The Journal of Supercomputing (2022) 78:5358–5393
https://doi.org/10.1007/s11227-021-04057-z

ETAS: predictive scheduling of functions on worker nodes of Apache OpenWhisk platform

Ali Banaei[1] · **Mohsen Sharifi**[1]

Accepted: 25 August 2021 / Published online: 23 September 2021

Abstract
Fast execution of functions is an inevitable challenge in the serverless computing landscape. Inefficient dispatching, fluctuations in invocation rates, burstiness of workloads, and wide range of execution times of serverless functions result in load imbalances and overloading of worker nodes of serverless platforms that impose high latency on invocations. This paper is concentrated on function scheduling within worker nodes of Apache OpenWhisk serverless platform and presents ETAS, a predictive scheduling scheme to reduce response times of invocations besides increasing workers throughputs and resource utilizations. ETAS schedules functions using their execution times, estimated by their previous execution's history, arrival times, and containers' status. We have implemented ETAS in Apache OpenWhisk and show that, compared to the OpenWhisk worker scheduler and several queue scheduling schemes, ETAS outperforms others by reducing the average waiting time by 30% and increasing the throughput by 40%.

Keywords Serverless computing platforms · Function scheduling · Apache OpenWhisk · Latency · Predictive scheduling

1 Introduction

Scalability and elasticity are two features of cloud computing that, after a decade, are out of reach for many cloud users [1]. Although cloud computing has relieved developers from the burden of managing the physical infrastructure, virtual resource management is still required for software deployments [2, 3]. These shortfalls, alongside the shift of enterprise application architectures to containers and

✉ Mohsen Sharifi
msharifi@iust.ac.ir

Ali Banaei
ali_banaei@cmps2.iust.ac.ir

[1] School of Computer engineering, Iran University of Science and Technology, Tehran, Iran

microservices, have led to the evolution of a new paradigm for the deployment of cloud applications called "Serverless Cloud Computing" [1–4].

Serverless computing (or Function as a Service - FaaS) completely decouples backend infrastructure maintenance from application development and users and hides the server management from users [5, 6]. In serverless computing, the cloud provider takes full responsibility for server management, executing functions, capacity planning, resource allocation, task scheduling, scalability, deployment, operational monitoring, and security patches [2, 4–7]. Serverless computing is a realization of event-driven programming wherein applications use small and stateless functions (or handlers) and events that trigger them. Users just need to upload the code, invoke stateless functions by triggering an event, and pay only for the time their code has been run to completion [3, 4, 6–10].

Shifting the responsibilities of server management to cloud providers has benefits and challenges for both cloud providers and users. From the users' viewpoint, besides the fact that they no longer need to worry about server management, they are provided with a simplified programming model for creating cloud applications that abstracts away many operational concerns. Autoscaling and scaling to zero that provides real pay-as-you-go billing are additional positive features of serverless computing. On the other hand, the lack of support for diverse programming languages and libraries, state management, monitoring, debugging, and limited execution times are examples of a number of challenges facing cloud users [4, 6, 7, 9]. From the perspective of a cloud provider, controlling the entire resources, reducing operational costs by efficient optimization, and management of cloud resources are new serverless opportunities. Cold start (time taken to launch a new function instance), scheduling policies, scaling, performance prediction, dynamic provisioning of resources in response to load, I/O bottlenecks, communication through slow storage, pricing models and security and privacy are considered as the most substantial challenges too [1–5, 7, 9, 11, 12].

Scheduling and initializing a sandbox for function execution can impose a notable latency [2]. For each invocation, the scheduler determines the placement and routes it to the correct core on the selected machine. Given that the scheduling policy can have a significant impact on system performance and invocation latency, it should look for a cost-effective place to run an invocation request [11, 13].

The challenge of proactively provisioning resources in response to load, without prediction of future load, requires smart scheduling policies [4, 11, 14]. Burstiness of function invocations raises the queueing delay of functions waiting for resources to become available, along with the latency for deploying and initializing a new function instance, are other issues that a scheduler must deal with [14–16]. There has been a considerable amount of prior works that have investigated this challenge from various aspects [13, 15–20]. Packing as many functions as possible into a single worker node improves the resource utilization and cost effectiveness [18, 19, 21]. This consolidation strategy alongside fluctuating invocation rates and inefficient dispatching can increase the load of workers and queueing time [18, 22]. Resource utilization, execution latency, system throughput, cold start rate, and QoS satisfaction are parameters that are all affected by the scheduling policy pursued by worker nodes. Although most serverless functions are short-lived, the range of execution

times is wide, from 100ms to 15min [15, 16]. A simple scheduling scheme can starve long-term functions or impose a long waiting-time on short-term ones. Dissipation of worker resources and high incidence of cold start are other consequences of any unsuitable scheduling policy [11].

To alleviate the aforenoted issues, we concentrate on the worker nodes' scheduling policy on serverless platforms and for the first step, we propose a predictive scheduling scheme for scheduling the serverless functions in the worker nodes (invokers) of Apache OpenWhisk platform[1], called ETAS (Execution Time-Aware Scheduling), which pursues two objectives, the first priority is to reduce the average waiting times of functions and increasing the throughputs of workers, and the second priority is to enhance utilization of worker resources.

ETAS is a queue scheduling scheme that includes two phases, predicting the execution times of invoked functions and scheduling them based on their predicted execution times. Execution time of a function is predicted using its previous execution history. To obtain the predicted time, the average of prior execution times is computed using exponential formula [23]. Scheduling is accomplished using the arrival times and the predicted execution times of invoked functions, the number of unallocated resources, and the states of existing instances.

We have implemented ETAS in the invoker component of Apache OpenWhisk platform (Sect. 2.2). To evaluate ETAS, we have designed a new benchmark suit with 4 different functions (based on benchmarks presented in [24]) in both Python and PHP. We have considered 2 classes of experiments to investigate the behavior of ETAS in different workloads. In the first class of experiments, the impacts of different factors have been observed individually and in the second one, the effects of the combination of all factors have been investigated. To provide a credible evaluation, we have also implemented SJF (Shortest Job First) [23], BJF (Best Job First) [25], and MQS (Multi Queue Scheduling) [26] scheduling schemes in OpenWhisk. We have compared these scheduling schemes using four fundamental measured parameters in our experiments: average waiting time, worker throughput, resource utilization, and the rate of use of warm instances.

The results of experiments indicated that ETAS reduces the average waiting time up to 30%, 10%, and 8% in comparison with the default worker (invoker) scheduler of OpenWhisk, namely the FCFS (First Come First Serve), SJF and BJF schemes, respectively. Throughput of a worker node also increases up to 40% and 2% compared to FCFS and BJF schemes. The results also showed that ETAS improves the resource utilization up to 1% and reuse of warm instances by up to 2% in comparison with other schemes. Moreover, when functions are invoked over time, ETAS does not impose high latency on function executions with long execution times and has lower average waiting time compared to SJF.

In summary, the key contributions of this paper are:

- Presentation of a new scheduling scheme, called ETAS, which concentrates on the efficient scheduling of functions on worker nodes of Apache OpenWhisk platform.

[1] https://openwhisk.apache.org/.

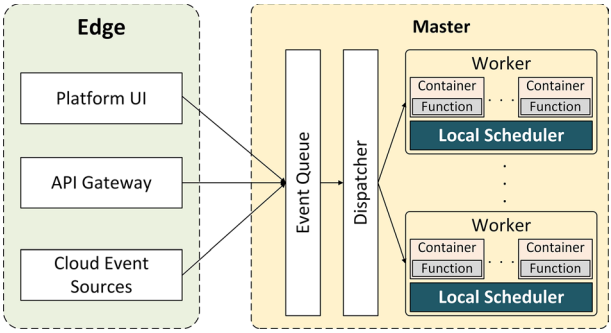


Fig. 1 Blueprint of serverless platform architecture

- Providing the real implementation of ETAS in OpenWhisk and its efficiency assessment in comparison with the schedulers mostly used in serverless computing platforms, namely SJF [23], BJF [25], and MQS [26].
- Design a new benchmark suit based on the benchmarks presented in [24] to fully evaluate ETAS experimentally.

The rest of paper is organized as follows. In Sect. 2, the serverless background and OpenWhisk architecture are described and related works are reviewed. Section 3 presents our scheduling scheme and its formalization. Section 4 elaborates the implementation and evaluation of ETAS, and Sect. 5 concludes the paper.

2 Background and related work

In this section, first, we provide an overview of serverless and Apache OpenWhisk architectures and then present notable related works.

2.1 Serverless architecture

Serverless computing has emerged as a new paradigm for developing and deploying cloud applications to provide zero administration by completely decoupling backend infrastructure maintenance from application development. In the serverless computing model, applications consist of standalone, small, stateless functions or handlers that are executed in response to events [4–6, 27, 28]. Cloud providers use isolated sandboxes (e.g., containers) to execute the handlers. This level of isolation along with stateless functions has led to autoscaling and real pay-as-you-go notions [2, 3, 7].

The core functionality of serverless platform architecture is depicted in Fig. 1 (adopted from [4]). According to this architecture, serverless platforms accept an event that is sent over HTTP or from another event source in the cloud. Internally, the platform consists of an event queue, a dispatcher, and workers. Once a request is received, the system determines which action(s) should handle the event. Then, the

dispatcher routes the request to a worker after fetching the code of the action(s) from a database.

The worker is responsible for executing the action(s). When the worker receives the action(s), if there are enough resources for the function, it launches one or more function instances (depending on the request volume) with restricted resources and sends the action(s) into the instance(s) (usually containers) to execute. Otherwise, the function(s) is enqueued until the required resources are released [18, 22, 29]

A smart load balancer can stop worker nodes to become overloaded. However, in the case of high rates of invocations and burstiness, a long execution queue resulting in long waiting times of invocations is formed. Moreover, in the case of using least nodes to respond to the invoked functions to increase resource utilization, with respect to the Quality of Service (QoS), some functions may be kept waiting in the queues of nodes. Hence, the local scheduling scheme has a critical impact on the average waiting times of functions, throughputs and resource utilization of worker nodes [17, 22, 30].

Launching an instance requires the initialization of the container with the necessary libraries, which can incur undesired startup latency to the function execution which is known as "cold start." To avoid cold start latency, cloud providers pause the instances after execution to reuse them in future invocations. After completing the function(s) executions or reaching the maximum execution time, the worker gathers execution logs and makes the response available to the user. Scale-to-zero allows users just to pay for the time and resources used when their functions are running [4–6, 22, 27].

2.2 Apache OpenWhisk architecture

Figure 2 depicts the Apache OpenWhisk architecture[2] that consists of two main components, Controller and Invoker that are constructed on top of Nginx[3], Kafka[4], Docker[5], and CouchDB[6]. All of these components come together to form a "serverless event-based programming service." OpenWhisk exposes a RESTful interface for submitting the functions and polling the execution results. Nginx forwards incoming requests to the Controller that is responsible for authentication and fetching the invoked functions from the database and dispatching them toward the invokers. Invokers, which are distributed among multiple machines and host the containers of serverless functions, are responsible for executing the invocations by provisioning resources within Docker containers and dedicating one to them. In other words, the invokers play the worker nodes role (Fig. 1) in OpenWhisk.

There is an in-memory queue on each invoker to enqueue functions, in case that there are not enough resources for a container. After release of required resources, functions are dequeued to execute according to First Come First Serve (FCFS) scheme. All invokers copy the same instructions that are written in the invoker component source code (written in Scala). As a result, all invokers work the same way.

[2] https://openwhisk.apache.org/documentation.html
[3] https://www.nginx.com/.
[4] https://kafka.apache.org/.
[5] https://www.docker.com/.
[6] https://couchdb.apache.org/.

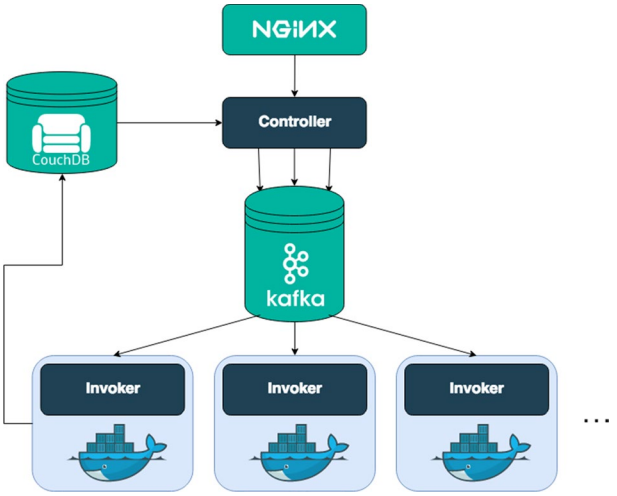


Fig. 2 Apache OpenWhisk architecture [31]

Users can register and upload their functions on the platform. The user can only specify the amount of memory usage required for executing the function. In the current implementation of OpenWhisk, only 128, 256, and 512MB are allowed.

2.3 Related work

Serverless computing is a new general-purpose compute abstraction that promises to become the future of cloud computing and includes many opportunities for improving the current performance shortcomings and overcoming its current limitations, making it attractive to researches [2]. Besides, it has been used in different areas such as parallel data processing [1, 32], Edge-Computing [33], low-latency video processing [34, 35], building adaptive and scalable networks [36], and machine and deep learning [37–39]. Moreover, some work has been concentrated on scheduling particular applications such as scientific workflow, analytics frameworks, and chained functions on serverless platforms [40–42].

There has been some research on stateful serverless computing [43–45], benchmark for serverless platforms [24, 30, 46–48], and developing new frameworks [3, 15, 28, 49–51]. Another group of researchers have investigated serverless challenges, performance issues, and performance prediction [2, 4, 5, 8–12, 52].

Cold start and scheduling policies are two main challenges of serverless platforms that have a significant impact on system performance and invocations' latency. Oakes et al. [53], Fingler et al. [54], and Agache et al. [55] have introduced novel and more lightweight sandboxes for serverless execution environments to decrease the setup time and cold start latency for invoked functions.

Akkus et al. [50] have presented SAND as a new serverless platform. They have redesigned function isolation level so that instead of using a distinct container for

each function, all functions of an application are executed by one container. They have also designed a hierarchical message bus in order to separate internal requests from external ones.

Bermbach et al. [56] have used knowledge on the composition of functions and predicted the future invocations. As a result, an instance(s) can be initialized before invocation(s) to save startup time. They have implemented their scheme as a light-weight middleware that can be co-deployed with serverless functions.

Oakes et al. [57] have investigated lean microservices that require large libraries. Developers use libraries to write their programs; therefore, large packages and libraries are required to be downloaded and installed whenever a function is called, which is extremely time consuming. In response to this problem, they have proposed Pipsqueak, a package-aware serverless platform based on OpenLambda [3] that caches the packages from the Python Package Index (PyPI) and preloads them before function execution.

Zhengjun Xu et al. [58] have introduced an Adaptive Warm-Up (AWU) and Adaptive Container Pool Scaling (ACPS) strategy to reduce cold start latency by predicting the invocation patterns of functions to warm up containers in advance.

Some other works have concentrated on resource management and scheduling in serverless platforms in order to increase resource utilization and reduce the response times of invocations. Notable works are as in the following.

Aumala et al. [59] have proposed PASch, a scheduling algorithm that seeks the packages of the invoked functions and dispatches them to a worker that is preloaded with proper packages. As a result, the functions can use the packages that are cached in the worker to reduce the initialization times of containers.

Stein [20] has investigated different approaches to the scheduling of functions and proposed a Non-cooperative Online Allocation Heuristic (NOAH) to schedule functions in serverless platforms to minimize the occurrence of cold starts. A waiting-time threshold is chosen and the number of instances that are required to keep the waiting-time below the threshold are estimated heuristically. Each worker node independently decides whether to enqueue a request or launch a new instance according to the estimation and the threshold.

Kaffes et al. [16] have argued for a cluster-level centralized and core-granular scheduler for serverless functions and have proposed a new cluster-level scheduler for serverless functions. There are various schedulers that each one has a group of idle workers. Each scheduler assigns an incoming request to an idle worker. If a scheduler has no idle worker, it can steal an idle worker from other schedulers.

De Palma et al. [60] have presented Allocation Priority Policies (APP) aiming to increase data locality in serverless platforms. APP is a language that can be written by both developer and provider and specifies scheduling policies for functions. APP attempts to select a worker on the same site with the database or at least with the same network domain.

There are also some works that have investigated the load balancer scheduler and schedulers of worker nodes. Notable works are as in the following.

Suresh et al. [19] have presented a scheduler for serverless functions, called FnSched, to minimize provider resource costs by imposing acceptable latencies to requests. FnShed predicts resource consumption and lifetime patterns of invoked

functions using classification and schedules functions based on their class. FnSched attempts to maximize the number of functions that can be executed simultaneously on a single worker node by mitigating CPU contention on the worker. Additionally, they have presented ENSURE [21] that relies on FnSched by adding an autonomous resource management. ENSURE concentrates on resource efficiency within and across containers. It uses FnSched for scheduling functions in a way to use minimum number of invokers in response to load and uses FnScale to proactively scale out the number of invokers avoiding setup delay. The main objective of these studies has been to increase resource utilization and decrease provider expenditure. In contrast, we have set the reduction in waiting-time the first priority of ETAS.

Singhvi et al. [15] have presented Archipelago, a scheduling framework to provide low latency function execution in a multi-tenant serverless setting while meeting deadlines. To achieve this goal, Archipelago partitions each cluster into smaller worker pools, each scheduled by a semi-global scheduler, and views each application as a DAG of functions. When requests arrive at various rates for different DAGs, they are dispatched to different clusters. Each cluster schedules incoming DAGs to minimize the possibility of missing deadlines. In each cluster, a local scheduler is responsible for scheduling arrived functions according to shortest remaining time criteria.

Gunasekaran et al. [17] have proposed Fifer, a serverless platform that tries to use fewer containers by bin packing jobs and executing the invoked functions to increase resource utilization without violating the application-level Service-Level Objectives (SLOs). Fifer attempts to schedule functions on each worker according to the Least-Slack-First policy by queueing them and predicting their execution times and slacks.

Fifer and Archipelago have tried to schedule functions on each worker node according to deadline priority. They have increased resource utilization without violating SLO constraints. Although ETAS is a worker node scheduler, there are differences between the objectives of these scheduling schemes. ETAS concentrates on reducing the waiting times of functions, while increasing resource utilization is its second priority. Nevertheless, ignorance of the SLO constraints is one of the main shortcomings of ETAS as it is discussed later in Sect. 5.

Almost all the addressed works in this category try to schedule functions on serverless platforms by predicting various features of serverless functions. Similarly, estimation is used by ETAS to estimate the execution times of functions. Alongside serverless computing, the scheduling issue has been argued in other areas in large too. As a result, we study the queue scheduling approaches in other areas that have similar circumstances to the subject matter of our paper.

Mu'alem et al. [61] have investigated resource fragmentation that results from FCFS (First Come First Serve) scheduling and leads to low utilization. A scheduler called EASY has been developed in which small jobs are moved ahead to fill in holes in the schedule, provided they do not delay the first job in the queue. In the enhanced EASY scheduler for workloads on SP2 systems [62], small jobs move ahead only if they do not delay any job in the queue. This scheme is not viable in serverless platforms, due to various unpredictable latency factors such as cold start that make it impractical to accurately estimate the termination times of functions.

Karthick et al. [26] have introduced a Multi-Queue Scheduling (MQS) scheme for cloud computing. MQS divides incoming jobs into three queues based on their burst times. All jobs are sorted in ascending order due to their burst times; the first 40 percent of jobs are considered as small, the next 40 percent as medium, and the last 20 percent as large. Equal weight is given to all queues to schedule the jobs based on first come first serve policy. Although MQS leads to a fairer selection than SJF and has no starvation, it works well only when all jobs arrive in the system at the same time. In the case of streaming of incoming jobs, queues could become unbalanced.

Al-Husainy [25] has proposed a scheduling algorithm for CPU scheduling, called Best Job First (BJF). BJF calculates the f factor for each process at the queue according to following relation: $f = Priority + ArrivalTime + BurstTime$. Then, scheduler selects the process with the lowest value of f. As a result, CPU starts to execute the process with highest priority, shortest burst time, and submitted earlier.

Table 1 compares the related works in terms of the main issues that have been investigated and their objectives or introduced approaches. A glance at the table shows the position of ETAS in comparison with other schemes and its similarities and differences in objectives.

3 Proposed scheduling scheme

In this section, the proposed scheduling scheme for scheduling functions on each worker node is described. First, problem assumptions and our system model are explained. After that, our scheduling scheme is elaborated in detail.

3.1 Problem statement

Wide range of execution times of serverless functions (from 100ms to 15min) [15, 16], besides, no estimation of future load [4] have made scheduling issue in serverless cloud computing more challenging than in traditional cloud services. Moreover, each worker node is responsible for managing its resources by allocating appropriate resources to functions' sandboxes [4, 22, 27]. As a result, we are motivated to propose a second level scheduling scheme for the worker nodes on serverless platforms to reduce response times of functions by decreasing the average waiting times, and increasing throughputs of worker nodes and resource utilization.

3.2 Assumptions

In this section, assumptions underlying our proposed scheduling scheme are stated. First of all, we have considered a system under high workload. The rest of the assumptions are as follows:

1. There are multiple tenants in the system.
2. Each function has its own resource setting (resource setting is the number of resources that is allocated to the function sandbox).

Table 1 Relate work comparison

System	Key issue		Purpose and approach / Works	Changing isolation policy	Pre-warming sandboxes	Increasing the reuse of warm sandboxes	Caching required packages	Optimizing and customized sandboxes	Reducing latencies	Improving resource utilization	Avoiding imbalances	Increasing data locality	Reducing provider costs	Reducing missed deadline rate	Increasing system throughput	Avoid starvation	Prediction
Serverless Platforms	Cold start latency		SAND [50]	✓					✓								
			SOCK [53]				✓	✓	✓								
			Usetl [54]					✓	✓								
			Firecracker [55]					✓	✓								
			Application Knowledge [56]		✓				✓								✓
			Lean Lambdas [57]				✓										
			AFLA [58]		✓				✓								✓
	Scheduling scheme	Load balancer scheduler (global scheduler)	Centralized Core Scheduling [16]						✓		✓						
			NOAH [20]						✓	✓							✓
			PASch [59]				✓		✓		✓						✓
			APP [60]						✓			✓					✓
		Global and local scheduler	Archipelago [15]						✓					✓			✓
			Fifer [17]							✓							✓
			ENSURE [21]							✓			✓	✓			✓
		Worker node (local scheduler)	ETAS			✓			✓	✓					✓	✓	✓
Other Systems	Scheduling scheme		BJF [25]						✓							✓	✓
			MQS [26]													✓	✓
			Backfilling [61]							✓							✓

3. A function can be invoked with various inputs, and its execution time can be affected based on its input.
4. Execution times of functions can be affected by their resource setting.
5. Functions are compute-intensive and do not require network and I/O.
6. Functions have no priority over each other.
7. Functions are independent, and we do not distinguish between a function that is invoked by an external request (from outside of the platform) or internal one (invoked by other functions).

Table 2 Symbols used in the assumed system

Formal notation	Description
$U = \{a_1, \ldots, a_n\}, n \in \mathbf{N}$	Set of users in the system
$F = \{f_1, \ldots, f_m\}, m \in \mathbf{N}$	Set of functions in the system
$R_i^f, i \in \{1, \ldots, m\}, m \in \mathbf{N}$	Required resource of function$_i$
$T_i^e, i \in \{1, \ldots, m\}, m \in \mathbf{N}$	Execution time of function$_i$ in ms
$W = \{w_1, \ldots, w_k\}, k \in \mathbf{N}$	Set of worker nodes in the system
$Q = \{q_1, \ldots, q_k\}, k \in \mathbf{N}$	The set of queues in the system. And q_i belongs to w_i
$R_i^w, i \in \{1, \ldots, k\}, k \in \mathbf{N}$	Total resource capacity of worker$_i$
$FR_i^w, i \in \{1, \ldots, k\}, k \in \mathbf{N}$	Total free resources of worker$_i$
$WC_i, i \in \{1, \ldots, k\}, k \in \mathbf{N}$	Pool of warm containers in worker$_i$
$T_i^a, i \in \{1, \ldots, m\}, m \in \mathbf{N}$	Arrival time of request for function$_i$ (based on origin of time) in ms
$T_i^f = T_i^a + T_i^e, i \in \{1, \ldots, m\}, m \in \mathbf{N}$	Ideal finish time of executing function$_i$ without delay
f^c	A candidate function that scheduler selects to execute after releasing the required resources

3.3 System model

As it is illustrated in Fig. 1, there are many worker nodes on serverless platforms, each having its own resource pool to supply a sandbox and allocate to the functions. Table 2 indicates the symbols we have used in modeling our assumed system, while this model is adopted from Apache OpenWhisk platform.

3.4 ETAS

ETAS consists of two parts, predicting and scheduling. First, ETAS predicts the execution times of functions by profiling the execution times of the previous function executions, then schedules them on worker nodes based on the prediction. To explain in more detail, first, the approach to predicting the execution times is explained, after that the scheduling scheme is described.

3.4.1 Execution time prediction

Estimating the execution times of functions is a crucial part of ETAS. There are various approaches to estimate task performance and execution time [63]. Historical data approach uses the historical data to predict the execution performance and execution time of tasks [64, 65]. Estimating the execution time is also a popular issue in workflow scheduling [66, 67]. Although the other option is to use the execution time estimation stated by user, as workload on parallel super-computers and serverless platforms is highly repetitive, predicting the execution times of functions based on historical data is viable with low overhead [3, 61].

Prediction of the execution times of functions is achieved by profiling the execution times of the previous executions. According to the aforementioned

assumptions, input and resource setting of functions can affect the execution time. As a result, in order to increase accuracy, for each function, resource setting, and input different records have been considered. Therefore, whenever a function is invoked by specific input, the predicted execution time based on the function name, resource setting, and input is returned. In other words, for each tuple of $< Username, Function - name, Resource\ Setting, Input >$ which belongs to a function, there is a record that indicates the predicted execution time of the function. Each registered function with a specific name and resource setting is called "action".

Two scenarios are considered to profile the execution times. After execution of an invoked action, the time is calculated and saved. In the first scenario, no record exists for an invoked action. Thus, the calculated time is recorded as the prediction for the next invocation. On the other hand, if there is a record, the average of the previous record and the new one is calculated and replaced according to an exponential average (Equation 1) [23].

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n, 0 \leq \alpha \leq 1 \tag{1}$$

The value of t_n contains the most recent execution time, while τ_n stores the past history and τ_{n+1}, be the predicted time for the next invocation. The higher value of α, the closer the predicted value to the recent history. Thus, we assume $\alpha = \frac{1}{2}$ so that the recent history and the past history are equally weighted.

Moreover, for each action with a specific resource setting, the weighted average of all predicted times of different inputs is calculated and updated on cascade. This weighted average is used in case that there is no estimation record for an invoked function with specific input. Saving the predicted time in database is the next step.

One of the assumptions we stated in Sect. 3.2 was that there are tenants with various functions, demanding different resource settings that are invoked with various inputs. As a result, there are separated records for each action and input. Hence, we have considered a two-level key-value structure. At first level, there is a key for each action, and the value is again a key-value structure. We use a hash function to create the keys, therefore for the first level the tuple $< Username, Function - name, Resource\ Setting >$ is used as the hash value. For the second level, the key is made by applying the hash function on each input. Figure 3 illustrates the schema of saving the previous execution times of functions. For each resource setting, an entry is considered which contains various times according to each input.

3.4.2 Scheduling

The worker node scheduler is a second-level scheduler (Fig. 4), and the scheduling discipline determines the order of function executions, which has a remarkable impact on the waiting times and throughputs of worker nodes. When a function grabs a container, retrieving the container is not efficient due to the cold start and loss of the execution state. As a result, we choose a non-preemptive scheduling discipline.

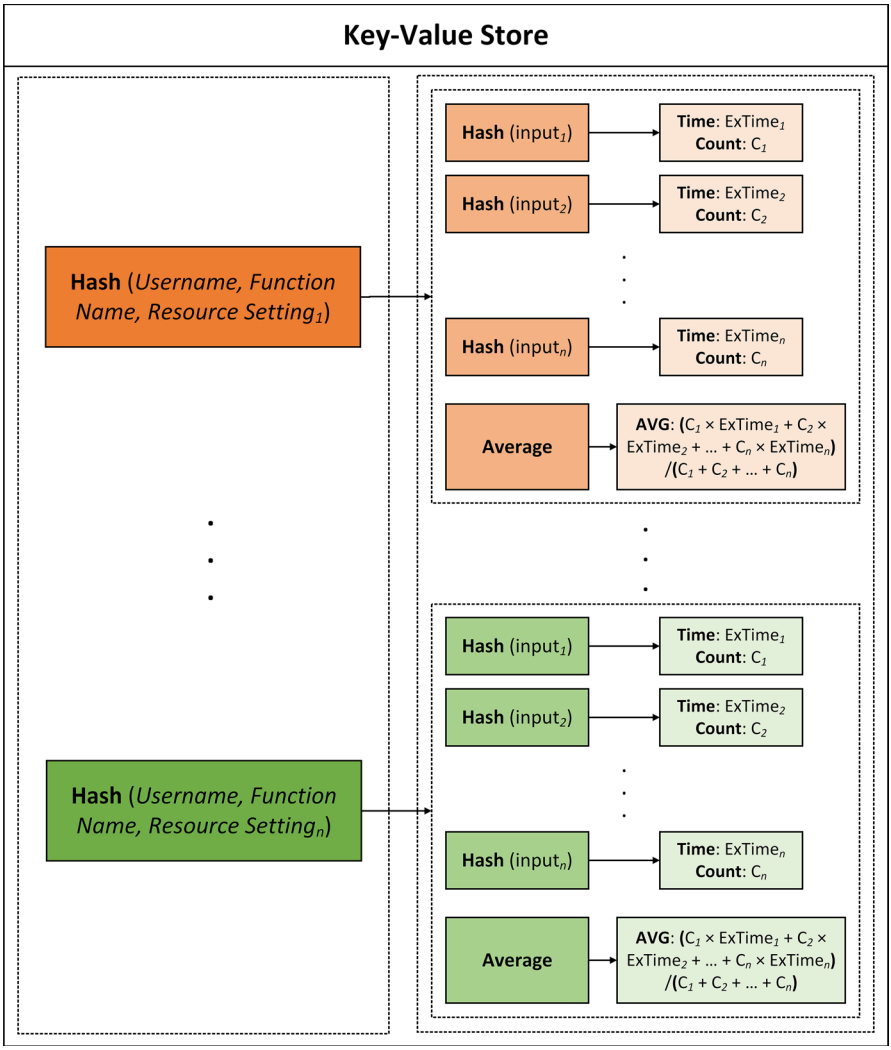


Fig. 3 A key-value structure for storing a function execution time

According to Sect. 2.1, functions are dispatched to the worker nodes to execute in dedicated sandboxes (e.g., containers). If there are not enough resources, functions are enqueued until the required resources are released. The queue scheduler determines which function should be executed when the required resources are released. The ETAS scheduling scheme decides the next function according to the following steps:

1. When a function (f_i) arrives in a worker, its predicted execution time is fetched (T_i^e).

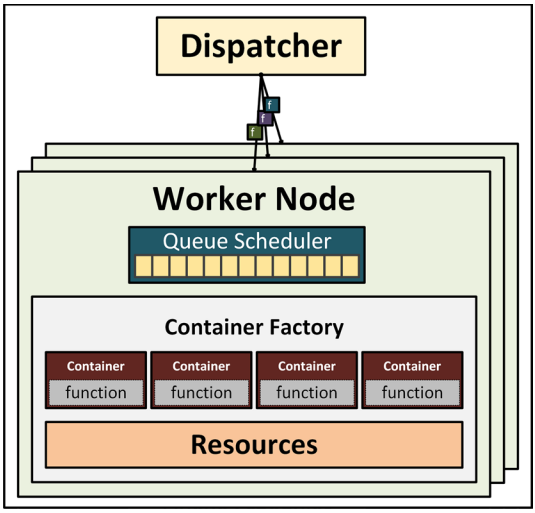


Fig. 4 Worker node architecture

2. Before enqueuing the function (f_i), T_i^f is calculated and stored with the other function's data.
3. When the scheduler needs to select the next function from the queue, it selects a function with minimum T^f and considers it as a candidate function (f^c).
4. Now, if there are enough resources to execute the f^c, it is executed. Otherwise, ETAS is seeking for a function that has a warm sandbox or accessible required resources. First, it is checked whether there is a warm sandbox for any function in the queue that has arrived earlier than f^c and if a function is found, it is executed before f^c, else, the scheduler checks whether there are enough resources for any function in the queue that has arrived earlier than f^c, and if a function is found, it is executed before f^c. Otherwise, f^c is executed.

Algorithm 1 shows how ETAS works, and Fig. 5 demonstrates the workflow of ETAS. Accordingly, the entered function (f_i) can only be executed if either the queue is empty or it is the candidate function (f^c), which is selected by scheduler discipline. Otherwise, related T_i^f is calculated and attached to f_i and enqueued (step 2).

Next, it is checked if there are enough resources to execute f_i or not. If so, f_i is sent to the dedicated container and the next function is selected from the queue (if the queue is not empty) as the candidate function, based on minimum T^f (step 3). Otherwise, in case that there are no resources to execute f_i, the scheduler checks whether the queue is empty (that means f_i is the first function that is enqueued and its T^f should be calculated) or not (that means the function is f^c). As a result, searching for a warm instance or free resources for another function in the queue is carried out according to step 4.

According to Table 2, T_i^f is the summation of arrival time (T_i^a) and the predicted execution time (T_i^e) of f_i. Therefore, ETAS selects a function with minimum T^f. In other words, among the functions that are invoked almost simultaneously the

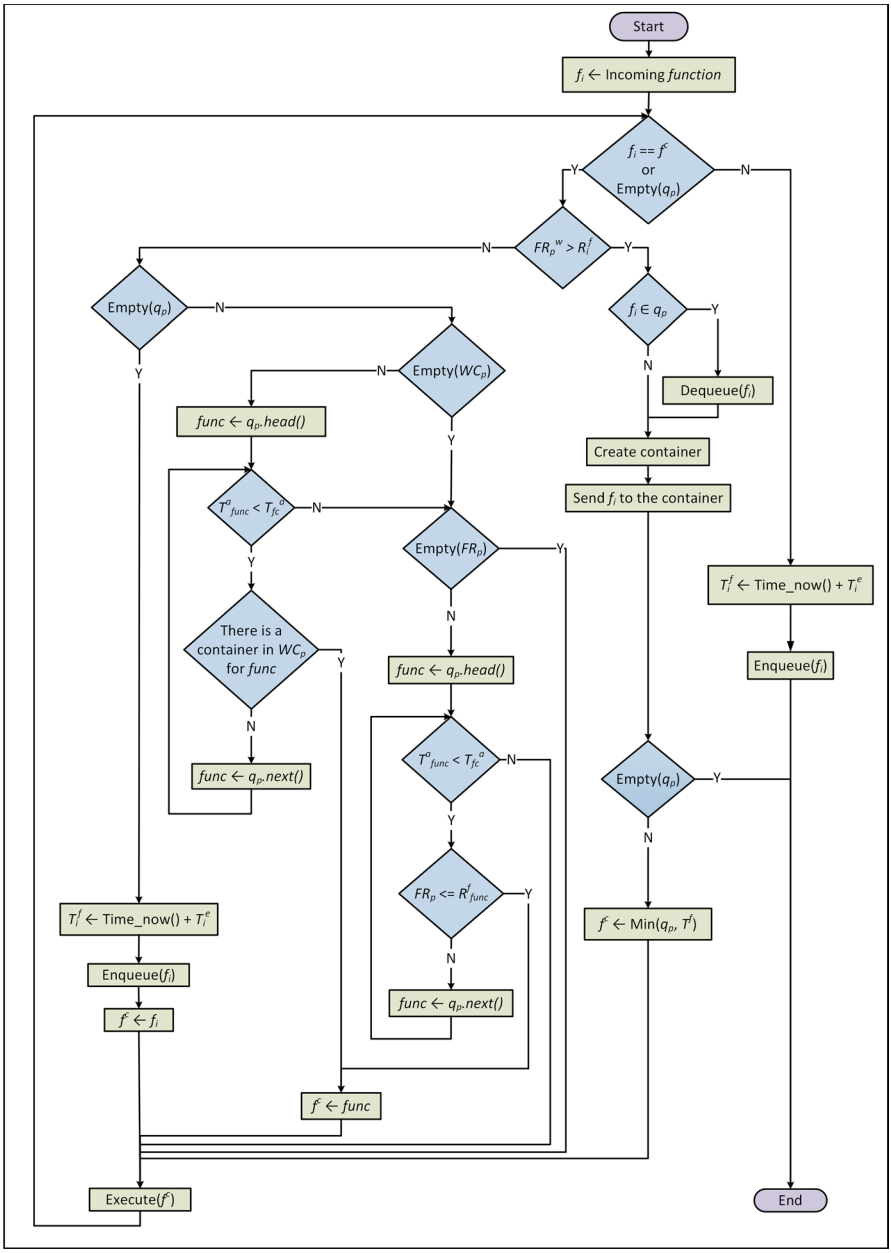


Fig. 5 ETAS workflow

shortest one (based on execution time) and among the functions that are invoked at longer intervals the first one (based on arrival time) are selected. As a result, the

smaller functions (based on execution time) are not stuck behind the larger ones and the large functions are not starved.

Algorithm 1 ETAS Algorithm

Require: A new function ($f_i \in F$, $i \in N$) which is arrived at the Worker node ($w_p \in W$, $p \in N$). And q_p is the queue in w_p

```
FunctionExecute(f_i)
if f_i == f^c or Empty(q_p): then
  if FR_p^w ≥ R_i^f: then
    if f_i ∈ q_p: then
      Dequeue(f_i)
    end if
    c_i = CreateContainer(f_i)
    InvokeContainer(c_i, f_i)
    if notEmpty(q_p): then
      f^c ← Min(q_p, T^f)
      Execute(f^c)
    end if
  else
    if Empty(q_p): then
      T_i^f ← Time_now() + T_i^e
      Enqueue(f_i)
      f^c ← f_i
      Execute(f^c)
    else
      if Empty(WC_p): then
        if Empty(FR_p): then
          Execute(f^c)
        else
          for funcinq_p: do
            if T_func^a < T_fc^a & FR_p ≤ R_func^f: then
              f^c ← func
              Execute(f^c)
              Break
            end if
          end for
        end if
      else
        for funcinq_p: do
          if T_func^a < T_fc^a & WarmContainerExist(WC_p, func): then
            f^c ← func
            Execute(f^c)
            Break
          end if
        end for
      end if
    end if
  end if
else
  T_i^f ← Time_now() + T_i^e
  Enqueue(f_i)
end if
END Function
```

According to Algorithm 1 and Fig. 5, it can be claimed that the order time of ETAS is $O(n^2)$. For the proof, let n be the number of total functions in $q_i \in Q$. As a result, the order of finding minimum T^f equals to $O(n)$. Iterations on q_i to find a function with warm container or free resources are equal $O(2n)$. Hence, the total order time of finding the f^c equals to $O(3n)$. Overall, the order time of ETAS for scheduling n functions is $O(n^2)$.

4 Implementation and evaluation

To evaluate ETAS, we have implemented it on Apache OpenWhisk that is an open-source serverless cloud platform. In the following, first, the implementation of ETAS and other schemes are described. Next, we introduce our benchmarks and experimental setup and finally report our experimental results.

4.1 Implementation

This section provides a complete explanation of the implementation of ETAS. ETAS is implemented within the invoker (Sect. 2.2) component and performs three steps of profiling and predicting execution time and scheduling.

4.1.1 Profiling and predicting execution time

Each action is sent to an invoker as an object type with all attributes (e.g., name, input, username). When an action arrives at an invoker, its predicted execution time, calculated earlier, is fetched from the database and attached to the action's object. The execution time of an action is calculated after the termination of each execution and sent to a new class (called "ExTimeProfiling"), via an asynchronous message, to be saved in the database. "ExTimeProfiling" is implemented to profile the predicted execution time for the function, according to Sect. 3.4.1.

4.1.2 Scheduling

Each action is sent the "ContainerPool" class to catch a sandbox. If there are not enough resources (memory), the action is enqueued. We have implemented the ETAS scheduler in this class according to Sect. 3.4.2. The default scheduler discipline of the OpenWhisk is FCFS. To compare ETAS with other scheduling schemes, we have implemented Shortest Job First (SJF) [23], Best Job First (BJF) [25], and Multi-Queue Scheduling (MQS) [26] schemes.

For a further explanation, a brief description of the aforementioned schemes implemented in the ContainerPool class is provided.

SJF: In this scheme, scheduler selects the action with minimum execution time [23]. As mentioned before, ETAS uses T^f, which is based on execution time and arrival time, to schedule the functions. Hence, if all jobs arrive at the same time, ETAS will perform exactly like SJF. Thus, we have decided to compare ETAS and

SJF in diverse situations. Therefore, in order to compare ETAS with SJF accurately we implemented it in OpenWhisk.

MQS: This scheme is mostly used in cloud systems [26]. According to this scheme, three different queues are considered as small, medium and long based on ascending order of execution times of the jobs. The first 40% of jobs are put in the small queue, the second 40% in the medium queue, and the rest in the long one. After that the jobs are executed based on dynamic selection. Unfortunately, there is no obvious information about how to select the jobs from the queues. We have implemented MQS based on the examples provided in the paper [26].

This research has been evaluated in a situation that all jobs were in the system. In order to study the effect of sequential arrival of jobs on MQS and making a comparison between one-queue and multi-queue scheduling, we have decided to implement MQS.

BJF: This scheme is mostly used in scheduling of processes in operating systems [25]. According to this scheme, processes are selected based on a new factor f. The equation is: $f = Priority + ArrivalTime + BurstTime$. A process with the lowest f is selected to be executed next. This is an incorrect algebraic summation because parameters with different units are added. However, in ETAS, we have used the summation of arrival time and execution time. Therefore, we have implemented a different version of BJF by eliminating Priority from the equation. Therefore, in modified version of BJF, the scheduler selects an action with minimum f using the equation: $f = ArrivalTime + ExecutionTime$. However, in ETAS despite selecting a function with minimum f the functions that have warm container instance or enough free resources are given priority.

Our insistence on implementing this scheme is that we want to investigate that how much the prioritization, which we have considered in ETAS, affects the waiting time. Although we have used our modified version of BJF, as its main idea is similar to ETAS, we have not changed the name to retain our respects for the introducers of BJF.

4.2 Benchmarks

To evaluate ETAS and compare it to FCFS (i.e., the default scheduler of OpenWhisk), SJF, MQS, and BJF, we have designed a benchmark including four computational functions in Python and PHP based on [24]. Functions are: *primeNumber* which finds the nth prime number, *factorial* that calculates the n factorial, *multiplyMatrix* that multiplies two matrices with size n, *sleep* function which sleeps for n seconds. The reason why we have picked these functions is that first, they are all CPU and memory intensive (not I/O intensive) and second, all of them are sensitive to the inputs and resource setting. And finally, their execution times with the same inputs are constant. Three different users are created, according to our system capacity and each function is assigned to all users with all resource settings (with different amount of memory).

4.3 Experimental setup

According to OpenWhisk architecture, all invokers, which host the containers of serverless functions, are copies of the invoker component instructions (Sect. 2.2). As

we are looking for the average waiting times of the functions in the queue, throughput, resource utilization, and rate of reusing warm containers within the invoker, with respect that all invokers are homogeneous, the following experiments are performed on a single invoker.

We have deployed and configured the Apache OpenWhisk with the default settings on a server machine with two 12 cores Intel Xeon CPUs with 2.67 GHz, and 8 GB of memory. We have used the Ubuntu Server 16.04 (kernel version 4.4.0-142-*generic*) as an operating system. Requests have been generated within the same host.

Moreover, a logger has been implemented within the "ContainerPool" to record the details of each invoked action and the queue status, and a watcher to check the invoker's memory usage. The logs are stored in the database and analyzed using another program in Python. The logger saves username, function's name, memory usage, arrival time, number in the queue, the queue length, exit time, queue wait time, total wait time (out of queue), container status, predicted and real execution time for each function. The watcher is supposing to store the total memory usage of the system per second. The watcher stores the total memory usage of the system per second. These parameters are used to calculate the average waiting time, average execution time, throughput, the states of the containers, and memory usage.

Three categories of experiments have been provided to evaluate and investigate ETAS behavior and make a reliable comparison between ETAS and other schemas. In the following section, each section is explained and the related results are reported and discussed in detail.

4.4 Experiments and results

Time intervals of invocations, execution times, the amount of workload, cold start, and functions resource settings are the factors that have effect on the waiting times of functions. In order to provide an accurate evaluation, we have defined two different experiments to calculate the effects of each factor. In the first class, each factor has been measured separately and in the second one, the effect of combining all the factors has been investigated.

ETAS is provided to reduce the average waiting times of functions in overload circumstances. In other words, ETAS is a queue scheduler scheme and is useful when a worker is overloaded and the queue is formed. As a result, all experiments are performed in overload situation.

4.4.1 One factor impact (Experiment 1)

As mentioned before, different factors contribute to the waiting times of functions. As a result, in each test of this category, we investigate the impact of the average time intervals of invocations, the average execution times of functions, the rate of invocations, cold start, and the functions resource settings. All scheduling schemes, except FCFS, need to be aware of the execution times of functions. Therefore, to eliminate the prediction error, we use sleep function, which

its execution time is known in advance. To keep the number of containers, and the resource settings constant, we use one user and an action with 512 MB memory. To avoid the effect of cold start delay, all sandboxes have been launched before starting the test. As a result, because of using one action, all containers are the same and warm. Exponential distribution is used to generate the time intervals and execution times of functions.

Because of using the whole resources with warm containers in the first three tests, the prioritization in ETAS becomes useless. Therefore, ETAS and the customized version of BJF are the same. As a result, this group of tests does not include BJF. Each test is repeated 25 times on each scheme.

- **Time interval impact**

In this test, we are looking for the impact of various time intervals on the average waiting times in each scheme. Therefore, the number of requests is constant and equal to 25 that means in each iteration 25 requests are sent. Average execution times of functions is also constant, which generated according to Exponential distribution with $\theta = 30$ s. The time intervals are generated according to Exponential distribution ($\theta = 0, 2, 4, 6$ s). Figure 6 indicates the average waiting times of actions and the standard deviations along with maximum queue lengths in each scheme. Given that these times are for the waiting times of functions in the queue. As we expected when requests are concurrent ($\theta = 0$) ETAS and SJF behave the same. Because, when the arrival times are the same, T^f is equal to T^e (Table 2). With the increase in time intervals, SJF demonstrates a lower waiting time than ETAS ($\theta = 4$). In contrast, when $\theta = 6$, it seems that ETAS has less waiting time. Since SJF always gives the priority to functions with lower execution time, it may impose long waiting time to functions with higher execution times which results in increasing the average waiting times (This claim is also observed in following experiments).

It is clear that MQS is not a suitable scheme especially when the requests are not concurrent. When the requests are concurrent, MQS has a lower waiting time than FCFS. While, with the increase in time intervals, MQS performs worse than FCFS. In fact, when the requests arrive over time, the overhead of dividing functions between queues increases and imposes more latency to functions. However, when $\theta = 6$, the queue length is equal to 5. According to MQS, 2 functions are enqueued in small queue, 2 functions are enqueued in medium queue, and 1 function is enqueued in long queue. Then, at first, the functions in small queue are executed, after that the functions in medium queue are executed and finally the last function is executed. As a result, it performs just like SJF with more overhead. The results also show that ETAS has the least standard deviation in comparison with other schemes which reveals that no function encounters high latency.

- **Execution time impact**

Execution time variation is another factor that impacts on waiting times of the invocations. In this test, the number of requests is 25 in each iteration, and the

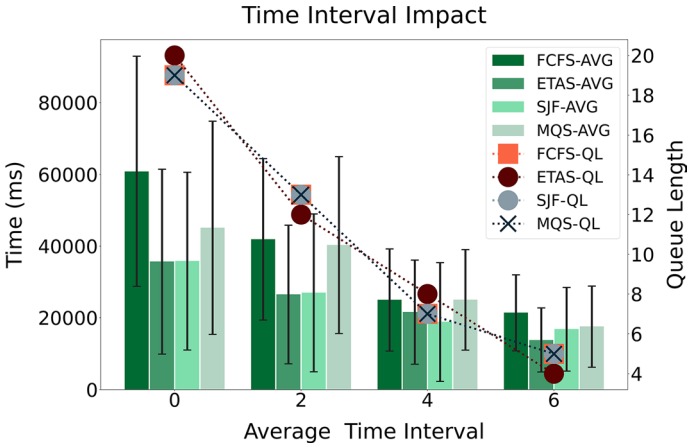


Fig. 6 Average waiting times and standard deviations for actions in the queue with maximum queue length in various average time intervals based on Exponential distribution ($\theta = 0, 2, 4, 6$ s) in each scheme. The number of requests is 25 in each iteration, and the average execution times is 30 s

average time intervals is constant ($\theta = 2$ s), while the average execution times increase($\theta = 5, 10, 15, 20, 25$ s). A glance at Fig. 7 reveals that ETAS and SJF have the least waiting times, especially in higher average execution times. With $\theta = 5$, the maximum number of functions (actions) in the queue is about 1, which is not enough to compare the schemes. With the increase in execution times, as we expected ETAS and SJF outperform other schemes, while FCFS encounters higher latency. When $\theta = 25$, ETAS shows lower latency than SJF. With the increase in execution times, some functions will have longer execution times. Therefore, since SJF gives priority to the functions with lower execution times, the longer ones would suffer from starvation.

Besides, the results also indicate that the longer execution times the less standard deviation in ETAS, which means ETAS attempts to treat all actions in a fair way.

- **Workload impact**

In this test, for the time intervals of requests $\theta = 2$ s, and for the execution times $\theta = 20$ s, while the number of requests increases. It is obvious that the greater number of requests results in the longer length of the queue. Previous tests reveal that MQS has not performed well. As a result, we have been convinced that to not include MQS in following tests.

It is clear from Fig. 8 that with an increase in load of requests FCFS has the most latency while ETAS performs better than other schemes. Moreover, SJF has the most standard deviations of waiting times, while ETAS has the least ones. As mentioned in previous tests, ETAS avoids imposing long latency to the functions by involving arrival time when selecting the next function as a candidate function.

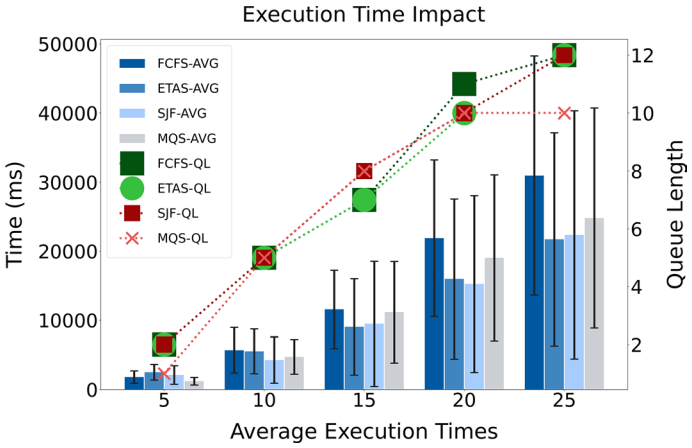


Fig. 7 Average waiting times and standard deviations for actions in the queue with maximum queue length in various average execution times based on Exponential distribution ($\theta = 5, 10, 15, 20, 25$ s) in each scheme. The number of requests is 25 in each iteration and the average time intervals is 30 s

- **Cold start and resource setting impacts**

When an action is invoked, if there is an instance for execution, it is dedicated to the action immediately (warm). Otherwise, a container is created (cold start). In case that there are not enough resources to create a container and one or more containers from paused containers (if exist) are terminated to release enough resources for the new container (recreated). And if there are not enough resources of paused container, the action is enqueued.

Figure 9 indicates the latency of containers startup in different states and sizes of memory. To measure the startup latency, we have defined three different situations, namely Similar, ASC, and DES. In Similar test, the startup latencies are computed based on the action's memory and an action with the same memory is replaced with the previous one (recreated). It is clear that the warm container has the least latency by far while recreating a container imposes the longest latency. Given that our functions do not need any package and library to install. Moreover, our results have shown that although higher memory has less latency, there is no difference between the latency of recreating a container by the same language or a different one.

In ASC and DES, containers are replaced with a number of containers with different sizes of memory. In ASC, first, all resources are occupied with containers with 128 MB. After that, functions with 256 MB and 512 MB are invoked, respectively, while in DES the reverse procedure is performed. It is obvious from Fig. 9 that recreating the containers with 256 MB in ASC test encounters higher latency than DES test. In fact, in ASC test, to initialize a container with 256 MB, two containers with 128 MB should be terminated while in DES, a container with 512 MB would be terminated.

Overall, we can claim that container startup latencies vary and depend on the situation. It is clear that amount of memory has a direct effect on startup delay.

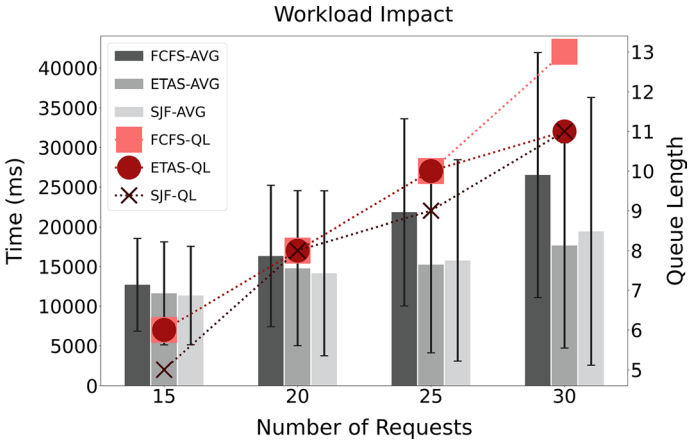


Fig. 8 Average waiting times and standard deviations for actions in the queue with maximum queue length in various workloads in each scheme (average time intervals is 2 s and average execution times is 20 s)

In this test, the impact of cold start and different resource settings is investigated. As a result, different users, functions, and resource settings are used and containers are not becoming warm at first. Three different users along with *sleep* function in both Python and PHP languages with 128, 256, and 512 MB sizes of memory are used. The average time intervals is 2 s ($\theta = 2$), the average execution time is 20 s ($\theta = 20$), and the number of requests is 40. The test is repeated 40 times on each scheme.

Figure 10 shows the average waiting times and the standard deviation for actions in the queue and throughput of the invoker in each scheme. ETAS, BJF, and SJF impose approximately equal waiting times, while SJF has the most standard deviation and throughput. The throughput shows the amount of work that has done in a second. FCFS has the most waiting time with the least throughput.

Figure 11 illustrates the rate of containers' states (Fig. 11a) and resource utilization (Fig. 11b) in each scheme. A glance at the charts reveals that ETAS by giving priority to the functions that have warm containers and the functions that there are enough resources to execute has the most reusing of warm containers and resource utilization. Although the result is not so significant, prioritization has no negative effect on waiting times.

4.4.2 Various factors impact (Experiment 2)

In this test, all factors that do impact on latencies of function invocations are combined. Besides, the other functions are used (Sect. 4.2) whose execution times are unknown in advance. ETAS has two main parts, prediction and scheduling. Hence, before performing the tests of this class, the prediction part is examined and the results are reported.

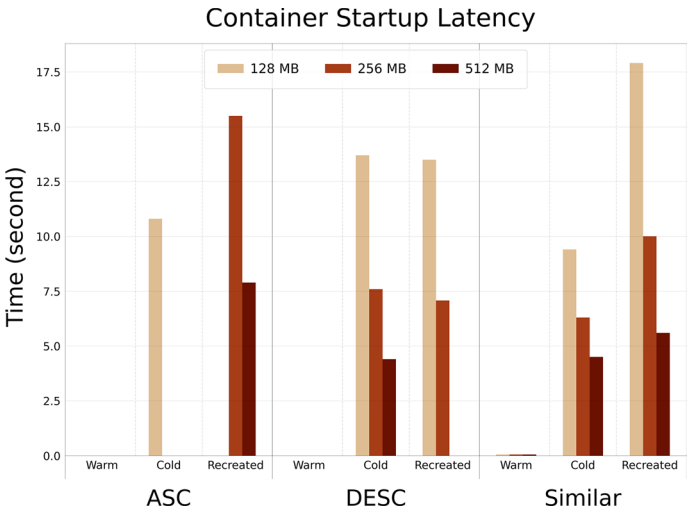


Fig. 9 Container startup latency in different states with different sizes of memory in three invocation patterns

According to Fig. 3, for each action and input, a unique key is considered to provide an accurate prediction. Figure 12 demonstrates the predicted and real execution times of a function in 10 invocations. Figure 12a is depicted to show the prediction in different resource settings (memory), and Fig. 12b shows the prediction with different inputs. It is clear that after second invocations, the execution time of a function with a specific resource setting and input will be predicted accurately. However, we assume that our functions are computational and with the same input almost have the same execution time. Although at first invocation, it is assumed that the execution times of the functions are 0, which is not the best choice and it is discussed in future work, it has no effect on our experiments since the results are discussed in stable states. Figures 12 and 13 are just provided to show the precision of the prediction.

Figure 13 illustrates the predicted and real execution times of two functions in 25 invocations in which they are invoked with different inputs. After 5 or 6 invocations, the most predicted times are close or precise to the actual execution times.

In this test, we use 3 users, 3 functions (*primeNumber that finds the n*th *prime number*, *factorial*, and *multiplyMatrix*) in Python, 3 resource settings (128 MB, 256 MB, and 512 MB). Therefore, 27 different actions are used. The average time intervals are 1, 2, and 3 s, and 6 different inputs for each function are used that produce different execution times (about 0.5, 2, 5, 10, 30, and 60 s). The number of requests is constant and equal to 50. As a result, this category includes three tests. Each test is repeated 30 times on each scheme. Table 3 shows the functions, inputs, their execution times, and invocations percentage in each time interval. The percentages are selected in order to provide an overload situation in the system.

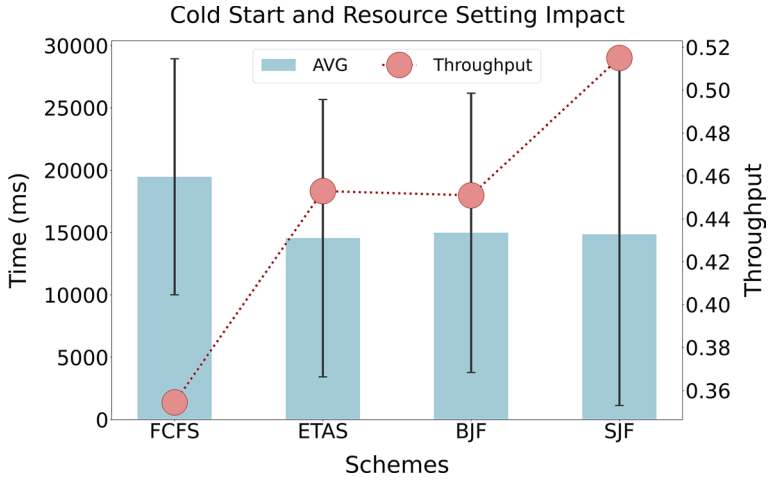
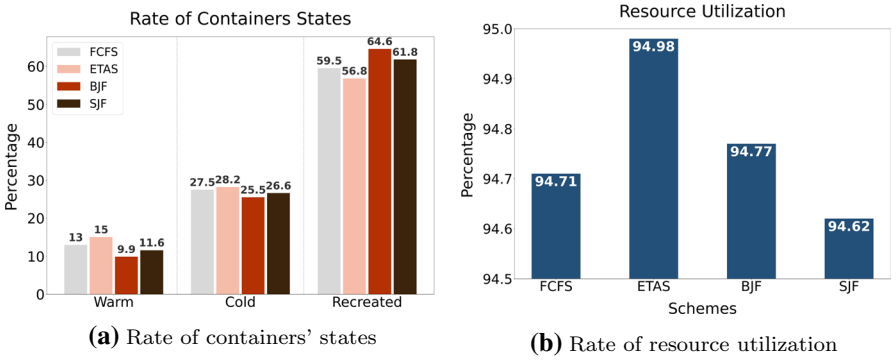


Fig. 10 Average waiting times and standard deviations for actions in the queue and throughput of the invoker in each scheme (average time intervals is 2 s, average execution times is 20 s, and the number of requests is 40)

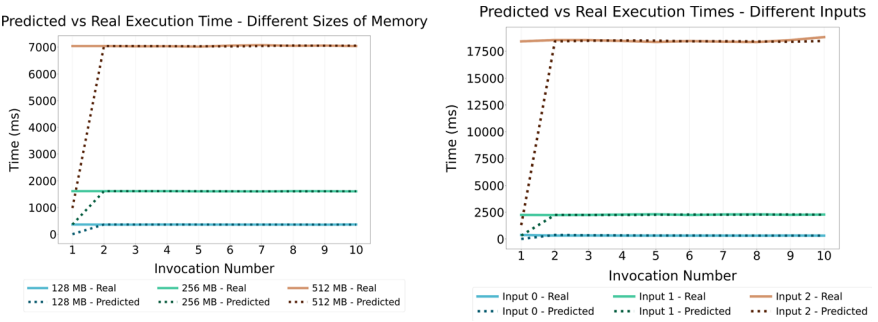


(a) Rate of containers' states **(b)** Rate of resource utilization

Fig. 11 Rates of containers' states and resource utilization in each scheme

Figure 14 illustrates the average execution times of the invoked functions in each test and each scheme. To provide a situation that functions are forced to stay in the queue, with an increase in the time interval the average execution times grow. However, with increases in the time intervals, the length of the queue experiences a slight reduction.

Figure 15 shows the average waiting times and standard deviations for actions in the queue and throughput of the invoker in different schemes and time intervals. FCFS has the lowest, and SJF has the highest throughput while ETAS and BJF are approximately equal in each time interval. We measure the throughput by computing



(a) The accuracy of the execution time prediction in different sizes of memory

(b) The accuracy of the execution time prediction using different inputs

Fig. 12 Comparing predicted and real execution times of a function in different sizes of memory and inputs in 10 invocations

the average number of actions that has been dequeued for execution per second. As we expected, FCFS has the most average waiting time. When the time interval is equal to 1 ($\theta = 1$), as it was discussed in Sect. 4.4.1 ETAS, BJF, and SJF behave similarly while with an increase in time interval SJF starves the functions with long execution times increasing the average waiting time and standard deviation.

We expected that with increases in the time intervals, the probability of reusing warm containers and resource utilization increases too (according to Sect. 3.4.2 an action can be replaced with a candidate function if it has arrived earlier), resulting in the reduction in average waiting times of actions in the queue and total latencies of functions. Although the results of Fig. 15 substantiate our assumption, Fig. 16 reveals the contrary.

Figure 16a shows the percentage of using containers with different states in each scheme and time interval. Obviously, in ETAS, reusing warm containers is higher than other schemes, while the total average of startup latencies (Fig. 16b) does not show any significant changes. On the other hand, almost FCFS experiences the least startup latencies. Despite the low delay in using warm containers, the impact of high latency in cold start and recreating containers, which varies in different circumstances (Fig. 9), overcomes the effect of the higher percentage of reusing warm containers. However, the lower container startup delay, when the time interval is equal to 3, leads to a lower average waiting time in ETAS in comparison with BJF. In other words, when $\theta = 3$, the starting delays of the containers (cold and recreated states) in all schemes are approximately equal, which leads to the manifestation of the effect of further reuse of warm containers.

To sum up, although it is expected that SJF has the shortest waiting time, it is clear that the container startup delay has a significant effect on the invocations waiting times. Moreover, with increase in time intervals, the average waiting times in SJF scheme grows as a result of starvation.

Figure 17 indicates the percentage of resource utilization in each scheme and the average time intervals. Although the increase in the percentage of resources in ETAS is not significant, the resource utilization has raised without imposing

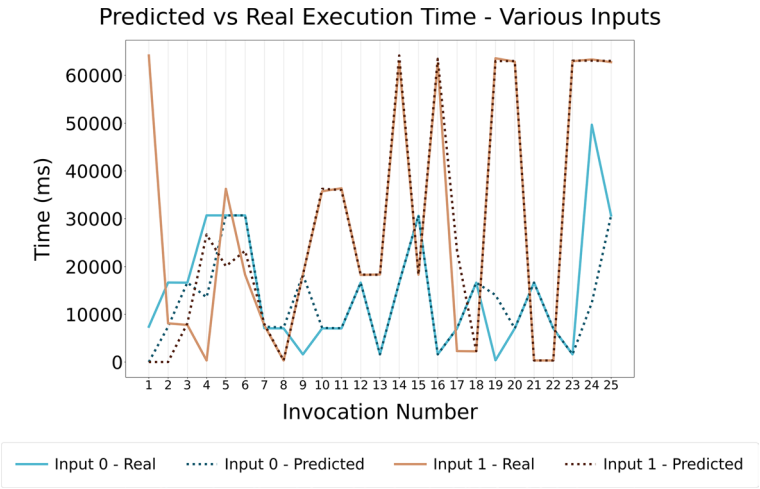


Fig. 13 Comparing predicted and real execution times of two different functions invoked using 6 different inputs in 25 invocations

Table 3 Input, execution times, and invocations percentage of benchmark's functions

Inputs	Functions			Execution Time (s)	Invocations Percentage		
	primeNumber	factorial	multiplyMatrix		$\Theta = 1(\%)$	$\Theta = 2(\%)$	$\Theta = 3(\%)$
1	1×10^3	5×10^4	1×10^2	0.5	15	10	5
2	2×10^3	10×10^4	2×10^2	2	25	15	10
3	4×10^3	15×10^4	3×10^2	5	25	25	20
4	6×10^3	20×10^4	4×10^2	10	20	25	25
5	8×10^3	25×10^4	5×10^2	30	10	20	25
6	10×10^3	30×10^4	$6 \times \times 10^2$	60	5	10	15

more latency on other actions in the queue (Fig. 15). Moreover, an increase in the resource utilization in all worker nodes (which we have not measured in this work) would be considerable.

Table 4 demonstrates the results of Figs. 14, 15, 16 and Fig. 17 to compare the parameters precisely. A glance at the chart reveals that (as we discussed before) despite increases in rate of reusing warm containers in ETAS, the containers startup latencies have not decreased. As it is depicted in Fig. 9, the delay has been affected by the memory, container image, and the recreated status (recreating from larger (base on memory) container to smaller one or reverse).

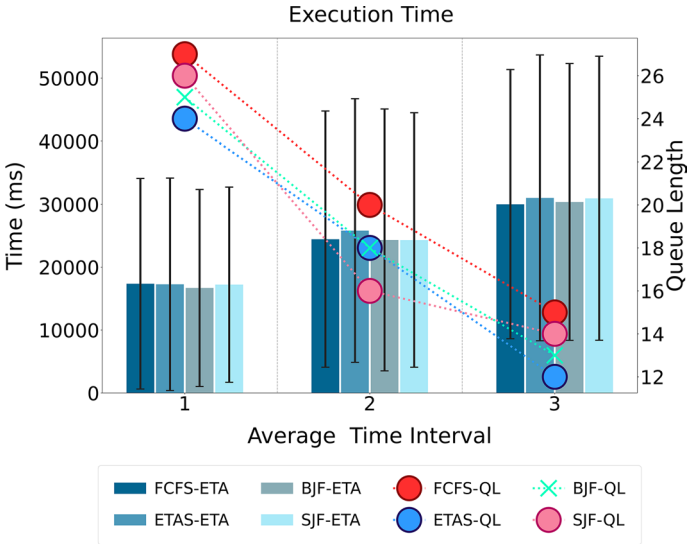


Fig. 14 Average and standard deviations of execution times of invoked functions with maximum queue length in each scheme with different time intervals. The number of requests is 50 in each iteration

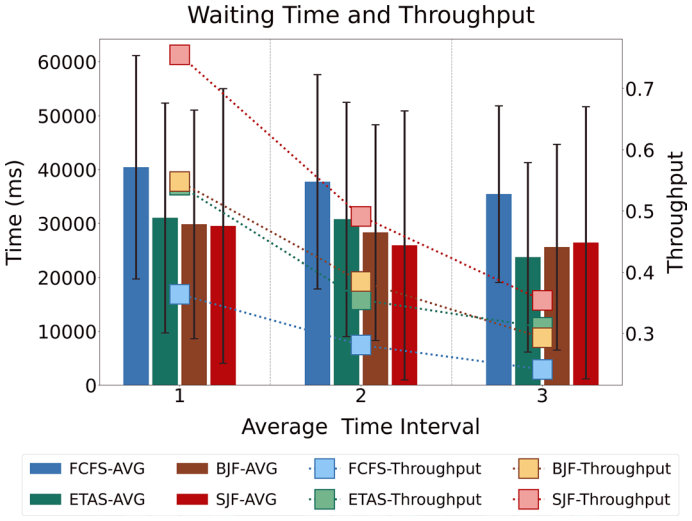
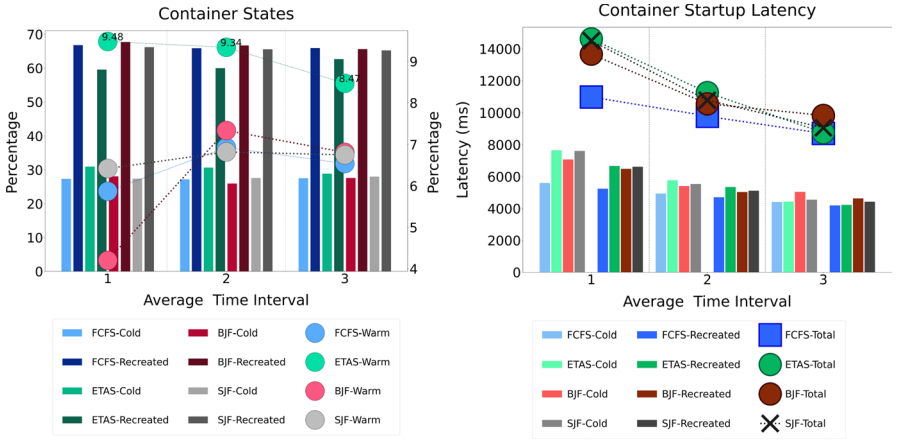


Fig. 15 Average waiting times and standard deviations for actions in the queue and throughput of the invoker in different time intervals and schemes. The number of requests is 50 in each iteration

4.4.3 Discussion

The goals of proposing ETAS are reducing the average waiting times of functions and increasing the throughput and resource utilization of the worker nodes of Apache OpenWhisk platform. To measure how much ETAS can accomplish its



(a) Percentage of using containers in different states

(b) Containers startup latency in each scheme and state

Fig. 16 Percentage of use of containers in different states and their startup latencies in different schemes and time intervals

purposes and explore its behavior along with other schemes (FCFS, BJF, SJF, and MQS), two experiments have been established. In experiment 1, the prediction part is ignored and various factors that have impact on the waiting times of functions are investigated, which are included time interval, execution time, workload, container's startup status. After that, the precision of prediction part is discussed and in experiment 2 all factors with prediction are measured and analyzed.

In summary, the outcome of our experimental results is as following:

- With increase in time intervals and execution times of functions and also load of requests, ETAS with preventing starvation leads to a lower waiting time and also a lower standard deviation.
- The results substantiated that MQS is not efficient when the jobs arrive sequentially. Our observations reveal that with fluctuation in the rate of invocations, the queues lose their balances, and distributing the functions among the queues imposes overhead on the system.

Cold start latency also varies from 1 to 20 s according to the type of the sandbox, the amount of memory that is dedicated to the sandbox, the requirements that a function demands (e.g., installing packages or library) or the function's language, and also the platform architecture [5, 29, 50, 52, 53, 57].

- According to our investigation, the startup latencies on our infrastructure and results reveal that the delay also depends on resources statuses. According to Fig. 9, replacing a container with 256 MB with 2 containers with 128 MB of

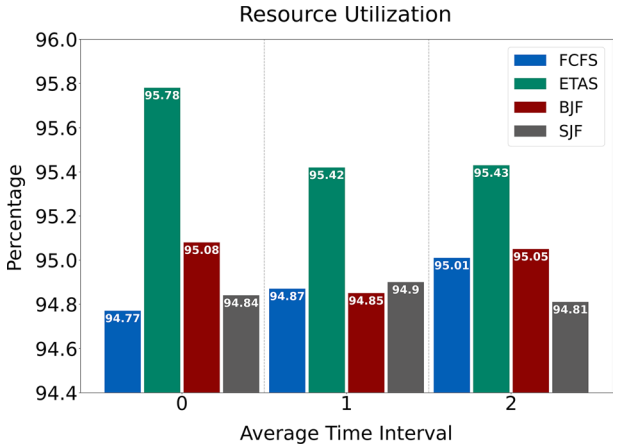


Fig. 17 Percentage of resource utilization in different schemes and time intervals

memory doubles the delay of substituting it for a container with 128 MB of memory.

- The percentage of reusing warm containers differs according to various workloads. Nevertheless, according to the experimental results, despite its slight influence, reusing warm containers not only decreases startup latencies, but also reduces the average waiting times of actions in the queue (Fig. 15, $\theta = 3$).
- Restricted memory allocation for functions in OpenWhisk (128MB, 256MB, and 512MB) affects resource utilization. Using these settings almost span the whole resource capacity. In other platforms such as AWS Lambda[7], users can set a value between 128MB and 1536MB in 64-MB increments [5] and this wide range of memory allocation would impact the results of resource utilization tests.
- Overall, although in different circumstances the superiority of ETAS varies, we can claim at best ETAS reduces the average waiting times of invocations up to 30% and increases the throughput of the worker node up to 40% in compare with FCFS, which is OpenWhisk default scheduler scheme. ETAS also has lower average waiting times in compare with SJF up to 10% in higher time intervals of requests by avoiding starving the functions with longer execution times. ETAS also reduces the average waiting times of function invocations up to 8% and increases the throughput up to 2% compared with the modified version of BJF.

5 Conclusion and future work

Serverless computing has become popular as a way to decouple backend infrastructure from users. This paradigm leads to complexity of allocating and provisioning resources to cloud providers. Autoscaling and resource provisioning without

[7] https://aws.amazon.com/lambda/

Table 4 Results of Experiment 2

Intervals	$\Theta = 1$				$\Theta = 2$				$\Theta = 3$			
Schemes	FCFS	ETAS	BJF	SJF	FCFS	ETAS	BJF	SJF	FCFS	ETAS	BJF	SJF
Average waiting times (ms)	40,436	31,040	29,849	29,530	37,736	30,777	28,321	25,941	35,459	23,724	25,603	26,432
Standard deviation	20,707	21,323	21,221	25,458	19,857	21,738	20,024	26,976	16,394	17,577	19,086	27,265
Average execution times (ms)	17,360	17,271	16,687	17,213	24,431	25,804	24,307	24,300	29,991	30,992	30,347	30,933
Throughput	0.364	0.542	0.548	0.755	0.281	0.355	0.385	0.491	0.241	0.31	0.293	0.354
Resource utilization (%)	94.77	95.78	95.08	94.84	94.87	95.42	94.85	94.9	95.01	95.43	95.05	94.81
Warm container usage (%)	5.87	9.48	4.2	6.42	6.93	9.34	7.34	6.82	6.54	8.47	6.81	6.75
Containers startup latencies (ms)	10975	14622	13664	14501	9784	11252	10552	10769	8709	8760	9819	9063

prediction of future load need smarter scheduling. The policy of using the least number of worker nodes to increase resource utilization, alongside burstiness of workloads, erratic request rates and wide range of execution times of serverless functions, can lead to high workloads on worker nodes and increase the response times of requests. To relieve this problem, we have focused on worker nodes' scheduling and proposed a predictive scheduling scheme, called ETAS (Execution-Time Aware Scheduling), for worker nodes of Apache OpenWhisk platform. The main objectives of ETAS were reduction in waiting times of functions and increasing the throughputs and resource utilizations of worker nodes. ETAS is a queue scheduler scheme, that schedules functions on each worker node by predicting the execution times of functions. The execution times of functions are estimated by logging pervious executions and the scheduler selects functions based on their arrival times and execution times to reduce the waiting times without starving them. ETAS has been implemented in the worker node component of Apache OpenWhisk and designed a benchmark to provide an evaluation. Moreover, to investigate the behavior of ETAS and provide an acceptable assessment other scheduling schemes (Shortest-Job-First, modified version of Best-Job-First, and Multi-Queue-Scheduling) have been implemented alongside the OpenWhisk default worker scheduler scheme, which is First-Come-First-Serve, to be able to compare ETAS with others. The experimental results demonstrate that ETAS outperformed other scheduling schemes and achieved all objectives that we had set for it. ETAS reduces the average waiting times of invocations up to 30% and increases the throughput of the worker node up to 40% in compare with FCFS and also has lower average waiting times in compare with SJF up to 10% in higher time intervals of requests by avoiding starvation. ETAS also reduces the average waiting times of function invocations up to 8% and increases the throughput up to 2% compared with the modified version of BJF.

ETAS has three main drawbacks that can be considered as future works. First, QoS is a notable feature in cloud computing that cannot be ignored. Unfortunately, ETAS has made no effort to observe QoS violations. Although ETAS reduces the standard deviation of waiting times of functions in order to establish fairness among functions, the lack of QoS consideration is evident. Second, there is no specific method for scheduling the functions that are invoked for the first time and their execution times are not predicted. Third, ETAS prediction based on previous executions is used just for the computational functions that did not need network or I/O. Therefore, a new mechanism is required to compute the execution times of these types of functions. ETAS is proposed as a local scheduler for worker nodes of Apache OpenWhisk and we assume that all workers are homogenous. For the next step, we are going to extend our scheme to other serverless platforms by tackling the aforenoted shortcomings. Moreover, by estimating the execution times of functions, smart coordination can be established between workers and dispatcher scheduling schemes to achieve more efficient scheduling.

Data Availability Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Jonas E, Pu Q, Venkataraman S, Stoica I, Recht B (2017) Occupy the cloud: distributed computing for the 99%. In: Proceedings of the 2017 symposium on cloud computing, pp 445–451
2. Jonas E, Schleier-Smith J, Sreekanti V, Tsai CC, Khandelwal A, Pu Q, Shankar V, Carreira J, Krauth K, Yadwadkar N et al (2019) Cloud programming simplified: a berkeley view on serverless computing. arXiv preprint arXiv:1902.03383
3. Hendrickson S, Sturdevant S, Harter T, Venkataramani V, Arpaci-Dusseau AC, Arpaci-Dusseau RH (2016) Serverless computation with openlambda, In: 8th {USENIX} workshop on hot topics in cloud computing (HotCloud 16)
4. Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A et al (2017) Serverless computing: current trends and open problems. In: Research advances in cloud computing. Springer, pp 1–20
5. Wang L, Li M, Zhang Y, Ristenpart T, Swift M (2018) Peeking behind the curtains of serverless platforms. In: 2018 {USENIX} Annual Technical Conference (USENIX, ATC, 18), pp 133–146
6. Castro P, Ishakian V, Muthusamy V, Slominski A (2019) The server is dead, long live the server: rise of serverless computing, overview of current state and future trends in research and industry. arXiv preprint arXiv:1906.02888
7. Adzic G, Chatley R (2017) Serverless computing: Economic and architectural impact. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 884–889
8. Shahrad M, Fonseca R, Goiri Í, Chaudhry G, Batum P, Cooke J, Laureano E, Tresness C, Russinovich M, Bianchini R (2020) Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. arXiv preprint arXiv:2003.03423
9. Hellerstein JM, Faleiro J, Gonzalez JE, Schleier-Smith J, Sreekanti V, Tumanov A, Wu C (2018) Serverless computing: one step forward, two steps back. arXiv preprint arXiv:1812.03651
10. Cordingly R, Shu W, Lloyd WJ (2020) Predicting performance and cost of serverless computing functions with saaf. In: 2020 IEEE International Conference on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech). IEEE, pp 640–649
11. Van Eyk E, Iosup A, Abad CL, Grohmann J, Eismann S (2018) A spec rg cloud group's vision on the performance challenges of faas cloud architectures. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pp 21–24
12. Shafiei H, Khonsari A, Mousavi P (2019) Serverless computing: a survey of opportunities, challenges and applications. arXiv preprint arXiv:1911.01296
13. Bhardwaj A, Gupta M, Stutsman R (2020) On the impact of isolation costs on locality-aware cloud scheduling, In: 12th {USENIX} workshop on hot topics in cloud computing (HotCloud 20)
14. Eismann S, Scheuner J, van Eyk E, Schwinger M, Grohmann J, Herbst N, Abad CL, Iosup A (2020) A review of serverless use cases and their characteristics. arXiv preprint arXiv:2008.11110
15. Singhvi A, Houck K, Balasubramanian A, Shaikh MD, Venkataraman S, Akella A (2019) Archipelago: a scalable low-latency serverless platform. arXiv preprint arXiv:1911.09849
16. Kaffes K, Yadwadkar NJ, Kozyrakis C (2019) Centralized core-granular scheduling for serverless functions. In: Proceedings of the ACM symposium on cloud computing, pp 158–164
17. Gunasekaran JR, Thinakaran P, Chidambaram N, Kandemir MT, Das CR (2020) Fifer: tackling underutilization in the serverless era. arXiv preprint arXiv:2008.12819
18. Kim YK, HoseinyFarahabady MR, Lee YC, Zomaya AY (2020) Automated fine-grained cpu cap control in serverless computing platform. IEEE Trans Parall Distrib Syst 31(10):2289–2301
19. Suresh A, Gandhi A Fnsched, (2019) An efficient scheduler for serverless functions. In: Proceedings of the 5th international workshop on serverless computing, pp 19–24
20. Stein M (2018) The serverless scheduling problem and noah. arXiv preprint arXiv:1809.06100

21. Suresh A, Somashekar G, Varadarajan A, Kakarla VR, Upadhyay H, Gandhi A (2020). Ensure: efficient scheduling and autonomous resource management in serverless environments. In: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), IEEE, pp 1–10
22. Shahrad M, Balkind J, Wentzlaff D (2019) Architectural implications of function-as-a-service computing. In: Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture, pp 1063–1075
23. Silberschatz A, Gagne G, Galvin PB (2018) Operating System Concepts, 10th edn. Wiley. https://www.wiley.com/en-us/Operating+System+Concepts%2C+10th+Edition-p-9781119320913
24. Kuntsevich A, Nasirifard P, Jacobsen HA (2018) A distributed analysis and benchmarking framework for apache openwhisk serverless platform. In: Proceedings of the 19th International Middleware Conference (Posters), pp 3–4
25. Al-Husainy MA (2007) Best-job-first cpu scheduling algorithm. Inform Technol J 6(2):288–293
26. Karthick A, Ramaraj E, Subramanian RG (2014) An efficient multi queue job scheduling for cloud computing. In: 2014 World congress on computing and communication technologies. IEEE, pp 164–166
27. Kratzke N (2018) A brief history of cloud application architectures. Appl Sci 8(8):1368
28. McGrath G, Brenner PR (2017) Serverless computing: design, implementation, and performance. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, pp 405–410
29. Li J, Kulkarni SG, Ramakrishnan K, Li D (2019) Understanding open source serverless platforms: Design considerations and performance. In: Proceedings of the 5th international workshop on serverless computing, pp 37–42
30. Van Eyk E, Scheuner J, Eismann S, Abad CL, Iosup A (2020) Beyond microbenchmarks: the spec-rg vision for a comprehensive serverless benchmark. In: Companion of the ACM/SPEC international conference on performance engineering, pp 26–31
31. Apache openwhisk documentation. https://openwhisk.apache.org/documentation.html
32. Shankar V, Krauth K, Pu Q, Jonas E, Venkataraman S, Stoica I, Recht B, Ragan-Kelley J (2018) Numpywren: serverless linear algebra. arXiv preprint arXiv:1810.09679
33. Aslanpour MS, Toosi AN, Cicconetti C, Javadi B, Sbarski P, Taibi D, Assuncao M, Gill SS, Gaire R, Dustdar S (2021) Serverless edge computing: vision and challenges. In: 2021 Australasian computer science week multiconference, pp 1–10
34. Fouladi S, Wahby RS, Shacklett B, Balasubramaniam KV, Zeng W, Bhalerao R, Sivaraman A, Porter G, Winstein K (2017) Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In: 14th {USENIX} Symposium on networked systems design and implementation (NSDI, 17), pp 363–376
35. Ao L, Izhikevich L, Voelker GM, Porter, G.: Sprocket, (2018) A serverless video processing framework. In: Proceedings of the ACM symposium on cloud computing, pp 263–274
36. Aditya P, Akkus IE, Beck A, Chen R, Hilt V, Rimac I, Satzke K, Stein M (2019) Will serverless computing revolutionize nfv? Proc IEEE 107(4):667–678
37. Ishakian V, Muthusamy V, Slominski A (2018). Serving deep learning models in a serverless platform. In: 2018 IEEE International Conference on Cloud Engineering (IC2E), IEEE, pp 257–262
38. Bhattacharjee A, Chhokra AD, Kang Z, Sun H, Gokhale A, Karsai, G (2019) Barista, (2019). Efficient and scalable serverless serving system for deep learning prediction services. In: 2019 IEEE International Conference on Cloud Engineering (IC2E), IEEE, pp 23–33
39. Jiang J, Gan S, Liu Y, Wang F, Alonso G, Klimovic A, Singla A, Wu W, Zhang C (2021) Towards demystifying serverless machine learning training. In: Proceedings of the 2021 International Conference on Management of Data, pp 857–871
40. Mahgoub A, Shankar K, Mitra S, Klimovic A, Chaterji S, Bagchi S (2021) Sonic: application-aware data passing for chained serverless applications. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21). USENIX Association, Virtual (forthcoming)
41. Zhang H, Tang Y, Khandelwal A, Chen J, Stoica I (2021) Caerus: Nimble task scheduling for serverless analytics. In: Proceedings of the 18th USENIX symposium on networked systems design and implementation, pp 653–669
42. Prakash V, Bawa S, Garg L (2021) Multi-dependency and time based resource scheduling algorithm for scientific applications in cloud computing. Electronics 10(11):1320
43. Shillaker S, Pietzuch P (2020) Faasm: Lightweight isolation for efficient stateful serverless computing. arXiv preprint arXiv:2002.09344

44. Sang B, Roman PL, Eugster P, Lu H, Ravi S, Petri G (2020) Plasma: programmable elasticity for stateful cloud computing applications. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp 1–15
45. Sreekanti V, Lin C.W.X.C, Faleiro JM, Gonzalez JE, Hellerstein JM, Tumanov A (2020) Cloudburst: Stateful functions-as-a-service. arXiv preprint arXiv:2001.04592
46. Maissen P, Felber P, Kropf P, Schiavoni V (2020) Faasdom: a benchmark suite for serverless computing. arXiv preprint arXiv:2006.03271
47. Kim J, Lee, K.: Functionbench, (2019). A suite of workloads for serverless cloud function service. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, pp 502–504
48. Pellegrini R, Ivkic I, Tauber M (2019) Function-as-a-service benchmarking framework. arXiv preprint arXiv:1905.11707
49. Spillner J (2017) Snafu: function-as-a-service (faas) runtime design and implementation. arXiv preprint arXiv:1703.07562
50. Akkus IE, Chen R, Rimac I, Stein M, Satzke K, Beck A, Aditya P, Hilt V (2018) Sand: towards high-performance serverless computing. In: 2018 {USENIX} Annual Technical Conference (USENIX, ATC, 18), pp 923–935
51. Chard R, Skluzacek TJ, Li Z, Babuji Y, Woodard A, Blaiszik B, Tuecke S, Foster I, Chard K (2019) Serverless supercomputing: high performance function as a service for science. arXiv preprint arXiv:1908.04907
52. Lloyd W, Ramesh S, Chinthalapati S, Ly L, Pallickara S (2018) Serverless computing: an investigation of factors influencing microservice performance. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). IEEE, pp 159–169
53. Oakes E, Yang L, Zhou D, Houck K, Harter T, Arpaci-Dusseau A, Arpaci-Dusseau R (2018) Sock: rapid task provisioning with serverless-optimized containers. In: 2018 {USENIX} Annual Technical Conference (USENIX, ATC, 18), pp 57–70
54. Fingler H, Akshintala A, Rossbach, C.J Usetl, (2019) Unikernels for serverless extract transform and load why should you settle for less? In: Proceedings of the 10th ACM SIGOPS Asia-Pacific workshop on systems, pp 23–30
55. Agache A, Brooker M, Iordache A, Liguori A, Neugebauer R, Piwonka P, Popa DM (2020) Firecracker: lightweight virtualization for serverless applications. In: 17th {USENIX} symposium on networked systems design and implementation (NSDI, 20), pp 419–434
56. Bermbach D, Karakaya AS, Buchholz S (2020) Using application knowledge to reduce cold starts in faas services. In: Proceedings of the 35th annual ACM symposium on applied computing, pp 134–143
57. Oakes E, Yang L, Houck K, Harter T, Arpaci-Dusseau AC, Arpaci-Dusseau, RH (2017) Pipsqueak: lean lambdas with large libraries. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, pp 395–400
58. Xu Z, Zhang H, Geng X, Wu Q, Ma H (2019) Adaptive function launching acceleration in serverless computing platforms. In: 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS). IEEE, pp 9–16
59. Aumala G, Boza E, Ortiz-Avilés L, Totoy G, Abad C (2019) Beyond load balancing: package-aware scheduling for serverless platforms. In: 2019 19th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID). IEEE, pp 282–291
60. De Palma G, Giallorenzo S, Mauro J, Zavattaro G (2020) Allocation priority policies for serverless function-execution scheduling optimisation. In: International Conference on Service-Oriented Computing. Springer, pp 416–430
61. Mu'alem AW, Feitelson DG (2001) Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Trans Parall Distrib Syst 12(6):529–543
62. Agerwala T, Martin JL, Mirza JH, Sadler DC, Dias DM, Snir M (1995) Sp2 system architecture. IBM Syst J 34(2):414–446
63. Yu J, Buyya R (2005) A taxonomy of workflow management systems for grid computing. J Grid Comput 3(3–4):171–200
64. Gibbons R (1997) A historical application profiler for use by parallel schedulers. In: Workshop on job scheduling strategies for parallel processing. Springer, pp 58–77
65. Smith W, Foster I, Taylor V (1998) Predicting application run times using historical information. In: Workshop on job scheduling strategies for parallel processing. Springer, pp 122–142
66. Miu T, Missier P (2012) Predicting the execution time of workflow activities based on their input features. In: 2012 SC companion: high performance computing, networking storage and analysis. IEEE, pp 64–72

67. Chirkin AM, Belloum AS, Kovalchuk SV, Makkes MX, Melnik MA, Visheratin AA, Nasonov DA (2017) Execution time estimation for workflow scheduling. Future Gen Comput Syst 75:376–387