request. FaaSRank [29] and ENSURE [23] attempt to pack load on the adequate number of invokers, allowing the additional unneeded invokers to idle, so as to reduce the completion time and cold starts of functions. Our method can be combined with these methods for finer-grained scheduling.

III. BACKGROUND

A. Serverless Computing

Serverless computing represents an emerging cloud programming paradigm where cloud providers fully manage the underlying infrastructure and allocate resources dynamically, enabling developers to concentrate solely on their application's core logic. In FaaS, developers implement services as stateless workloads, *i.e.* functions, designed to respond to an individual event using pre-defined rules, called the *trigger* [36]. Developers only pay for the actual computation resources on a per-execution basis, making it cost-efficient and scalable [37]. Besides, a could service-based *application* is usually broken down into separate, independent functions in practical implementation, where these functions are sometimes explicitly chained together [38]. The adoption of serverless computing has witnessed rapid growth, especially in web services, machine learning training [39], etc. Notwithstanding its advantages, serverless computing poses new challenges, such as ensuring that a service adheres to quality of service (QoS) demands related to response time or tail latency [40].

B. Cold Start Challenge

Figure 2 displays a serverless function's lifecycle. A FaaS provider first downloads the code of users and initiates the

execution environment (or sandbox) in the memory on cluster machines, known as a *cold start*. A cold start involves retrieving code from storage, container (or VM) initialization, loading code into memory, and executing the function's handler. In contrast, a *warm* start jumps directly to execution, resulting in a faster response. Upon serving a request, its environment remains idle for a while without other invocations before the orchestration system decommissions it [41]. Naturally, the more reused environments, the less cold starts. There exists a trade-off in keeping the instance alive: saving start-up resources and speeding up subsequent requests but incurring *idle time* costs. Such costs can be quantified by *wasted memory time (WMT)*, i.e., the time when the image of a function is kept in memory, but the function is not actually invoked. WMT is an important metric to measure resource waste in practice [13].

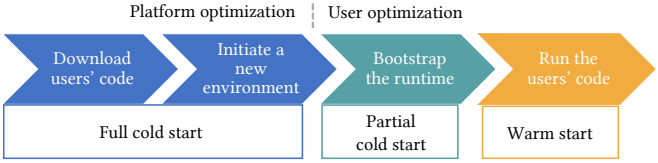


Fig. 2. A serverless function's lifecycle.

The cold start problem is critical because it significantly increases runtime latency, yet serverless providers should meet the QoS requirements and maintain an "always-ready" illusion to users. On the one hand, cold starts can dominate the overall execution time [21], [15], [42]. [19] shows that the "Execution/Overall" latency ratio of most tested functions in gVisor can not even achieve 30%. On the other hand, the cold start happens very frequently [43], [18], [44]. Hence, the additional latency incurred by cold starts is exceptionally unbearable. Considering that memory is limited and expensive, mitigating the cold start challenge aims to either load/unload proper function instances or speed up cold start initiation.

IV. Advanced Methods

In this section, we introduce three advanced methods for cold start optimization in serverless functions: Hybrid, Defuse, and SPES.

A. Hybrid: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider

We use insights from our characterization to design an adaptive resource management policy, called *hybrid histogram policy*. The goal is to reduce the number of cold start invocations with minimum resource waste. We refer to a *policy* as a set of rules that govern two parameters *for each application*:

— *Pre-warming window.* The time the policy waits, since the last execution, before it loads the application image expecting the next invocation. A pre-warming window = 0 means that the policy does not unload the application after one of its functions executes. Aggressive pre-warming (a large window) reduces resource usage but may also cause cold starts, in case the next invocation occurs sooner than expected.

— *Keep-alive window.* The time during which an application's image is kept alive after (1) it has been loaded to memory (pre-warming window $\geq$ 0) or (2) a function execution (pre-warming window = 0). (Note that our definition for this window differs from the keep-alive parameter in fixed keep-alive policies, which is the same for all applications and only starts at the end of function executions.) Longer windows have the potential to reduce cold starts by increasing the chances of an invocation falling into this window. However, this may also waste resources, *i.e.* leave them idle, in case the next invocation does not happen soon after loading.

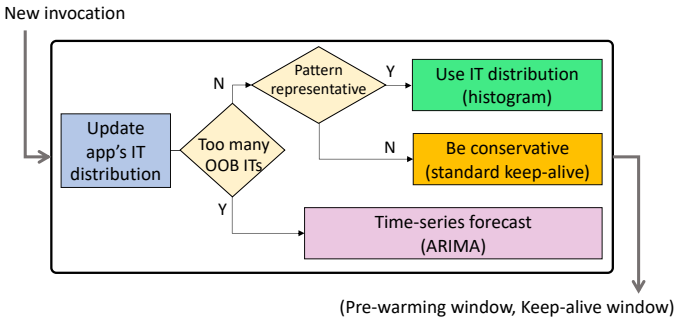


Fig. 3. Overview of the hybrid histogram policy.

A *no-unloading policy* would keep every application image loaded in memory all the time (*i.e.*, infinite keep-alive window and pre-warming window = 0). This policy would get no cold starts but would be too expensive to operate.

1) Design Challenges: Designing a *practical policy* poses several challenges:

- *Hard-to-predict invocations*: Many applications are triggered by timers. A timer-aware policy could leverage this information to pre-warm applications right before the next invocation. However, predicting the next invocation is challenging for other triggers.
- *Heterogeneous applications*: The invocation frequency and pattern vary substantially across applications. A one-size-fits-all fixed policy is certain to be a poor choice for many applications. A better policy should adapt to each application dynamically.
- *Applications with infrequent invocations*: Some applications are invoked very infrequently, so an adaptive policy would take some time to learn their invocation patterns. The same applies to applications that it sees for the first time.
- *Tracking overhead*: Adapting the policy to each application means tracking each application individually. For this reason, the cost to track the information for each application should be small. For example, we need to consider the size of the data structures that will keep this state.
- *Execution overhead*: Since function executions can be very short (*i.e.*, more than 50% of executions take less than 1 second), running the policy and updating its state need to be fast. This is especially critical considering providers charge users only during their function execution times (*e.g.*, based on CPU, memory). For instance, we cannot take 100 ms to update a policy for each 10 ms-long execution. Due to these overheads, expensive prediction techniques, such as time-series analysis, cannot be used for all applications.

2) Hybrid Histogram Policy:

a) Overview: Our hybrid histogram policy addresses all the above challenges. To address challenges #1 and #2, our policy adjusts to the invocation frequencies and patterns of

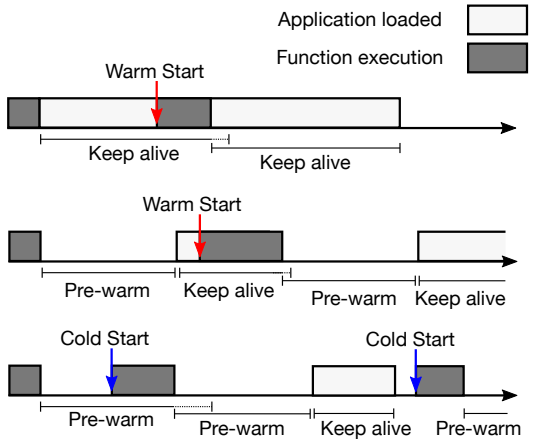


Fig. 4. Timelines showing a warm start with keep alives and no pre-warming (top); a warm start following a pre-warm (middle); and two cold starts, before a pre-warm, and after a keep alive (bottom).

each individual application. It identifies the application's invocation pattern, removes/unloads the application right after each function execution ends, reloads/pre-warms the application right before a potential next invocation (after a "pre-warming window" elapses), and keeps it alive for a period (until a "keep-alive window" elapses). The pre-warming window starts after each function execution, and the keep-alive window starts after each pre-warming. If the pre-warming window is 0, we do not unload the application after an execution, and the end of the execution still starts a new keep-alive window. We explain how exactly we compute the length of these windows below.

Figure 4 shows the pre-warming and keep-alive windows in three scenarios. In the top scenario, the pre-warming window is 0, and an invocation that happens before the keep-alive window ends is a warm start. The end of the execution starts a new keep-alive window. In the middle, the next invocation is a warm start, as the application is re-loaded after a pre-warming window. The end of the execution starts a new pre-warming window. In the bottom scenario, there are two cold starts: the first resulting from an invocation arriving before the pre-warming window elapsed, and the second from an invocation arriving after the keep-alive period elapsed.

The policy comprises three main components: (1) a range-limited histogram for capturing each application's "idle" times (ITs); (2) a standard keep-alive approach for when the histogram is not representative, *i.e.* there are too few ITs or the IT behavior is changing (again, note that this differs from a fixed keep-alive policy); and (3) a time-series forecast component for when the histogram does not capture most ITs. Figure 3 overviews our policy and its components. Ultimately, the policy defines the pre-warming and keep-alive windows for each application. Next, we describe each component in turn.

b) Range-limited histogram: To address challenges #4 and #5, the centerpiece of our policy is a compact histogram data structure that tracks the IT distribution for each application. Each entry/bin of the histogram counts the number of ITs of the corresponding length that have occurred. We use 1-minute bins, which strikes a good balance between metadata size and the resolution needed for policy actions. Keep-alive

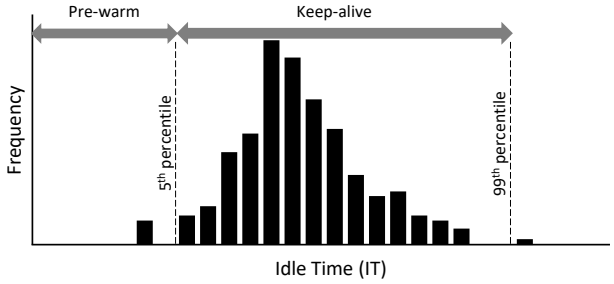


Fig. 5. Example application idle time (IT) distribution used to select pre-warming times and keep-alive windows.

time scales are in orders of minutes for FaaS platforms. We use the same scale for pre-warming. In addition, the histogram tracks ITs of up to a configurable duration (*e.g.*, 4 hours). Any ITs longer than this are considered "out of bounds" (OOBs).

Given the ITs that are within bounds, our policy identifies the head and tail of the IT distribution. We use the head to select the pre-warming window for the application, and the tail to select the keep-alive window. To exclude outliers, we set the head and tail by default to the 5th- and 99th-percentiles of the IT distribution. (When one of these percentiles falls within a bin, we "round" it to the next lower value for the head or the next higher value for the tail.) These two configurable thresholds strike a balance between managing cold starts and resource costs. Figure 5 shows the histogram for a sample application, and the head and tail markers. To give the policy a little room for error, our implementation uses a 10% "margin" by default, *i.e.* it reduces the pre-warming window by 10% and increases the keep-alive window by 10%.

Figure 6 shows nine real IT distributions over a week. The three histograms in the left column show cases where both head and tail cutoffs are easy to identify. These distributions produce the ideal situation: long pre-warm windows and short keep-alive windows. The center cases show no head cutoff as the head marker rounded down to 0. In these cases, the pre-warming window is 0 and the policy does not kill the application after a function execution.

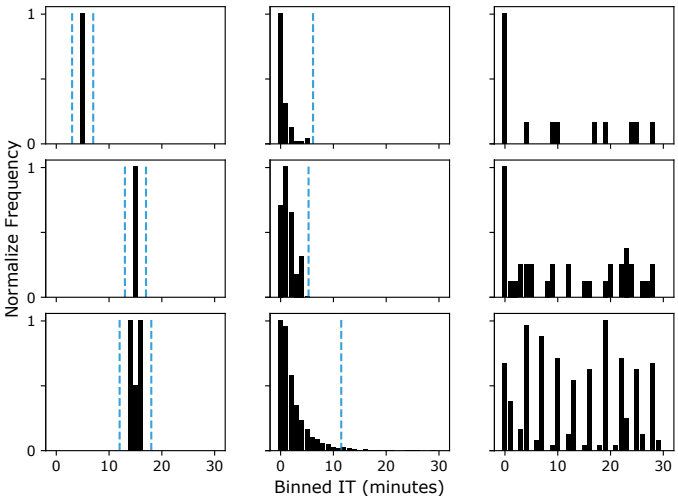


Fig. 6. Nine normalized IT distributions from real FaaS workloads over a week.

c) Standard keep-alive when the pattern is uncertain: The histogram might not be representative of an application's behavior when (1) it has not observed enough ITs for the application, or (2) when the application is transitioning to a different IT regime (*e.g.*, change from a consistent pattern to an entirely new one). When the histogram is not representative, we revert to a standard keep-alive approach: pre-warming window = 0 and keep-alive window = range of the histogram (*e.g.*, 4 hours). This conservative choice of keep-alive window seeks to minimize the number of cold starts while the histogram is learning a new pattern. Our policy reverts back to using the histogram when it becomes representative (again).

We decide whether a histogram is representative by computing the CV of its bin counts. A histogram that has a single bin with a high count and all others 0 would have a high CV, whereas a histogram where all bins have the same value would have CV = 0. The histogram is most effective in the former case, where there is a large concentration of ITs (left and center of Figure 6). It is not as effective when ITs are spread widely (right of Figure 6). Thus, if the CV is lower than a threshold, we use the standard keep-alive approach. To track the CV efficiently, we use Welford's online algorithm [45].

d) Time-series analysis when histogram is not large enough: A compact histogram cannot represent ITs larger than its range. Thus, applications with very infrequent invocations (challenge #3) may exhibit many out-of-bounds ITs. For these applications, our policy uses time-series analysis to predict the next IT. Specifically, we use ARIMA modeling [46].

With an IT prediction, our policy sets the pre-warm window to elapse just before the next invocation and a short keep-alive window. In more detail, we used the `auto_arima` implementation from the `pmdarima` package [47], which automatically searches for the ARIMA parameters (p, d, q) that produce the best fit. As applications using ARIMA are invoked very infrequently, we update the model for each of them after every invocation. To give the prediction some room more inaccuracy, we include a (configurable) margin of 15%. For example, if the predicted IT is 5 hours, we set the pre-warming window to 4.25 hours (5 hours minus 15%) and the keep-alive window to 1.5 hours (15% of 5 hours in each side of the IT prediction).

e) Justification: Like other FaaS cold start policies, our policy eagerly frees up memory when it is not needed. An alternative would have been to leverage standard (lazy) caching policies, which free up cache space only on-demand. Section II explains the differences between these types of policies that justify our approach. Our policy uses a standard keep-alive with a long window, when it does not have accurate IT data about the application, to conservatively prevent cold starts. A shorter window would lower cost but would incur more cold starts. We prefer our approach because it often quickly reduces memory usage greatly, after the histogram becomes active for the application. Instead of using a histogram, we could attempt to predict the next invocation or idle time using time-series analysis or other prediction models. We experimented with some models, including ARIMA, but found them to be inaccurate or excessively expensive for the bulk of invocations. The histogram is accurate, compact, and fast to update. So, we rely on ARIMA only for the applications that cannot be represented with a compact histogram. Producing an ARIMA model is expensive, but can be off the critical path. Moreover, these applications involve only a small percentage of invocations, so computation needs are kept small. Nevertheless, we can easily replace ARIMA with another model.

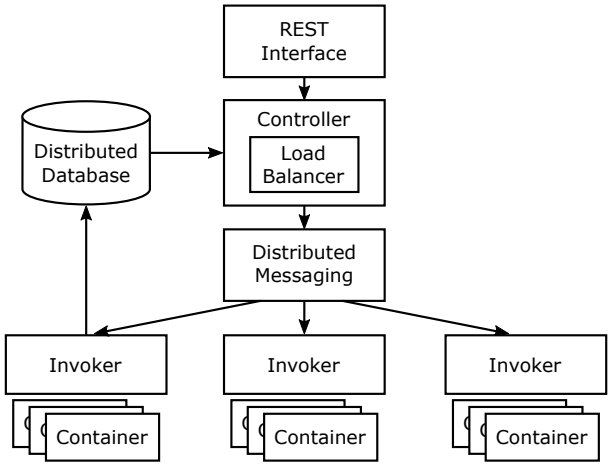


Fig. 7. OpenWhisk architecture.

3) Implementation in Apache OpenWhisk: We implement our policy in Apache OpenWhisk [48], which is the open-source FaaS platform that powers IBM's Cloud Functions [49]. It is written in Scala.

a) OpenWhisk architecture: Figure 7 shows the architecture of OpenWhisk [48]. It exposes a REST interface (implemented using Nginx) for users to interact with the FaaS platform. A user can create new functions (*actions* in OpenWhisk terminology), submit new invocations (*activations* in OpenWhisk terminology), or query their status. Here, we focus on function invocation and container management. Invocation requests are forwarded to the Controller component, who decides which Invoker should execute each function instance. This logic is implemented in the Load Balancer, which considers the health and available capacity of the Invokers, as well as the history of prior executions. The Controller sends the function invocation request to the selected Invoker via the distributed messaging component (implemented using Kafka). The Invoker receives the invocation request, starts the function in a Docker container, and manages its runtime (including when to stop the container). By default, each Invoker implements a fixed 10-minute keep-alive policy, and informs the Controller when it unloads a container.

b) Implementing our policy: We modify the following OpenWhisk components to implement the hybrid policy:

1) **Controller:** Since all invocations pass through the Load Balancer, it is the ideal place to manage histograms and other metadata required for the hybrid policy. We add new logic to the Load Balancer to implement the hybrid policy and to update the keep-alive and pre-warm parameters after each invocation. We also modify the Load Balancer to publish the pre-warming messages.

2) **API:** We send the latest keep-alive parameter for a function to the corresponding Invoker alongside the invocation request. To do this, we add a field to the *ActivationMessage* API,

specifying the keep-alive duration in minutes.

3) **Invoker:** The Invoker unloads Docker containers that have timed-out in the *ContainerProxy* module. We modify this module to unload containers based on the keep-alive parameter received from *ActivationMessage*.

B. Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms

This section introduces the design of Defuse. We will first demonstrate the workflow of Defuse. Then we will present each step in detail, i.e., dependency mining, dependency set generation, and scheduling.

1) Overview: There are three steps of Defuse, dependency mining, dependency set generation, and scheduling. First, Defuse takes the invocation histories of serverless functions as the input and conducts dependency mining on these invocation records. The dependencies are divided into two categories, i.e. strong dependencies and weak dependencies. A function dependency graph is constructed based on the mined dependencies. Then we generate dependency sets of serverless functions according to the graph. All these dependency sets are the input of the dependencyguided scheduling policy, in which all serverless functions in a dependency set are scheduled as a whole. FaaS platforms can decide when to load a dependency set and how long to keep it in the memory based on the scheduling policy.

2) Dependency Mining: The first challenge to solve is how to discover the dependencies among serverless functions. We will start by defining the dependency we want to find and the intuition behind it. Then we will present how to acquire the dependencies from the invocation histories of serverless functions and how to generate dependency sets.

a) Definition of the Dependency: Two aspects need to be considered when defining the dependencies among serverless functions. The first is how to precisely describe the dependencies demonstrated in Section III. There are two properties of the dependencies. First, dependent functions are likely to be invoked together. Second, dependencies only exist among serverless functions of the same client because the dependencies result from the usage pattern of the clients. Typically, clients only have access to their own serverless functions. It would be meaningless to define dependencies across clients.

The second aspect is the cold starts incurred by the unpredictable functions. If these ubiquitous unpredictable functions cannot be properly dealt with, the latency of FaaS platforms will degenerate greatly. The dependencies between predictable and unpredictable functions could be leveraged to solve the problem. Here is a motivating example. In serverless-trainticket, users may book tickets at any time. This will lead to the unpredictable invocation of the function `preserve-ticket`. During the execution of `preserve-ticket`, it will invoke `dispatch-seat`, which is a common service that is invoked frequently and is predictable. The above dependencies help us relate unpredictable functions with predictable ones. The unpredictable functions could be scheduled according to the invocation patterns of predictable ones, which will reduce the cold starts.

Therefore, we define the strong dependencies and weak dependencies among serverless functions.

- *Strong Dependency*: Function fa and function fb have strong dependency iff. 1) they belong to the same client and 2) there is high probability of them being simultaneously invoked in a small time window. It is a bidirectional relationship ($f_a \leftrightarrow f_b$).
- *Weak Dependency*: Function fa have weak dependency on function f_b iff. 1) they belong to the same client and 2) there is high probability that f_a is invoked under the condition that fb is invoked. It is a single-directional relationship ($f_a \longrightarrow f_b$).

Both strong and weak dependencies should satisfy two conditions, i.e., (1) the ownership condition, and (2) the probability condition. The strong dependencies describe the relationship between globally frequently invoked functions, which are likely to be predictable. The weak dependencies describe the relationship between unpredictable and predictable functions.

b) Strong Dependency Mining: : The purpose of strong dependency mining is to find the relationships among functions that are frequent and predictable. We need to find combinations of a client's functions that have a high probability of being invoked together. Frequent pattern mining [50] naturally fits this requirement. Given a set of transactions, frequent pattern mining can find all the itemsets with frequency greater than a given threshold. Hence, Defuse adopts frequent pattern mining to uncover the strong dependencies among serverless functions.

Specifically, for each client, the invocation records of all her functions can be represented as a set $R = r_i | i = 1, 2, \ldots, m$, where m is the number of functions of the client, $r_i = (t_1, t_2, \ldots, t_n)$ is the invocation records of function f_i, and t_j is the timestamp of its jth invocation. Defuse first divides the period where the records are sampled into small non-overlapping time windows and counts the number of invocations of each function in the time window. Then we get the invocation matrix I for the client, where $I_{i,j}$ is the number of invocations of function i in time window j. After that, for each column of I, Defuse gathers all the functions with non-zero invocation count into a single transaction and gets all the transactions of the client. Finally, Defuse employs FP-Growth [51] to conduct frequent pattern mining on the generated transactions. The outputs of frequent pattern mining are frequent itemsets. All functions in a frequent itemset have a high probability of being invoked in the given small time window, which satisfies the probability condition. Additionally, since Defuse conducts frequent pattern mining only on functions that belong to the same client, the ownership condition is satisfied as well. The strong dependencies of all serverless functions in FaaS platforms can be retrieved by repeating the above steps to each client on the platform.

c) Weak Dependency Mining: The goal of weak dependency mining is to find the dependencies between unpredictable and predictable functions. Since the coefficient of variations (CV) of idle time (IT) histograms of unpredictable functions are small. Defuse distinguishes unpredictable functions with predictable ones with the CV of function IT

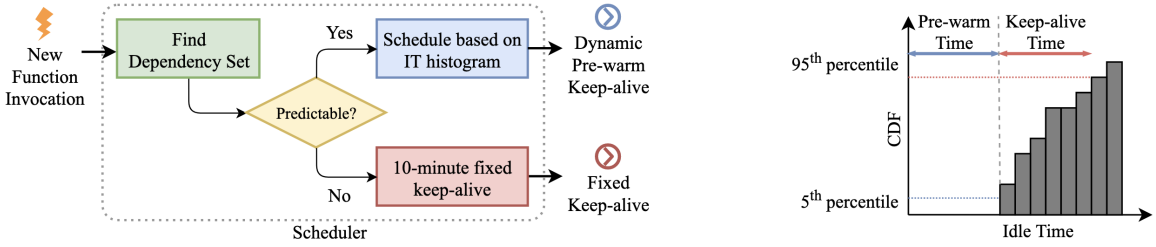


Fig. 8. The Scheduling Policy. (a) The scheduling procedure when a new invocation is triggered. (b) Deciding keep-alive time and pre-warm time by the CDF of idle time between function invocations.

histograms. Particularly, Defuse generates the IT histogram of each function from its corresponding vector in the invocation matrix I. Then Defuse calculates the CV for each function and discriminates them by a threshold.

Defuse mines weak dependencies by positive point-wise mutual information (PPMI) [52]. Suppose the possibilities of an unpredictable function f_u and a predictable function f_p being invoked individually are P_u and P_p, respectively. Let $P_{u,p}$ indicate the probability of them being invoked together. The PPMI of the invocation of f_u and f_p can be represented as:

$$PPMI\left(f_u, f_p\right) = \max\left(0, PMI\left(f_u, f_p\right)\right) \tag{1}$$

Where $PMI\left(f_u, f_p\right)$ is the point-wise mutual information (PMI) [53] between f_u and f_p. It can be represented as:

$$PMI\left(f_u, f_p\right) = \log_2 \frac{P_{u,p}}{P_u \cdot P_p} \tag{2}$$

Intuitively, if f_u and f_p are dependent, the probability of them being invoked together will be higher than they are each invoked independently. As a result, $P_{u,p}$ will be greater than $P_u \cdot P_p$, which means PMI will be positive. The higher the $PMI\left(f_u, f_p\right)$ the stronger the dependency. Since $PPMI\left(f_u, f_p\right)$ is the maximum of $PMI\left(f_u, f_p\right)$ and 0 , it is also positively related to the degree of dependency.

To get PPMIs, Defuse first constructs a co-occurrence matrix C based on the function invocation matrix of predictable and unpredictable functions. Each row of C represents an unpredictable function and each column of C represents a predictable one. $C_{i,j}$ represents the number of co-invocations of function f_i and f_j in a small time window. Then we estimate the probability by invocation frequencies and calculate $PPMI$ based on C. For each unpredictable function f_{u_i}, a vector $v_{u_i} = (PPMI(f_{u_i}, f_{p_1}), \ldots, PPMI(f_{u_i}, f_{p_w}))$ is generated by Defuse. After sorting each vector in descending order, Defuse assigns the top k predictable functions to be weakly dependent on the unpredictable function f_{u_i}, where k can be defined by users.

3) Dependency Set Generation: The second step of Defuse is to generate dependency sets based on the mined dependencies. As mentioned in Section II a scheduler needs to decide the granularity of scheduling and how long a function should stay in memory. However, the mined dependencies are relationships among serverless functions, which cannot be directly exploited. To facilitate the scheduling step, Defuse conducts dependency set generation to convert the relationships into function sets. First, Defuse constructs a function dependency graph as shown in Figure 9. Each vertex in the graph is a serverless function and each edge represents either strong or weak dependency. The dependency sets are defined as connected components on the graph. Then Defuse uses union-find to extract all these connected components and group them as dependency sets. Implied by the definition of dependencies, functions connected with each other have a high probability of being jointly invoked. Scheduling functions in a dependency set together reduces the occurrences of cold starts.

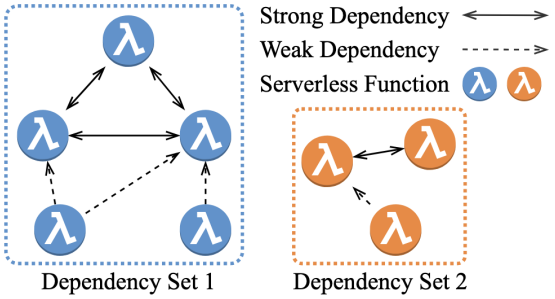


Fig. 9. The Dependency Graph and Dependency Sets of Serverless Functions.

4) Scheduling: The last step of Defuse is to schedule serverless functions based on the generated dependency sets. As discussed in Section II, for each dependency set, the scheduler needs to decide 1) when to pre-warm it by loading it into the memory (pre-warm time) and 2) how long to keep it in memory after it is invoked (keep-alive time). As the IT histogram is proved to be effective in scheduling functions in [13], we adopt the same policy to determine the pre-warm time and keepalive time of each dependency set. Figure 8b illustrates the cumulative distribution function (CDF) of an IT histogram of a dependency set. We set the 5^th percentile of the IT histogram as the pre-warm time and load the set of functions into the memory this period after its invocation. Then we set the time between the 5^th and 95^th percentile as the keep-alive time, which means a dependency set will be reserved in the memory without being invoked for this period.

Besides, there exist some dependency sets without clear invocation patterns. We distinguish dependency sets as predictable sets and unpredictable sets based on their CVs. The scheduling policy is shown in Figure 8a. For predictable sets, Defuse decides the pre-warm time and keep-alive time based on their IT histograms. For unpredictable sets, Defuse steps back to a fixed-timeout policy. Since the granularity of scheduling is finer, Defuse can employ more aggressive

timeout settings and further reduce the negative effect brought about by these unpredictable sets.

C. SPES: Towards Optimizing Performance-Resource Trade-Off for Serverless Functions

This section introduces the design of SPES. An ideal scheduler should decide to load a function exactly before its invocation and evict it from memory after the execution if no more invocations are imminent. The decision-making relies on fine-grained invocation prediction. To this end, we propose SPES, whose overview is presented in Figure 10. SPES consists of four parts: deterministic function categorization, indeterminate function assignment, adaptive strategy application, and function provision based on invocation prediction. Specifically, we first summarize typical invocation patterns, which result from pre-defined triggers [54] and the combinatorial methods of function calling (in common application scenarios [55]) and then formulate their corresponding definitions. If a function satisfies the definitions, it is categorized; Otherwise, the indeterminate functions are assigned to three supplementary types. We also design two extra adaptive strategies to handle the concept shifts as FaaS evolves. Finally, SPES provisions functions according to rule-based invocation prediction, where each type follows its own predicting rule.

Note that different triggers can exhibit the same timing feature in invocations. For example, the service bus trigger and the event gird trigger both can handle moving datagrams (though focusing on different message types). It is neither necessary nor desirable to correspond each trigger to a function type in the context of the cold start problem.

Let us start with three definitions:

- *Waiting time (WT)*: the length of successive idle time. Take an invocation sequence $(28, 0, 12, 1, 0, 0, 0, 7)$ as an example, where each value denotes the invocation count at each sampling slot, derived to a sequence of WTs $\{WT\} = (1, 3)$, as no invocation exists at the 2nd slot or in slots 5–8. If we foreknow the next WT, for example, at the 4th slot, we know no invocations will arrive in slots 5-8, then we can easily make a perfect decision: evict the function now and re-load it at the end of slot 9. Hence, we can transform invocation prediction to WT prediction. Different from a previously proposed inter-arrival time (IAT) [13], WT depicts the slot-grained interval between two successive invocation sequences, from the last's end to the next's start. WT is also different from idle time (IT). IT is the time without invocations or subsequent function executions, so IT exists after any single execution. WT, instead, only appears when a successive series of invocations ends, and the end of a single execution does not necessarily incur a WT.
- *Active time (AT)*: the length of successive time with invocations. The above-mentioned invocation sequence delivers an AT sequence $\{AT\} = (1, 2, 1)$ because the function is invoked in slots 1, 3–4, and 9.
- *Active number (AN)*: a descendable definition, the number of successive invocations during AT. Again, the above sequence delivers $\{AN\} = (28, 13, 7)$.

1) Deterministic Function Categorization: This section defines five invocation types based on typical invocation patterns and categorizes functions accordingly. Table I shows an overview of the five types. The guiding principle for definition is from easy to difficult, also serving as the categorization priority: if a function fits a former type, it will not fit any latter type. Our introduction follows this order.

a) Always warm: This type describes consistently active functions, such as long-running operations and hyperfrequent calls, which usually involve durable or timer-triggered functions (whose interval is small enough). For example, functions in continuous integration/deployment (CI/CD) pipelines are consistently invoked to automate code building, testing, and deployment. A function is defined as "always warm" if 1) it is invoked at every sampling time or 2) the sum of inter-invocation time is $\leq$ one-thousandth the observing time. Such functions are undoubtedly always loaded.

b) Regular: Regular task processing, usually using timer-triggered functions, is a basic functionality (e.g., polling). Yet, their actual invocations may not be strictly periodic (with a constant WT) since there are undesired fluctuations: 1) The exact first/last WTs during an observing period are almost impossible to record. 2) Periodically generated events can be blocked or delayed by contingencies such as concurrency limits, network connectivity issues, etc. 3) Other events can invoke a mostly regularly invoked function occasionally.

Here comes the slacked definition: A "regular" function satisfies either 1) the difference between the 5th and 95th percentile of its WT sequence is ≤ 1 or 2) the coefficient of variation of WTs is close to zero (≤ 0.01 in practice). Otherwise, we remove the first and last WTs and re-check if the function follows the definition. If the answer is still no, we apply another slacking rule by merging adjacent small WTs. In particular, for each WT closely valued to the WT mode, its adjacent small WTs are gradually merged until reaching ① the sequence's end or ② another WT close to the mode or ③ an already merged WT. In this way, a WT sequence valued by $(1439, 1438, 1, 1439, 1438, 1)$ is processed to be $(1439, 1439, 1439, 1439)$, and then it seems "regular". With a processed WT sequence satisfying the above definition, the function belongs to the "regular" type. We partially eliminate accidental factors by processing WTs with slacking rules and modeling essential invocation behaviors. Finally, we record the median of WTs as the *predictive value* for invocation prediction, leveraged in Section IV-C4.

c) Approximatively regular: This is the derived type from "regular". A regular function may experience invocation variance and long-term disturbance. For example, a data-processing function is expected to receive updates from a data station every three minutes (IoT Hub trigger functions). Yet, limited by the data transmission capability, the function is actually invoked every 3–5 minutes. We define such functions as "approximatively (appro-)regular". Particularly, a function is "appro-regular" if the occurring count of the first n frequently appearing values (modes) of WTs is $\geq$ a large percent (90% practically) of the WT sequence's length. n is a pre-defined integer. The predictive values for these "appro-regular" functions are the first n modes.

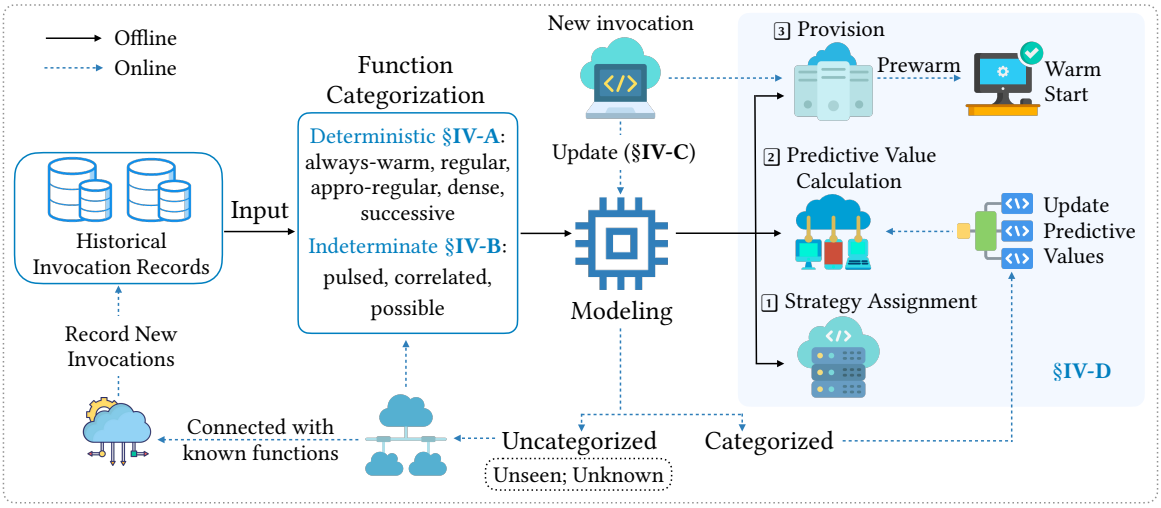


Fig. 10. Overview of SPES. In the offline phase, SPES models the historical invocation records for each function and categorizes functions according to their mined invocation patterns. During the online phase, if the new invocation comes from a categorized function, its function's provision directly follows its provision rule, and uncategorized functions are tried to be connected with known ones. The invocation prediction and predictive values/indicators are then updated upon new invocations.

TABLE I
A BRIEF OVERVIEW OF THE WELL-DEFINED FUNCTION TYPES.

Type name	Characteristics	Definition	Predictive values
Always warm	Almost invoked all the time	Invoked at every time Inter-invocation time $\leq$ observing time‰	–
Regular	Almost invoked periodically	(Processed) $P_{95}(\{WT\}) - P_5(\{WT\}) \leq 1$ CV of $\{WT\} \leq 0.01$	Median of WTs
Appro-regular	Invoked quasi-periodically	(Processed) the count of $\text{Mode}_n(\{WT\}) \geq$ $0.9\times$ sequence length	$\text{Mode}_n(\{WT\})$
Dense	Frequently invoked	$P_{90}(\{WT\}) \leq$ a small constant	$[\min, \max]$ of $\text{Mode}_k(\{WT\})$
Successive	Successively invoked	$\min(\{AT\}) \geq \gamma_1$; $\min(\{AN\}) \geq \gamma_2$ $\gamma_1 < \gamma_2$	–

d) Dense: This characterizes irregular but frequent invocations with intermittent idle durations. "Dense" functions can involve several triggers: queues (or service bus using the queue flavor) for frequent messages in asynchronous processing, influenced by variations in message arrival times; Cosmon DB triggers for frequent real-time data processing; HTTP triggers for processing frequent HTTP requests. Functions with a 90^{th} percentile of the WTs $\leq$ a small constant are defined as "dense." The range of the first k modes derives predictive values, *i.e.* $[\min(\text{Mode}_k(\text{WTs})), \max(\text{Mode})_k(\text{WTs})]$, where k is an empirical integer. Such a function should be unloaded only if its idle time is larger than the constant.

e) Successive: This describes inactive functions experiencing consecutive invocations in a short timeframe (temporal locality) before returning to an inactive state. It can involve HTTP trigger functions (e.g., bursty social media trends), storage trigger functions (e.g., uploading files to Blob Storage), etc. In practice, feedback loops engaged functions, caching mechanism implementations, and load balancing-involved applications can also exhibit such behavior. We define this as $\min(\{AT\}) \geq \gamma_1$ or $\{AN\} \geq \gamma_2$, where $\gamma_1 < \gamma_2$ and they are pre-defined lower bounds. Predicting the start of such an invocation wave is highly difficult. Nevertheless, we can tolerate the first invocation and keep the function alive until the wave ends. With limited cold starts, we save lots of memory.

2) Indeterminate Function Assignment: This section handles functions that do not satisfy the above five definitions using two strategies: ① "forgetting" previous records to recheck whether the not-too-long history invocations satisfy the five definitions, and ② assigning the remaining functions with three extra types.

a) Forgetting: As discussed in §**??**, function invocations experience concept shifts. For example, a regular invoked function can be categorized as "always warm" if the interval is adjusted to be smaller than the recording unit. Therefore, the near-the-present records should be given priority. We slice out the sequential invocation observations in days, from the beginning to the half. Then, we gradually remove the older observations from the original d-day observations and check whether invocations in 2nd–dth days conform to existing definitions. If not, we check the invocations in 3rd–dth days, and so on, until the $\lfloor d/2 \rfloor$th day.

b) Assigning: The remaining indeterminate functions are assigned to three types according to the validation results:

D1. *Pulsed* describes functions showing less obvious temporal locality in invocations than "successive" ones. Similarly, we allow a cold start at the first invocation time and keep the function warm until its ideal time reaches a predefined threshold.

D2. *Correlated* invocations are predicted based on other functions. Functions usually work in a logic workflow topology, interacting with other functions. This is present in several common application patterns, including function chaining (where a sequence of functions executes in a specific order), fan-out (where a function calls multiple functions in parallel), and fan-in (where a function waits for multiple functions to finish) [55]. We define such functions using T-lagged co-occurrence rate (T-COR), derived from COR in §**??**. T-COR is the COR between the target invocation sequence and an T-time-lagged invocation sequence of its candidate functions (those sharing an application/user). The lagged COR can better reflect how the invocation of candidate functions indicates near-future invocations of the target function. If the T-COR reaches a threshold (practically 0.5, $T \leq 10$), the candidate invocation is an important predictive indicator, and these two functions are linked. A function can have multiple linkages with different functions, including categorized and uncategorized ones.

D3. *Possible* functions are infrequently invoked but have at least one mode of WTs (appears more than once). By taking the modes as predictive values, it is possible to predict invocations, though such prediction is deemed to be insufficiently satisfactory.

If any of the three strategies yields both the least cold starts and the minimum wasted memory during validation, the indeterminate function is directly assigned accordingly. Otherwise, the rate of rise is decided. Denote the cold starts of the above three strategies as (cs_1, cs_2, cs_3) and the corresponding wasted memory are wm_1, wm_2, wm_3. Assume definition D1 delivers the minimum cold starts while strategy D2 delivers the minimum wasted memory. We compute the rise rates: $\Delta cs = (cs_2 - cs_1)/cs_1$ and $\Delta wm = (wm_1 - wm_2)/wm_2$. If $\Delta cs \times \alpha \leq \Delta wm$, then the function is assigned to D1, *i.e.* "plused"; otherwise, D2 prevails, where $\alpha \in (0, 1)$ is a scaling factor, and the smaller $\alpha \in (0, 1)$, the more importance is put on cold starts than wasted memory.

Most functions have been categorized so far, except those that have never been invoked in the validation. We simply leave them as "unknown" since they are likely very infrequently invoked and hardly have meaningful predictive indicators. Some unknown functions may be categorized during the online provision based on our adaptive strategies illustrated in the next section (§IV-C3).

3) Adaptive Strategy Application: This section designs two adaptive strategies to handle concept shifts during the online provision: one adjusts the predictive values; the other is based on inter-function correlation. These strategies can further address some unknown or unseen functions (those that never appear in the training data) with meaningful patterns during the online provision. The application of these strategies is also presented in Figure 10, where the invocation prediction and the prediction values are adaptively updated upon online invocations.

a) Adjusting: This adaptive strategy contains three steps:

S1. Record the WTs during the online provision. If there are enough WTs, then initiate the adaptive updating.

S2. Update the predictive values if the corresponding values computed from newly collected WTs change significantly with the mean of the old and new ones.

S3. If the new WTs for an unknown or unseen function conform to previous definitions with enough samples, categorize the function as the corresponding type.

S2 requires more explanations. We record the predictive values for four types: 'regular", "appro-regular", "dense", and "possible". Their predictive values are the median, the first n modes, the range of the first k modes, and the WT values occurring more than twice, respectively. Take "regular" as an example. Suppose the absolute difference between the median of offline WTs (*i.e.* the predictive value) and that of online WTs is larger than the standard of offline WTs. The new predictive value is updated by the mean of the old median and the new median. Other types adopt a similar adjusting strategy.

b) Online correlation: This strategy correlates unseen functions with known functions or appeared unseen functions, as the "correlated" strategy does. To speed up the computation and obtain informative indicators, we only consider candidate functions sharing the same trigger with the target function (*aka.* unseen function). Initially, if one of the candidates is invoked, we also pre-load the target function. Afterward, we gradually remove less correlated candidates. Again, we adopt COR to measure the degree of correlation. We count the pair-wise target-candidate COR at each slot and record the maximum. If the difference between a COR and the maximum is large enough, the corresponding candidate is kicked out from consideration unless its COR returns close to the maximum.

4) Next Invocation Prediction and Provision: This section predicts invocations and pre-loads functions based on the predictive values or indicators. "(Appro-)regular" functions have discrete predictive values and "dense" functions use continuous ones. In terms of "possible" functions, if the range of predictive values is larger than a threshold, the values are regarded as discrete; otherwise, we consider continuous integers inside the range of predictive values. The predicted invocation times are naturally the last invoked time added by each of the predictive values.

For the online provision, if one of the predicted invocation times falls in $[t - \theta_{prewarm}, t + \theta_{prewarm}]$ with a pre-defined parameter $\theta_{prewarm}$ at the time of t, the function will be pre-loaded. If a loaded function's current WT is $\geq \theta_{givenup}$, the function will be evicted from the memory, where the parameter $\theta_{givenup}$ differs among function types. Algorithm 1 shows our provision optimization, which is rule-based with good scalability and implements our defined strategies faithfully. Every time the algorithm returns the updated MemSet, based on which we load function instances.

V. EVALUATION

We evaluate Hybrid, Defuse and SPES by answering four research questions:

- **RQ1**: How effectively does these methods decrease cold starts?
- **RQ2**: How much memory waste and computation overhead does these methods incur?

Algorithm 1: The provision algorithm of SPES.

Input: FList: hash ids for all functions; FState: recording the necessary information about all functions, such as the current WT, history WTs, predictive values, the time of the last invocation, etc; Invo$^{(t)}$: invocation numbers of each function at the time of t; MemSet: hash ids of loaded functions; UCorr: the correlations of unseen functions.

Output: The updated MemSet, FState and UCorr

Function `Provision`(*FList, FState, Invo$^{(t)}$, MemSet, UCorr*)**:**
  for $f \in$ *FList* **do**
    if *Invo$^{(t)}$[f] > 0* **then**
      FState[f].last_invoked $\leftarrow t$;
      FState[f].update_WTs(FState[f].current_WT);
      FState[f].current_WT $\leftarrow 0$;
      // Adaptively adjusting predictive values.
      FState[f].update_predictive_values(t);
      if $f \notin$ *MemSet* **then**
        FState[f].cold_start_record(t);
        MemSet.add(f);
      end
    else
      FState[f].current_WT += 1;
      pre_load_flag $\leftarrow$ FState[f].pre_load(FState[f].$\theta_{prewarm}$);
      if *pre_load_flag is* ***False AND*** *FState[f].current_WT* $\geq$ *FState[f].*$\theta_{givenup}$ **then**
        MemSet.remove(f);
      else if *pre_load_flag is* ***True*** **then**
        MemSet.add(f); // Pre-loading.
    end
    // Adaptively processing unseen functions.
    UCorr.update();
  end
  return *MemSet, FState, UCorr*
End

- **RQ3**: How does these methods trade off memory waste with latency reduction?
- **RQ4**: How do complementary designs influence these methods?

A. Experiment Settings

We evaluate these methods on the most widely used industrial dataset [56] released by Microsoft Azure Function [13] with real-world invocation traces. The dataset contains the invocation counts per minute for 14 days. The first 12 days are used for pattern modeling (training), and the last two days are used for the simulation, conducted on a workstation with an 8-core Intel i5-3470S CPU and 16 GB memory. The records involve 15,097 users, 24,964 applications, and 83,137 functions. 71,616 functions appear in the training data, 39,388 functions appear in the simulation data, and 743 never appeared during training.

We adopt the simulation principles following [13]. First, assume all executions finish within one minute since 1) most (96%) functions have very short execution time (less than 60 seconds), and 2) we can thereby calculate the worst-case wasted resource time. Second, we assume that cold-start latency for each function is the same, so we only need to care about the number of cold starts and the wasted memory time.

1) Baselines: We compare SPES with five state-of-the-art baselines applied to the application layer. Approaches involving system renovations are out of this paper's scope and will be discussed in Section **??**. Our baselines include FaaSCache [15], Defuse [22], Hybrid [13], and a fixed keep-alive policy. We reproduce these methods as they do not provide open-source code by strictly following the original papers and referring to a reproduction attempt [57]. Note that the original Hybrid method works at the application unit (Hybrid-Application, HA), so we derive a Hybrid-Function method by employing its design on the function granularity (Hybrid-Function, HF), following [22]. All the parameters are set according to their original papers. The fixed keep-alive policy adopts a length of 10 mins. FaaSCache requires a pre-defined memory limit, so we adopt the maximum memory size of SPES during the whole simulation.

2) Metrics and parameters: To quality the cold-start optimization, we measure the function-wise (application-wise for HA) *cold-start rate (CSR)*, *i.e.* the number of cold starts divided by the number of invocations. We apply wasted memory time (*WMT*, §III-B) to gauge the idle resource waste. Naturally, the lower the CSR or WMT, the fewer cold starts or wasted resources, the better. We also monitor the *effective memory consumption ratio (EMCR), which measures the fraction of invoked function instances relative to the total loaded instances on each host machine, serving to assess resource efficiency. A higher EMCR signifies wiser memory allocation, as it indicates a greater proportion of memory is used by active instances rather than remaining idle.* As for the parameters, we set $\theta_{prewarm}$ as two. The $\theta_{givenup}$ for "dense" and "plused" is five, whereas one for the other types. Section V-D further discusses the impact of these pre-defined parameters.

B. RQ1: Effectiveness in Cold-Start Reduction

Figrue 11 displays the cumulative distribution (CDF) of CSR under the provision decisions of SPES and baselines. With a fixed y-axis value, the line representing SPES is positioned to the left of other baseline lines, which indicates that SPES consistently leads to fewer cold starts across a variety of functions, each corresponding to different invocation frequencies. In particular, SPES reduces the 75th percentile cold-start rate (Q3-CSR, for simplicity) from 0.215 to 0.108 compared to Defuse, the best-performing baseline, achieving 49.77% improvement, and reduces 75-CSR by 64.06%–89.20% compared to other baselines. We care more about Q3-CSR because infrequently invoked functions benefit most from optimization [13]. Moreover, 57.99% functions experience no cold starts with SPES, indicating that SPES can allow most functions to be warmly invoked. In contrast, 25.61%–52.59% functions completely experience warm startup using baselines where FaaSCache is the best one. On the other hand, regarding infrequently invoked functions, SPES reduces the 90th percentile CSR by 19.87% compared to the best-performing baseline, HA. Such improvements imply that SPES exhibits significant optimization performance for both frequently and

infrequently invoked functions, while no baseline can achieve second-best performance with different percentile CSRs.

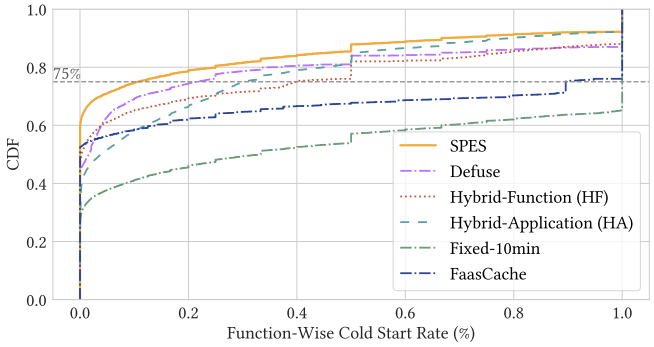
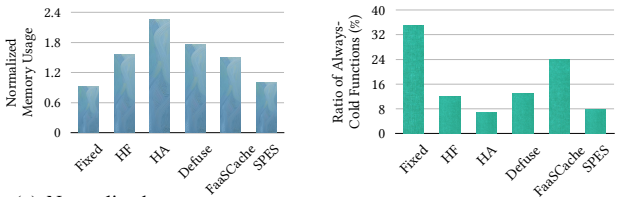


Fig. 11. Cold start behaviors of SPES and its competitors.

Furthermore, SPES significantly reduces "always-cold" functions, which always experience cold starts upon invocations (with CSR equal to 1.0). Figure 12(b) further presents the always-cold percentage, and that of SPES is just less than 8%. Among the baselines, HA has the fewest always-cold functions, which is closest to that of SPES, whereas Defuse and HF, both function-grained methods, increase always-cold functions dramatically. This can be attributed to infrequently invoked functions. About 3.82% functions are invoked less than twice during training, and 6.14% are only once during the simulation, so always-cold functions seem inevitable with limited records. HA mitigates this issue by grouping and loading functions together, yet resulting in more memory consumption and waste. SPES instead connects unseen and unpredictable functions with known functions using trigger, application, and user information and gradually updates the associated functions based on actual invocations. Hence, without much extra memory, SPES performs close to HA.

In addition to cold start reduction, Figure 12(a) presents the memory usage normalized by the averaged one of SPES. SPES's memory usage is only 8.08% more than the most resource-efficient method, the fixed keep-alive policy, and saves 36.07%–55.55% memory compared to the other baselines on average. It only consumes about half of the memory of Defuse, the best baseline for cold start reduction.



(a) Normalized memory usage (b) Percentage of always-cold functions

Fig. 12. SPES is memory-efficient compared to most baselines with relatively few always-cold functions.

Figure 13 shows the averaged CSR among different types. As the simulation period is not long enough, we only categorize unknown functions into the type of "possible", denoted by "newly-possible". We can see that "unknown" functions contribute the most to cold starts, and "pulsed" functions also incur high CSRs. This is attributed to insufficient historical invocation records. Actually, SPES intentionally connives a cold start when an "unknown" or "pulsed" invocation arrives after a long idle time. Though we can leverage less predictive information, such as the averaged WT, to predict the next invocation or even keep such functions always warm, this can lead to considerable unbillable and wasted memory, undesired for FaaS providers. This outcome is deliberating on the trade-off between performance and resource allocation. Such an issue can be mitigated with more invocation histories revealing predictive indicators for better provision.

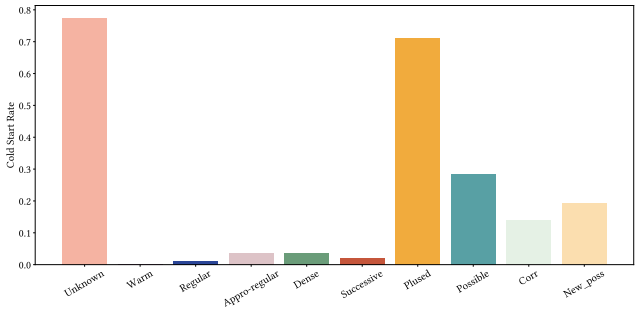
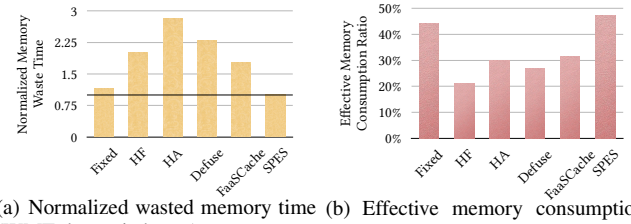


Fig. 13. Averaged cold start rate of each type.

Meanwhile, though "correlated" and "possible/newly-possible" functions also lack predictive information with infrequent invocations, their invocations are more likely to be predicted. Thus, we connect some of them with deterministic functions or adaptively extract meaningful behavior indicators from incoming invocations. In this way, their cold starts are suppressed effectively. We will further discuss the usefulness of such strategy designs in Section V-E.

C. RQ2: Wasted Memory Time and Overhead

1) Wasted Memory Time: Figure 14(a) presents that SPES significantly decreases the wasted memory time by 10.89%–63.50% compared to all baselines. Particularly, compared to Defuse, the most effective baseline on cold-start reduction, we reduce 57.06% of the WMT. Figure 14(b) also demonstrates that SPES efficiently uses memory resources, whose EMCR is 46.32%, 5.20%–120.89% higher than compared approaches. The success of SPES can be attributed to careful pattern modeling and differentiated strategies.



(a) Normalized wasted memory time (WMT, lower is better) (b) Effective memory consumption ratio (EMCR, higher is better)

Fig. 14. SPES significantly reduces memory waste and achieves more effective memory consumption.

To further demonstrate how each type contributes to WMT, we derive a metric called the *ratio of WMT*, which is the WMT divided by the number of invoked times for each serverless function. If a type contains more functions or function invocations, the bespoke scheduling strategy tends to make more proactive provisions, likely resulting in more WMT. Thus, the ratio of WMT can better portray the accuracy of invocation prediction and the usefulness of pre-loading.

Figure 15 shows the distribution of the ratio of WMT among different function types, from which we can see the "possible" functions have the highest probability of generating WMT. The predominant reason for this is the infrequent invocation of possible functions, which leads to a scarcity of patterns in the historical database and complicates establishing associations with other functions. Despite these challenges, we persist in our efforts to forecast and pre-warm the invocation of these functions, aiming to reduce cold starts as much as possible. Different from "pulsed" or "unknown" functions that are allowed to generate cold starts, we encourage aggressive prediction attempts for "possible" functions since the latter have at least a duplicated WT, enabling potential predictive value obtaining. However, this strategy unavoidably results in augmented resource wastage.

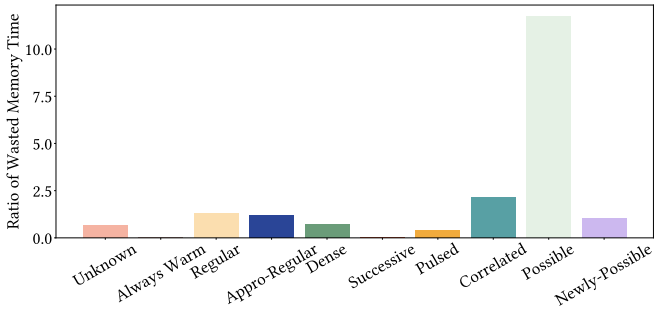


Fig. 15. The ratio of WMT of each function type.

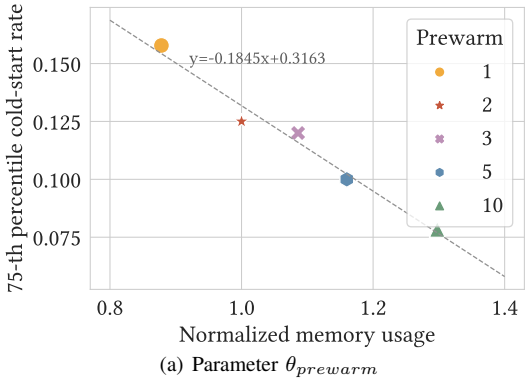
2) SPES Overhead: Herein, we measure the additional latency induced by our implementation. The time complexity within each time window of our approach and the fixed keep-alive policy is $\mathcal{O}(n)$, where n represents the number of functions, as it only requires accessing the corresponding function's category and its predictive value(s). For each invocation, the time complexity of pre-load/unload operations is merely $\mathcal{O}(1)$. Within each time window, the time complexity for the fixed keep-alive method is $\mathcal{O}(m)$, where m is the number of loaded functions. In contrast, FaaSCache has a time complexity of $\mathcal{O}(\log m)$, which involves identifying the container with the lowest priority. Other methods, such as HA, HF, and Defuse, have higher complexities, mainly due to the computational bottleneck in updating histograms. The simulation results align with these expectations. Compared to the fastest fixed keep-alive baseline, which has an average overhead of 0.024 sec per minute, our method adds 0.44 sec's overhead per minute, mainly due to our more complex strategies for each provision action. Contrasted with the second fastest, FaaSCache, our overhead is reduced by 6.8%. In summary, our overhead is inconsequential compared to the typical latency found in most existing serverless platforms.

D. RQ3: Trading-off resources and latency

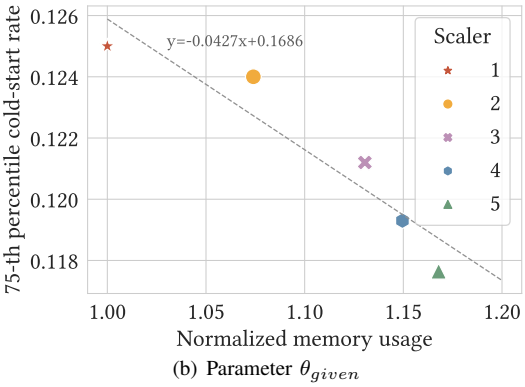
We control the trade-off through two parameters: $\theta_{prewarm}$ and $\theta_{givenup}$. As introduced in Section IV-C4, the former decides how long to pre-load a function with a predicted nearby invocation, and the latter decides how long to keep an idle function warm. Intuitively, the larger these two parameters, the more likely a function is to pre-load or keep loaded with more memory usage and potential wasted memory, and the less cold starts. In RQ1 (§V-B), we set $\theta_{prewarm}$ as two, and the $\theta_{givenup}$ for "dense" and "plused" is five, whereas one for the other types. Our original simulation setting is denoted by the red star ($\star$) in the following figures.

Figure 16(a) shows the trade-off under different $\theta_{prewarm}$, where a point (x, y) represents using x-unit memory and obtaining the 75-CSR of y, under a certain $\theta_{prewarm}$. The memory is normalized to that under the original setting. The normalized memory usage and the 75-CSR are nearly linearly correlated. We can conveniently choose a proper setting by controlling $\theta_{prewarm}$. Since the red star is below and the most distant from the fitting line, $\theta_{prewarm} = 2$ is the optimal value.

As $\theta_{givenup}$ should be integers and different from each type, we simply multiply the original $\theta_{givenup}$ setting by 2, 3, 4, and 5, respectively. The results are shown in Figure 16(b), where the linear relationship still approximately holds, but larger $\theta_{givenup}$s have less impact on cold start reduction. This indicates that keeping invoked functions too long is sub-optimal and idle functions should be evicted promptly.



(a) Parameter $\theta_{prewarm}$



(b) Parameter θ_{given}

Fig. 16. Under different $\theta_{prewarm}$, the memory usage and Q3-CSR exhibit an approximately linear relationship. Similarly, under different θ_{given}, the linear relationship still approximately holds, yet dramatically increasing the memory does not guarantee cold start mitigation.

E. RQ4: Impact of Strategy Designs

1) Impact of inter-function correlations: This section gauges our design in processing ill-informed functions by developing inter-function correlations. As introduced in §IV-C2, we propose a simple yet effective *T-lagged co-occurrence rate* metric to connect ill-informed functions with categorized ones. Those with closely related known functions are "correlated". This strategy is applied during both training and simulation.

Figure 17 presents the impact of this strategy. *w/o Corr* means re-categorizing "correlated" functions into "pulsed", "possible" or "unknown" during training, but we still deal with unseen functions during the simulation. *w/o Online-Corr* denotes removing the simulation-applied strategy, *i.e.* regarding unseen functions as "unknown" but still retaining the "correlated" functions obtained during training. It is shown that the latter strategy slightly reduces the Q3-CSR, whereas the former makes a significant contribution. We attribute the results to the influenced function number. 4.71% of the functions belong to "correlated" whereas only 1.89% are unseen.

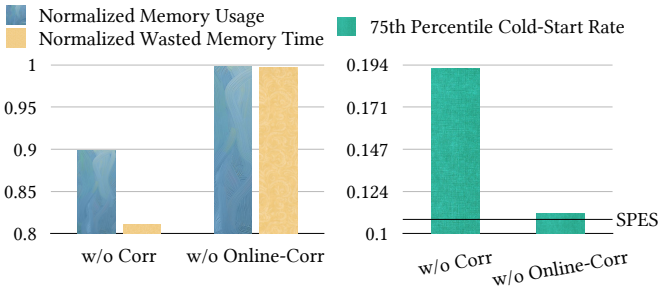


Fig. 17. The correlation strategy contributes to the cold-start and memory-waste reduction.

2) Impact of designs regarding concept shifts: This section studies the impact of two designs regarding concept shifts: 1) *forgetting*: forcing unknown functions to a defined type by ignoring older invocations while focusing more on recent data (§IV-C2); 2) *adjusting*: adaptively adjusting the predictive values during the simulation (§IV-C3). Figure 18 depicts the effectiveness after omitting these two adaptive designs, respectively. Removing the second shifting strategy has a slighter impact. Similarly, this is attributed to the fact that the forgetting strategy involves more function categorization efforts. We do not pre-load unknown functions, so the forgetting strategy categorizing 340 unknown functions has a larger impact. In contrast, the adjusting strategy only categorizes 174 unknown functions into the "newly-possible" type and updates the predictive values of 499 "(appro-)regular/dense" functions, resulting in less impact. Nevertheless, both designs contribute to the effectiveness of SPES. These designs will play a greater value with more data and a longer simulation period (usually indicating larger shifts).

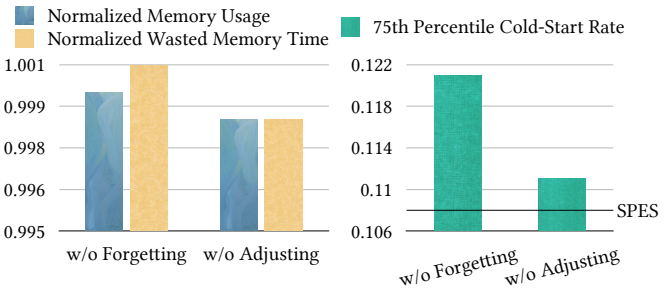


Fig. 18. SPES' adaptivity benefits cold-start reduction.

VI. CONCLUSION

This survey has reviewed three state-of-the-art approaches—SPES, Hybrid, and Defuse—that address the cold start issue in serverless computing while optimizing the performance-resource trade-off. Each approach offers distinct strategies for improving serverless function provisioning, reducing latency, and minimizing resource wastage. SPES employs a differentiated scheduling strategy based on predictable invocation patterns to optimize function pre-loading and unloading, thereby reducing both cold start latency and memory overhead. Hybrid characterizes the FaaS workload and proposes a resource management policy that effectively balances cold start mitigation with resource efficiency, making it suitable for large-scale cloud environments. Defuse introduces a dependency-aware scheduling method that leverages function invocation patterns to reduce cold starts, improving system efficiency. These approaches present valuable contributions to the optimization of serverless function deployment. Further research is needed to refine these methods and explore hybrid solutions that integrate multiple strategies, thereby enhancing the scalability, responsiveness, and cost-efficiency of serverless systems.

ACKNOWLEDGMENTS

This work was carried out as a report of my project for the Advanced Operating Systems (CSC 6032) course. I would like to express my sincere gratitude to Professor Yeh-Ching Chung for his guidance and support throughout the project. I also wish to thank Teaching Assistant ShiHao Hong for his valuable assistance and feedback. Their contributions were instrumental in the completion of this work.

REFERENCES

[1] K. Namee, R. Phoarun, G. M. Albadrani, J. Polpinij, S. Tanessakulwattana, and P. Sphanphong, "A form and API data management platform for progressive web application and serverless application architecture," in *CIIS 2019: The 2nd International Conference on Computational Intelligence and Intelligent Systems, Bangkok, Thailand, November, 2019*. ACM, 2019, pp. 144–149. [Online]. Available: https://doi.org/10.1145/3372422.3372452

[2] R. Kesavan, D. Gay, D. Thevessen, J. Shah, and C. Mohan, "Firestore: The nosql serverless database for the application developer," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2023, pp. 3376–3388. [Online]. Available: https://doi.org/10.1109/ICDE55515.2023.00259

[3] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: a native serverless system for low-latency, high-throughput inference," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 2022, pp. 768–781. [Online]. Available: https://doi.org/10.1145/3503222.3507709

[4] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. H. Katz, "Cirrus: a serverless framework for end-to-end ML workflows," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 2019, pp. 13–24. [Online]. Available: https://doi.org/10.1145/3357223.3362711

[5] H. Yu, H. Wang, J. Li, X. Yuan, and S. Park, "Accelerating serverless computing by harvesting idle resources," in *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*. ACM, 2022, pp. 1741–1751. [Online]. Available: https://doi.org/10.1145/3485447.3511979

[6] S. Gupta, S. Rahnama, E. Linsenmayer, F. Nawab, and M. Sadoghi, "Reliable transactions in serverless-edge architecture," in *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2023, pp. 301–314. [Online]. Available: https://doi.org/10.1109/ICDE55515.2023.00030

[7] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing function workflows on serverless computing platforms," in *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*. ACM / IW3C2, 2020, pp. 939–950. [Online]. Available: https://doi.org/10.1145/3366423.3380173

[8] L. Ao, G. Porter, and G. M. Voelker, "Faasnap: Faas made fast using snapshot-based vms," in *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 2022, pp. 730–746. [Online]. Available: https://doi.org/10.1145/3492321.3524270

[9] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 149:1–149:26, 2019. [Online]. Available: https://doi.org/10.1145/3360575

[10] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[11] R. Haas, R. Niedermayr, T. Roehm, and S. Apel, "Is static analysis able to identify unnecessary source code?" *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, pp. 6:1–6:23, 2020. [Online]. Available: https://doi.org/10.1145/3368267

[12] A. Eivy, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 6–12, 2017. [Online]. Available: https://doi.org/10.1109/MCC.2017.32

[13] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. ": USENIX Association, 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[14] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ": ACM, 2021, pp. 559–572. [Online]. Available: https://doi.org/10.1145/3445814.3446714

[15] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 2021, pp. 386–400. [Online]. Available: https://doi.org/10.1145/3445814.3446757

[16] C. Chen, L. Nagel, L. Cui, and F. P. Tso, "S-cache: Function caching for serverless edge computing," in *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking, EdgeSys 2023, Rome, Italy, 8 May 2023*. ACM, 2023, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3578354.3592865

[17] P. Vahidinia, B. J. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems, COINS 2020, Barcelona, Spain, Aug 31 - Sept 2, 2020*. IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/COINS49042.2020.9191377

[18] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018, pp. 57–70. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/oakes

[19] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ": ACM, 2020, pp. 467–481. [Online]. Available: https://doi.org/10.1145/3373376.3378512

[20] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: enable efficient workflow execution for function-as-a-service," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 2022, pp. 782–796. [Online]. Available: https://doi.org/10.1145/3503222.3507717

[21] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, jul 2023. [Online]. Available: https://doi.org/10.1145/3585007

[22] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu, "Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms," in *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*. ": IEEE, 2021, pp. 194–204. [Online]. Available: https://doi.org/10.1109/ICDCS51616.2021.00027

[23] 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 *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020*. IEEE, 2020, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ACSOS49614.2020.00020

[24] J. Sampé, G. Vernik, M. S. Artigas, and P. G. López, "Serverless data analytics in the IBM cloud," in *Proceedings of the 19th International Middleware Conference, Middleware Industrial Track 2018, Rennes, France, December 10-14, 2018*. ACM, 2018, pp. 1–8. [Online]. Available: https://doi.org/10.1145/3284028.3284029

[25] C. Lee, Z. Zhu, T. Yang, Y. Huo, Y. Su, P. He, and M. R. Lyu, "Spes: Towards optimizing performance-resource trade-off for serverless functions," *arXiv preprint arXiv:2403.17574*, 2024.

[26] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*. ": ACM, 2022, pp. 289–305. [Online]. Available: https://doi.org/10.1145/3542929.3563468

[27] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: warming serverless functions better with heterogeneity," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ": ACM, 2022, pp. 753–767. [Online]. Available: https://doi.org/10.1145/3503222.3507750

[28] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. ": ACM, 2020, pp. 356–370. [Online]. Available: https://doi.org/10.1145/3423211.3425690

[29] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd, "Faasrank: Learning to schedule functions in serverless platforms," in *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Washington, DC, USA, September 27 - Oct. 1, 2021*. ": IEEE, 2021, pp. 31–40. [Online]. Available: https://doi.org/10.1109/ACSOS52086.2021.00023

[30] B. Sethi, S. K. Addya, and S. K. Ghosh, "Lcs: Alleviating total cold start latency in serverless applications with lru warm container approach," in *Proceedings of the 24th International Conference on Distributed Computing and Networking*. New York, NY, USA: Association for Computing Machinery, 2023, p. 197–206. [Online]. Available: https://doi.org/10.1145/3571306.3571404

[31] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. ACM, 2020, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3423211.3425682

[32] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018, pp. 923–935. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/akkus

[33] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo, "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *2022 USENIX Annual Technical Conference, ATC Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 2022, pp. 69–84. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[34] K. Suo, Y. Shi, X. Xu, D. Cheng, and W. Chen, "Tackling cold start in serverless computing with container runtime reusing," in *Proceedings of the 2020 Workshop on Network Application Integration/CoDesign, NAI@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*. ACM, 2020, pp. 54–55. [Online]. Available: https://doi.org/10.1145/3405672.3409493

[35] A. Mohan, H. S. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/mohan

[36] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proc. VLDB Endow.*, vol. 15, no. 8, pp. 1591–1604, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p1591-burckhardt.pdf

[37] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, and B. C. Ooi, "Serverless data science - are we there yet? A case study of model serving," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 2022, pp. 1866–1875. [Online]. Available: https://doi.org/10.1145/3514221.3517905

[38] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. R. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*. ACM, 2021, pp. 153–167. [Online]. Available: https://doi.org/10.1145/3472883.3486992

[39] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards demystifying serverless machine learning training," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2021, pp. 857–871. [Online]. Available: https://doi.org/10.1145/3448016.3459240

[40] Z. Zhou, Y. Zhang, and C. Delimitrou, "AQUATOPE: qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ": ACM, 2023, pp. 1–14. [Online]. Available: https://doi.org/10.1145/3567955.3567960

[41] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*. ACM, 2021, pp. 138–152. [Online]. Available: https://doi.org/10.1145/3472883.3486981

[42] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018, pp. 133–146. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[43] C. Q. Adamsen, A. Møller, S. Alimadadi, and F. Tip, "Practical AJAX race detection for javascript web applications," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 38–48. [Online]. Available: https://doi.org/10.1145/3236024.3236038

[44] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. ": USENIX Association, 2021, pp. 443–457. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/wang-ao

[45] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[46] G. E. Box and D. A. Pierce, "Distribution of residual autocorrelations in autoregressive-integrated moving average time series models," *Journal of the American statistical Association*, vol. 65, no. 332, pp. 1509–1526, 1970.

[47] Pmdarima. [Online]. Available: https://github.com/alkaline-ml/pmdarima

[48] A. OpenWhisk. How openwhisk works. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/about.md

[49] IBM. Ibm cloud functions. [Online]. Available: https://www.ibm.com/cloud/functions

[50] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993, pp. 207–216.

[51] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data mining and knowledge discovery*, vol. 8, pp. 53–87, 2004.

[52] Y. Niwa and Y. Nitta, "Co-occurrence vectors from corpora vs. distance vectors from dictionaries," *arXiv preprint cmp-lg/9503025*, 1995.

[53] K. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Computational linguistics*, vol. 16, no. 1, pp. 22–29, 1990.

[54] Microsoft. (2023) Azure functions triggers and bindings concepts. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings

[55] ——. (2023) Azure functions scenarios. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-scenarios?pivots=programming-language-csharp

[56] M. Azure and M. Research. (2019) Azure functions trace 2019. Microsoft Research. [Online]. Available: https://azurecloudpublicdataset2.blob.core.windows.net/azurepublicdatasetv2/azurefunctions_dataset2019/

[57] A. Kuster, J. Shen, M. Dai, and J. Guo. (2022) reproducing-serverless-in-the-wild. [Online]. Available: https://github.com/andreaskuster/reproducing-serverless-in-the-wild