POLITECNICO DI TORINO
Repository ISTITUZIONALE

Dynamic Management of Constrained Computing Resources for Serverless Services

Original
Dynamic Management of Constrained Computing Resources for Serverless Services / Adeppady, Madhura; Conte, Alberto; Giaccone, Paolo; Karl, Holger; Chiasserini, Carla Fabiana. - In: IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT. - ISSN 1932-4537. - STAMPA. - (2025). [10.1109/TNSM.2024.3497155]

Availability:
This version is available at: 11583/2994279 since: 2024-11-15T08:12:16Z

Publisher:
IEEE

Published
DOI:10.1109/TNSM.2024.3497155

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

21 November 2024

Dynamic Management of Constrained Computing Resources for Serverless Services

Madhura Adeppady, *Student Member, IEEE*, Alberto Conte, *Member, IEEE*,
Paolo Giaccone, *Senior Member, IEEE*, Holger Karl, *Member, IEEE*, Carla Fabiana Chiasserini, *Fellow, IEEE*

***Abstract*—In resource-constrained cloud systems, e.g., at the network edge or in private clouds, serverless computing is increasingly adopted to deploy microservices-based applications, leveraging its promised high resource efficiency. Provisioning resources to serverless services, however, poses several challenges, due to the high cold-start latency of containers and stringent Service Level Agreement (SLA) requirements of the microservices. In response, we investigate the behavior of containers in different states (i.e., running, warm, or cold) and exploit our experimental observations to formulate an optimization problem that minimizes the energy consumption of the active servers while reducing SLA violations. In light of the problem complexity, we propose a low-complexity algorithm, named AiW, which utilizes a multi-queueing approach to balance energy consumption and system performance by reusing containers effectively and invoking cold-starts only when necessary. To further minimize the energy consumption of data centers, we introduce the two-timescale COmputing resource Management at the Edge (COME) framework, comprising an orchestrator running our proposed AiW algorithm for container provisioning and Dynamic Server Provisioner (DSP) for dynamically activating/deactivating servers in response to AiW's decisions on request scheduling. COME addresses the mismatch in timescales for resource provisioning decisions at the container and server levels. Extensive performance evaluation through simulation shows AiW's close match to the optimum and COME's significant reduction in power consumption by 22–64% compared state-of-the-art alternatives.**

***Index Terms*—Microservices, Serverless Edge Computing, Container Retention, Energy consumption**

I. Introduction

Edge computing serves many real-time computational tasks, reducing the consumption of bandwidth towards the cloud as well as end-to-end latency [2]. Nonetheless, widespread deployment of edge computing is still challenging [3]. From the system perspective, provisioning computing resources at the granularity of virtual machines, as done in conventional cloud computing, brings in long provisional delays and wastes resources, which is unacceptable for resource-constrained edge servers and time-critical services. Serverless edge computing, with its Function-as-a-Service (FaaS) offer, redefines the way of deploying services [4], as it enables the decomposition of their logic into microservices (MS) and efficiently uses underlying resources [5], [6]. These resource constraints and latency requirements are particularly relevant in edge computing, but they can also apply to other environments, such as private clouds and small-scale data centers.

A key point is that, in serverless edge computing, MSs run inside containers only when requested. Thus, serving a request involves creating a new container with appropriate runtime, which may involve downloading the necessary image from a remote repository, and fetching and loading essential libraries and dependencies before executing the actual function. This process is known as *cold-start* and the long delay involved in the initialization setup is referred to as startup latency, which is one of the main performance issues faced by serverless computing platforms [7]–[9]. A *warm* container, instead, keeps the MS instance alive in the memory, with a negligible startup latency when the warm container is reused for serving a later request for the same MS. Due to the limited memory at the edge nodes, serving all the requests with warm containers is practically impossible. Further, keeping too many warm containers in memory reduces resource utilization.

Existing issues and research gaps. Many research efforts have focused on minimizing cold-starts by optimizing the keep-alive time of warm containers [7]–[9]. An ideal keep-alive time must tradeoff the resource overhead of warm containers with their reusability in the near future. Since MSs have widely varying resource needs, invocation frequencies, and startup overhead, it is challenging to determine an ideal keep-alive time. Other recent approaches focus on utilizing specialized sandbox mechanisms like unikernels [10], MicroVMs [11], and other virtualization techniques [12] to reduce the cold-start latency. Although these sandbox mechanisms use fewer resources than warm containers, they still suffer from resource overheads. Further, they either allocate resources to the containers by largely overlooking MS-specific Quality of Service (QoS) requirements, e.g., a given target delay, or CPU allocations are adjusted proportionally to the memory demands of the MSs [13]. This QoS-agnostic resource allocation might result in severe Service Level Agreement (SLA) violations, particularly for edge services that have stringent QoS requirements.

Scientific challenges. Given the above issues and research gaps, it is critical to face the cold-start problem in serverless edge computing with the twofold aim of (i) ensuring the level of QoS required by the MSs offered to users, and (ii) reducing edge data centers' energy footprint. To do so, we draw on the following key observations.

Observation 1. As shown later by our experiments, cold-start latency can contribute up to 81% of the total service

M. Adeppady, P. Giaccone, and C. F. Chiasserini are with the Electronics and Telecommunications Dept., Politecnico di Torino, Italy, and with CNIT, Parma, Italy. Email: {firstname.lastname@polito.it}. C. F. Chiasserini is also with Chalmers University of Technology, Sweden. A. Conte is with the Nokia bell labs, Nozay, France. Email: alberto.conte@nokia-bell-labs.com. H. Karl is with the Hasso Plattner Institute, University of Potsdam, Germany. Email: holger.karl@hpi.de.

This is an extended version of our IEEE GLOBECOM 2023 paper [1].

time for MS requests. Thus, for MSs with stringent delay constraints, serving the requests using a cold container may be impossible, necessitating warm containers. Serving however all requests at the edge with warm containers may also be impractical due to the limited memory available at the edge servers.

Observation 2. It is well known that edge data centers consume a significant amount of energy, and, since their energy consumption mainly depends on their CPU load, the containers executing on these servers, in turn, determine their energy consumption. Notice that, even if feasible, using cold containers to serve MS requests while meeting their target latency may require a high CPU speed (hence energy consumption), as the long startup latency must be compensated for by a reduced execution time. In contrast, using warm containers may be more energy-efficient because their negligible startup latency allows a lower CPU speed allocation.

Observation 3. It is well known that, even when servers are idle but still turned on, they consume a non-trivial amount of energy [14]. Therefore, it is important to turn them off whenever possible, without compromising QoS requirements of MSs.

Observation 4. MSs have short lifespans, leading to the need for dynamic toggling of containers between warm/cold and running states based on real-time service demands. Consequently, this may result in idle servers or necessitate additional servers at short timescales. On the other hand, decisions on activating/deactivating the servers act on a comparatively longer timescale.

Our contribution. Motivated by the above observations, we introduce Always in Warm (AiW), which utilizes a *multi-queueing system* for efficient container provisioning, while minimizing SLA violations. Leveraging one queue per MS type allows a higher *reusability* of MSs containers and enables prompt cold-starts only if necessary. Additionally, the multi-queueing system is motivated by the fact that warm containers of one MS cannot be reused to serve the requests for other MSs, as code and data of the former MS continues to exist in a warm container [15]. In more detail, AiW increases the container reusability by queueing the requests based on the residual processing time of the running containers and efficiently deciding the number of warm containers based on the current system load. Further, AiW's ability to effectively reuse containers leads to *lower CPU allocations*, and eventually to *reduced energy consumption.*

Finally, we propose *COmputing resource Management at the Edge (COME), a two-timescale framework for deploying serverless MSs.* COME reduces the energy footprint of edge data centers by optimizing energy consumption at *both the server and edge data center levels*, while *minimizing the potential SLA violations.* The framework consists of two modules, (i) an orchestrator running AiW algorithm for deploying and managing containers in active servers promptly and efficiently at a *short timescale*, and (ii) the *Dynamic Server Provisioner (DSP)* that further minimizes the energy consumption by dynamically activating/deactivating the servers in response to the orchestrator's decisions at comparatively a *longer timescale*. Specifically, the DSP module turns on a new server when the average load of the servers that are currently active exceeds a predefined threshold. An idle server is instead turned off only if this action keeps the average load across the active servers below threshold. To the best of our knowledge, COME is the first framework that allows reducing the energy usage at both server and edge data center levels for serverless services.

To summarize, our main contributions are as follows.

- We first present experimental evidence, conducted on the OpenWhisk [16] testbed, highlighting the crucial need to minimize the cold-start latency of MSs in resource-constrained edge environments. As mentioned, our findings indicate that cold-start latency can contribute very substantially to the total service time for requests, significantly impairing the responsiveness of the MSs.
- Since decisions made on container and server activations/deactivations occur at distinct timescales, we propose COME, a novel two-timescale approach to minimize the energy footprint of the edge data centers.
- Through a detailed, yet tractable, model of the system, we formulate an optimization problem for short timescale decisions that, looking at a finite time horizon, minimizes the servers' energy consumption by leveraging cold, warm, and running containers.
- Since the problem turns out to be NP-hard, we investigate AiW, a low-complexity, yet highly effective, multi-queueing solution that closely matches the optimum.
- For long timescale operations, we introduce DSP, a mechanism that activates/deactivates servers according to the average CPU load at the edge data center.
- Finally, through an extensive performance analysis based on simulations, we demonstrate that COME minimizes the power consumption of the edge data center by 22–64% compared to state-of-the-art schemes.

Paper organization. Sec. II provides experimental evidences on how cold-start latency impacts the responsiveness of MSs. Sec. III gives an overview of the proposed COME framework, while Sec. IV introduces the system model. Sec. V describes the problem formulation for short timescale decisions; owing to the problem complexity, Sec. VI and Sec. VII present, respectively, our low-complexity AiW orchestration solution and DSP algorithm for handling the data center servers. We highlight the improvements of COME against state-of-the-art alternatives in Sec. VIII. Finally, Sec. IX summarizes related work and highlights our novel contributions, while Sec. X concludes the paper.

II. Experimental Evidence and Work Motivation

To characterize the startup latency and resource overhead of cold and warm containers, we consider a few MSs from the faas-profiler [17] and serverless-faas-workbench [18]. The description of these MSs, along with their input parameters, is given in Table I. We conducted experiments on an Apache OpenWhisk setup running on a Single Intel Xeon server, which hosts both the framework and the MS instances in Docker containers.

TABLE I
LIST OF MSS: DESCRIPTION AND INPUT PARAMETERS

MSs	Description
`float operation`	finds sin, cos, and sqrt of numbers up to 10^7
`xml`	renders an HTML table of dimension 1000×1000
`matmul`	multiplies two 1000-dimensional square matrices
`img-resize`	resizes given image to several icons
`img-ocr`	finds text in the given image using Tesseract OCR
`encry`	performs private key encryption and decryption on a string of length 2,500 using 100 iterations

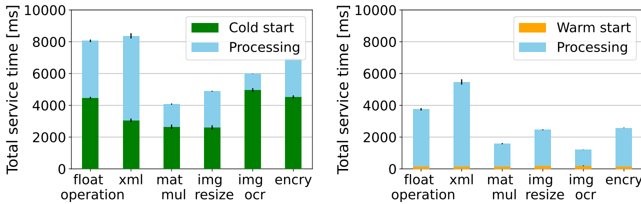

Fig. 1. Breakdown of total service time for cold-start (left) and warm-start (right).

A. Service time

The total service time of a MS request consists of i) cold/warm-startup latency and ii) processing time. Cold-start latency is measured for the initial MS request when no warm container exists. After the execution, OpenWhisk keeps the container in the warm state to serve future requests faster. Thus, warm-start latency is measured by sending another request for the same MS. Finally, we also measure the processing time, which is the actual time taken to execute the request; note that the processing time depends on the input parameters in the MS request. All latency and execution time data are extracted from OpenWhisk logs; average cold and warm-start latencies are computed with a 95% confidence interval.

Fig. 1(left) shows the obtained results, highlighting that the cold-start latency of the MSs is indeed significant, ranging between 2.5 and 5 seconds. For some MSs, the cold-start latency is even higher than the request processing time, accounting for up to 81% of the total service time. On the contrary, Fig. 1(right) reveals that warm-start latency for the considered MSs is always below 100 ms, making them negligible.

B. Resource overhead

In our experiments, after executing a request and before the arrival of the next one for the same MS, the container remains in the warm state. During this time, we measured the memory occupied by the warm container using `docker stats` command. The memory occupied by the warm containers of the various MSs is non-negligible, as shown in Table II: it varies from 9 MB for `float operation` up to 68 MB for `img-ocr`.

Note that uncontrolled swapping occurs when the memory used by the containers exceeds the server's capacity, harming performance. We avoid this by adding a constraint on the

TABLE II
MEMORY OCCUPIED BY WARM CONTAINERS FOR VARIOUS MSS

MSs	Warm container memory [MB]
`float operation`	8.4
`xml`	25.8
`matmul`	31.3
`img-resize`	23.3
`img-ocr`	68.9
`encry`	18.7

available memory at the server such that we never saturate the memory.

In summary, our experiments indicate that cold-start latency negatively impacts the responsiveness of the MSs. Thus, *meeting strict latency requirements* of MSs with cold containers demands shorter processing times, requiring high CPU speed and consequently increasing *overall energy consumption.* On the contrary, *warm containers* play a crucial role in significantly reducing the *energy footprint of edge servers* by efficiently handling requests with low CPU allocations.

III. THE COME FRAMEWORK

This section introduces COME, a framework for serverless services that minimizes the energy consumption at the server and edge data center levels while reducing SLA violations.

A. Architecture

To cope with the time mismatch between server activation/deactivation and MS lifespan, COME is designed to operate with a two-timescale approach. Accordingly, it comprises an orchestrator and a DSP module (see Fig. 2). The orchestrator handles individual queues of MSs by scheduling pending requests on containers that run on servers, while a DSP module at the edge data center level decides on the server activation/deactivation.

The orchestrator (bottom of Fig. 2) handles two events in the system: i) the arrival of a new request for an MS; ii) the completion of a request execution on a container. To effectively process these two events, the orchestrator maintains a dedicated queue for each MS, as containers dedicated to an MS cannot handle the requests for other MSs. Incoming requests wait in their respective MS queues, enabling the reuse of already running containers to process them within target delay. Thus, while processing an MS events, it is sufficient to focus on the queue and containers of that specific MS.

Note that, since the workload requirement (in terms of total CPU cycles) of MSs is already known and the orchestrator allocates CPU cycles/s for the containers to serve the requests, the request execution time is deterministic. Thus, the orchestrator knows residual processing times of those containers to which requests were assigned. On the other hand, new request arrivals occur randomly, and the orchestrator lacks prior knowledge of these arrivals.

DSP module (top of Fig. 2) monitors the average load across active servers after the orchestrator handles arrival and execution end events. It keeps track of active, inactive, and active-but-idle servers in the system and takes activation or deactivation actions on these servers whenever necessary.

B. High-level workflow

Upon the arrival of a new request for an MS, the orchestrator makes a scheduling decision on it. An overview of the actions that the orchestrator can take for handling a new request is given in Fig. 2. Using a warm container to start serving the request right away is the best-case scenario, since no queueing delay or cold-start latency is experienced (top actions of request arrival event on Fig. 2). If warm container is unavailable, the orchestrator decides between enqueuing this request (middle action of request arrival event on Fig. 2) or initiating a cold start for the newly arrived request (top actions of request arrival event on Fig. 2) based on the queued requests and existing containers of the MS. Intuitively, the idea of enqueueing a request is to foster the reuse of containers and reduce resource consumption. Nevertheless, cold start is preferred if enqueuing results in a very high queueing delay, eventually necessitating a very high CPU speed allocation. If none of these decisions is possible, the target delay cannot be met and the newly arrived request is dropped as described in bottom action of request arrival event on Fig. 2. A CPU speed is allocated to the container running a request, hereby impacting the overall energy consumption of the servers.

Similarly, when a container completes serving a request for an MS (request execution end event), the orchestrator determines whether to reuse the container for serving the Head of Line (HoL) request from that MS's queue. It is possible that container reuse is not feasible due to insufficient computing or memory resources on the server. In such cases, the orchestrator checks for a warm start on other active servers, provided they have warm containers available. If no warm start is feasible on any active server, the orchestrator then explores the possibility of a cold start. If neither warm nor cold starts are possible, the HoL request is dropped. All these actions are highlighted in top actions of request execution end event in Fig. 2.

As before, decisions on request scheduling impact CPU speed allocations to the containers and, consequently, energy consumption. Finally, keeping too many warm containers in memory causes resource overhead and unpredictable performance in case of paging out. The orchestrator evaluates the required number of warm containers by considering the current system load and arrivals observed in the past. It then decides to transit the warm containers to the cold state if there are more warm containers than necessary (bottom actions of request execution end event on Fig. 2).

DSP monitors the CPU load across the active servers whenever the orchestrator deals with (i) a new request arrival or (ii) an existing container finishing its execution events. As described in Fig. 2, the DSP module monitors the system (bottom action of DSP on Fig. 2) only during the arrival of new requests or the completion of execution events for existing containers because CPU load changes during these events. DSP activates a new server if the average load on the currently active servers exceeds the pre-defined threshold, and it deactivates an active idle server if there are enough resources to serve the requests (top actions of DSP on Fig. 2). Conversely, if the average load on the active servers drops below threshold, the DSP goes through the set of servers searching for an idle server and deactivates it, only if deactivation maintains the average load below the threshold.

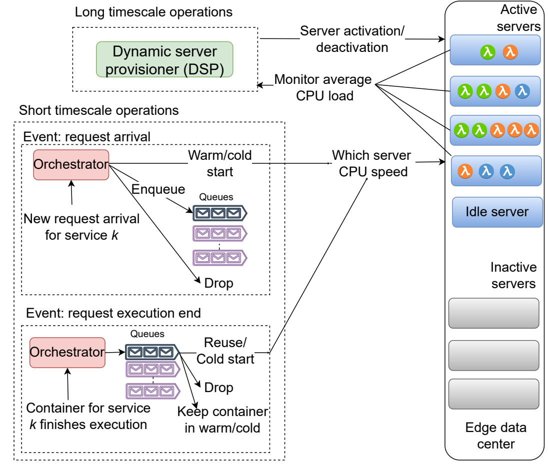

Fig. 2. Structure of COME framework.

IV. System Model

A service orchestrator may receive requests for any MS $k \in \mathcal{K}$, and serves them using the serverless computing paradigm. Any request for MS k demands certain amount of memory and workload (in CPU cycles), denoted by τ_k and w_k, respectively, and has a target maximum delay D_k in terms of time lapse from when the request arrives till the service execution is completed. We focus on a single data center and let $\mathcal{S}$ be the set of servers available therein, with $s \in \mathcal{S}$ having $\hat{\tau}_s$ bytes of memory and $\hat{\mu}_s$ CPU capacity (expressed in cycles/s).

A container can be in any of the three states: running (R), warm (W), cold (C), or in any of these two transition states: from C to R ($T_{C\to R}$), or from R to C ($T_{R\to C}$). Notice that we consider only $T_{C\to R}$ and $T_{R\to C}$, since the transition to/from W involves negligible startup latency. For a container destined to implement service k, let $\delta_{k,T_{C\to R}}$ and $\delta_{k,T_{R\to C}}$ denote the startup latency of cold-start and transition time from R to C, respectively. We remark that, as shown by our experiments on `float operation` MS, $\delta_{k,T_{C\to R}}$ and $\delta_{k,T_{R\to C}}$ are not equal. Specifically, for such an MS, we observed that $\delta_{k,T_{R\to C}}{=}1.5$ s while $\delta_{k,T_{C\to R}}{=}4.5$ s; recent studies [9], [19] corroborate this observation.

Let $c_{k,i,R}(t)$ be the container in state R serving the i-th request for MS k, $r_{k,i}$, at time t. When starting to serve $r_{k,i}$, the orchestrator assigns to container $c_{k,i,R}$ a CPU speed (in cycles/s), denoted by $\mu_{c_{k,i,R}}$, such that (i) the overall CPU capacity at the server is not exceeded, and (ii) $r_{k,i}$ is served within D_k. Additionally, the memory assigned to any container implementing MS k in state R is equal to the memory demand of an instance of MS k. We stress that a container in warm state at time t that implements MS k, $c_{k,W}(t)$, consumes only memory $\tau_{k,W}$. Instead, when in cold state, a container consumes neither CPU nor memory. Further, let $c_{k,i,T_{C\to R}}(t)$ (or, $c_{k,i,T_{R\to C}}(t)$) be the container in transition from C to R to serve $r_{k,i}$ at t (or, from R to C as it just finished serving $r_{k,i}$). A container of MS k in any of the above mentioned

transition states consumes both memory and CPU cycles/s. Let the CPU cycles/s and memory consumed by a container of MS k in the transition state $T_{C\to R}$ be $\mu_{k,T_{C\to R}}$ and $\tau_{k,T_{C\to R}}$, respectively. Similarly, for $T_{R\to C}$, the CPU cycles/s and memory consumption are denoted by $\mu_{k,T_{R\to C}}$ and $\tau_{k,T_{R\to C}}$, respectively. Since the container cleanup is much simpler than its startup [9], the CPU and memory consumption of the two transition states are not the same. To verify this, we performed some experiments on `float operation` and quantified the CPU cycles/s and memory consumption using perf [20] and vmstat [21] tools, respectively. Our measurements confirmed the above intuition: we found that total CPU cycles used by $T_{R\to C}$ is 0.9 G clock cycles $<$ 7 G clock cycles of $T_{C\to R}$, and $\tau_{k,T_{R\to C}} = 30\,\text{MB} < \tau_{k,T_{C\to R}}{=}55\,\text{MB}$.

Incoming requests for MS $k{\in}\mathcal{K}$ are enqueued on the corresponding dedicated queue, denoted by Q_k. When a request is assigned to a cold container, it must wait for the cold-to-running transition to complete before being served. To handle such cases, each MS has an (additional) *auxiliary queue* where the request assigned to a transiting container waits for the transition to finish.

Serverless computing typically use three operation modes: scale-per-request, concurrency-value-scaling, and metric-based-scaling [22]. In scale-per-request approach, each new request is served in a warm or cold container and after the request is served, the container is either kept in warm or transitioned back to cold state. Concurrency-value-scaling allows a single container to handle multiple requests simultaneously, based on the defined concurrency level. Metric-based-scaling maintains CPU or memory usage within defined ranges. Our system model uses the scale-per-request pattern, due to its widespread adoption in popular serverless platforms like AWS Lambda, Azure Functions, OpenWhisk, and their edge variants such as Lambda@Edge and Lean OpenWhisk.

The orchestrator makes a decision regarding MS k only when either of these two events occur: (i) a new request for MS k arrives; (ii) container $c_{k,i,R}(t)$ finishes serving request $r_{k,i}$ (at any point in time, there could be multiple containers of service k in state R). Another possible event is a container finishing its transition, upon which the orchestrator does not make any decision. If a container finishes a transition to the cold state, it is destroyed. Otherwise, if $c_{k,i,T_{C\to R}}(t)$ finishes its transition to become $c_{k,i,R}(t)$, the request $r_{k,i}$ is removed from the MS's auxiliary queue to get executed on it.

Also, the orchestrator acts upon each queue Q_k according to a FIFO policy, by serving the requests on containers implementing that specific MS. For serving HoL request in the queue, we have two cases. In the first one, we consider that no warm container is available. Hence, the orchestrator creates a new container and, once the container reaches state R, HoL request is served on it. In the second case, we consider that a warm container implementing the requested MS is available. In such a situation, the orchestrator can serve the request in a warm or cold container. For the former, the request is immediately removed from the queue and executed on the warm container. For the latter, the request is removed from the queue of waiting requests and assigned to the MS's auxiliary queue, where it waits for the container to finish the transition.

Let t be the time at which any of the previously mentioned events (arrival of a new request, container finishes serving a request, or container completes the transition) occurs. We denote by $\Omega^s_{k,W}(t)$, $\Omega^s_{k,R}(t)$, $\Omega^s_{k,T_{C\to R}}(t)$, and $\Omega^s_{k,T_{R\to C}}(t)$ the set of all containers of MS k on server s that at time t are in state W, R, and $T_{C\to R}$, and $T_{R\to C}$, respectively. We define $\Omega_{k,R}(t){=}\cup_{s\in\mathcal{S}}\,\Omega^s_{k,R}(t)$, and $\Omega_{k,T}(t){=}\cup_{s\in\mathcal{S}}\,\Omega^s_{k,T}(t)$ as the set of running and transiting containers across all the servers, respectively.

Processing time $T_{k,i}$ of request $r_{k,i}$ running on container $c_{k,i,R}$ is given by $T_{k,i}{=}w_k/\mu_{c_{k,i,R}}$ where $\mu_{c_{k,i,R}}$ is the CPU speed allocated to $c_{k,i,R}$ to process $r_{k,i}$ (note that the expression holds independently from the concurrent requests). The total service time of $r_{k,i}$ is the sum of the queueing delay before being served, eventually including the startup latency of the container on which the request is scheduled to run, and the processing time.

The power consumption of a server is composed of (i) a constant, idle power consumption when a server is active, P_{idle}, and (ii) a function $P(\cdot)$ of the server's actual CPU load, which, according to experimental results [14], is often assumed to be linear. Thus, the power consumption of server s at t with CPU load $\lambda_{s,t}$ is given by $P_s{=}\gamma_{s,t}\cdot P_{\text{idle}}{+}P(\lambda_{s,t})$, where $\gamma_{s,t}{=}1$ if the server is in active state at t.

The key parameters and decision variables used in the problem formulation are summerized in Table III.

V. Problem Formulation for Short Timescale Operations

We now describe the event-driven problem formulation that aims to reduce the data center's power consumption as well as the possible SLA violations. We emphasize that the orchestrator operates without knowledge of future request arrivals. Therefore, its design focuses on optimizing energy consumption based solely on the currently known events in the system (e.g., request execution ends event).

Below, we denote the case of queue Q_k not including any unhandled request by setting $\epsilon_k{=}1$; otherwise, $\epsilon_k{=}0$. Also, at time t, the CPU load $\lambda_{s,t}$ of a server s is the sum of the CPU load of all the running and transiting containers on s, i.e.,

$$\lambda_{s,t}{=}\sum_{k\in\mathcal{K}}\Bigg(\sum_{c_{k,i,R}\in\Omega^s_{k,R}(t)}\mu_{c_{k,i,R}}{+}\sum_{c_{k,T_{C\to R}}\in\Omega^s_{k,T_{C\to R}}(t)}\mu_{k,T_{C\to R}}{+}\sum_{c_{k,T_{R\to C}}\in\Omega^s_{k,T_{R\to C}}(t)}\mu_{k,T_{R\to C}}\Bigg). \tag{1}$$

Similarly, at t, the amount of consumed memory, $\nu_{s,t}$, on server s is the sum of that consumed by all the running, transiting, and warm containers on s, i.e.,

$$\nu_{s,t}{=}\sum_{k\in\mathcal{K}}\Bigg(\sum_{c_{k,i,R}\in\Omega^s_{k,R}(t)}\tau_k{+}\sum_{c_{k,W}\in\Omega^s_{k,W}(t)}\tau_{k,W}{+}\sum_{c_{k,T_{C\to R}}\in\Omega^s_{k,T_{C\to R}}(t)}\tau_{k,T_{C\to R}}{+}\sum_{c_{k,T_{R\to C}}\in\Omega^s_{k,T_{R\to C}}(t)}\tau_{k,T_{R\to C}}\Bigg). \tag{2}$$

TABLE III
LIST OF KEY SYMBOLS USED IN THE PROBLEM FORMULATION

Symbol	Description
Parameters for Servers	
$\mathcal{S}$	Set of available servers
$\hat{\tau}_s$	Memory capacity of server s (Bytes)
$\hat{\mu}_s$	CPU capacity of server s (cycles/s)
$\Omega^s_{k,X}(t)$	Sets of containers of MS k on server s at time t, where X can be $W, R, T_{C\to R}$ or $T_{R\to C}$
P_s	Power consumption of server s
$\lambda_{s,t}$	CPU load of server s at time t
$\nu_{s,t}$	Memory load of server s at time t
$\gamma_{s,t}$	State of the server t at time t
Parameters for MSs	
$\mathcal{K}$	Set of available service types
τ_k	Required memory of MS k (Bytes)
w_k	Required computation of MS k (CPU cycles)
D_k	Target maximum delay of MS k (Seconds)
Q_k	Queue for pending requests of MS k
Parameters for Containers	
C, W, R	States of a container: cold, warm, and running
$T_{C\to R}$	Transition state of a container from C to R
$T_{R\to C}$	Transition state of a container from R to C
$c_{k,i,R}(t)$	Container in state R serving $r_{k,i}$ at time t
$c_{k,i,T_{C\to R}}(t)$	Container in transition from C to R to serve $r_{k,i}$ at time t
$c_{k,W}(t)$	Container in state W at time t
$\delta_{k,T_{X\to Y}}$	Transition time of a container of MS k from state X to state Y, where X, Y can be C or R (seconds)
$\tau_{k,X}$	Memory consumed by a container of MS k in state X, where X can be $W, T_{C\to R}$ or $T_{R\to C}$ (Bytes)
$\mu_{k,X}$	CPU consumed by a container of MS k in state X, where X can be $T_{C\to R}$ or $T_{R\to C}$ (Bytes)
$\Omega_{k,X}(t)$	Sets of containers of MS k at time t, where X can be $W, R, T_{C\to R}$ or $T_{R\to C}$
Parameters for Requests	
$r_{k,i}$	i-th request for MS k
$T_{k,i}$	Processing time of request $r_{k,i}$ (seconds)
$t_{r_{k,i}}$	Queue admission time of request $r_{k,i}$
Decision variables for new request arrival event	
$x_{r_{k,j},c_{k,W}}$	Binary decision variable indicating whether HoL request to handle in the queue $r_{k,j}$ is scheduled on the warm container $c_{k,W}$
$q_{r_{k,i}}$	Binary decision variable indicating whether the newly arrived request $r_{k,i}$ is queued
$z_{r_{k,j},c_{k,C}}$	Binary decision variable denoting whether HoL request $r_{k,j}$ is assigned to a cold container $c_{k,C}$
$y_{r_{k,j},s}$	Binary decision variable denoting whether the container $\hat{c}_{k,j,R}$ is scheduled to run on server s
$\mu_{r_{k,j}}$	Discrete decision variable indicating the CPU speed allocated to the container $\hat{c}_{k,j,R}$ serving request $r_{k,j}$
$\mu_{r_{k,j+n}}$	Discrete decision variable indicating the CPU speed allocated to n-th queued request $r_{k,j+n}$ in Q_k
$\mu_{r_{k,i}}$	Discrete decision variable indicating the CPU speed allocated the newly arrived request $r_{k,i}$ if $q_{r_{k,i}}$=1
Decision variables for request execution end event	
$\hat{x}_{r_{k,j},c_{k,i,R}}$	Binary decision variable indicating whether HoL request in the queue $r_{k,j}$ runs on the running container that had just finished its current execution
$e_{c_{k,i,r}}$	Integer decision variable indicating whether the container $c_{k,i,R}$ is kept in R, W, or C state
$\mu_{r_{k,j}}$	Discrete decision variable indicating the CPU speed allocated to the container $\hat{c}_{k,j,R}$ serving HoL request $r_{k,j}$
$\mu_{r_{k,j+n}}$	Discrete decision variable indicating the CPU speed allocated to the n-th queued request $r_{k,j+n}$ in Q_k

A. Problem formulation of new request arrival event

At t, upon the arrival of a new request, $r_{k,i}$, the orchestrator makes one of the following decisions on MS k:
1) To queue $r_{k,i}$ into the queue Q_k. In this case, let $t_{r_{k,i}}$ be the queue admission time of $r_{k,i}$, which is set to t. The decision of enqueueing $r_{k,i}$ is expressed by setting $q_{r_{k,i}}$=1. While enqueueing $r_{k,i}$, the orchestrator decides the CPU speed to allocate to it, $\mu_{r_{k,i}}$. Notice that the newly arrived request is queued only if existing containers (including the newly started container for the HoL request) can serve it within the target delay. To ensure this, one must estimate the worst-case queueing delay for the new request by considering when currently running/transiting containers become available to serve it. This estimation relies on the processing times, and thus, CPU speeds allocated to the already-queued requests. It follows that deciding the CPU speed for the queued requests is a crucial step. Importantly, if later on the number of containers serving that queue increases or decreases, the CPU speed chosen for the queued requests can be revised.

(1a) To schedule HoL request in the queue $r_{k,j}$ on an existing warm container $c_{k,W}(t)$ in server $s \in \mathcal{S}$. We denote this decision by setting $x_{r_{k,j},c_{k,W}}$=1 (Case 1.1 of Fig. 3). The decision variable $y_{r_{k,j},s}$=1 indicates that the request is served in server s. The container $c_{k,W}(t)$ moves to state R (denoted by $c_{k,j,R}(t)$) with negligible startup latency, and serves $r_{k,j}$. The CPU and the memory allocated to $c_{k,j,R}$ are set equal to, respectively, the CPU allocated to $r_{k,j}$ and to MS k's memory requirements. Also, the orchestrator revises the CPU speed allocated to all the requests in the queue. For the n-th request $r_{k,j+n}$ in Q_k, with j corresponding to the first request, let $\mu_{r_{k,j+n}}$ be the revised CPU speed allocated to it, where $n \in \{0, 1, ..., |Q_k| - 1\}$.

(1b) To schedule HoL request in the queue $r_{k,j}$ on a cold container in server s ($y_{r_{k,j},s}$=1), denoted by $z_{r_{k,j},c_{k,C}}$=1 (Case 1.2 of Fig. 3). That is, at t, a new container is created and starts the transition to R ($c_{k,j,T_{C\to R}}(t)$). Additionally, $r_{k,j}$ is removed from the queue of waiting requests and it waits in the MS's auxiliary queue to finish the transition of $c_{k,j,T_{C\to R}}(t)$. After the startup latency of $\delta_{k,T_{C\to R}}$, $r_{k,j}$ is run on $c_{k,j,R}(t + \delta_{k,T_{C\to R}})$. At t, the orchestrator sets the CPU speed of $c_{k,j,R}(t+\delta_{k,T_{C\to R}})$ to $\mu_{r_{k,j}}$, while the memory allocated to $c_{k,j,R}(t + \delta_{k,T_{C\to R}})$ is that required by MS k. As before, the orchestrator revises the CPU speed allocated to all the queued requests.

(1c) Not to start a cold or warm container to serve the first request in the queue, denoted by setting $z_{r_{k,j},c_{k,C}}$=0 and $x_{r_{k,j},c_{k,W}}$=0, i.e., the first request in Q_k, $r_{k,j}$, remains in the queue for the time being (Case 1.3 of Fig. 3), and the CPU speed assigned to queued requests is not revised.
2) The orchestrator drops the new request $r_{k,i}$ (Case 2 of Fig. 3). This decision is denoted by setting $q_{r_{k,i}}$=0.

B. Queueing delay

We now present a recursive procedure to obtain the worst-case queueing delay of the newly arrived request $r_{k,i}$ and the residual queueing delays of the queued requests in Q_k. In such calculations, we do not make any prediction on whether

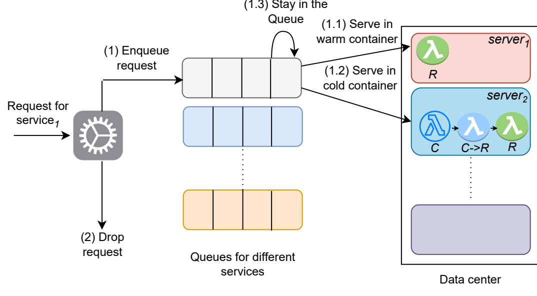

Fig. 3. Decisions that the orchestrator can make upon a new request arrival.

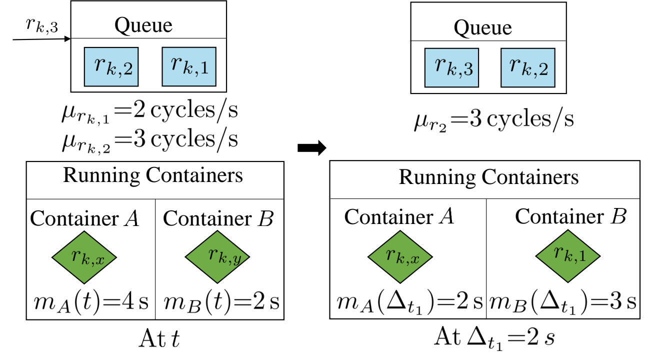

Fig. 4. An example of queueing delay calculations for queued requests.

queued requests are served in cold or warm containers, except for the first request to handle in the queue; instead, the orchestrator reuses the currently running, transiting (in transition state $T_{C\to R}$) containers whenever they finish their assigned requests. Also, we do not predict new request arrivals, in order not to further increase the problem complexity.

Let $m_k(t) \in \mathcal{M}_k(t)$ denote the time at which currently running, transiting, or warm containers are available to serve the first request to handle in the queue, relative to current time t. For the first request to handle, $r_{k,j}$, let $\mu_{r_{k,j}}$ and Δt_j be its revised CPU speed allocation and the time at which an existing container will start the execution of $r_{k,j}$, relative to t. The orchestrator will run the first request $r_{k,j}$ waiting in the queue on the container that can start serving such request at the earliest, i.e., at $\Delta t_j{=}\min(\mathcal{M}_k(t))$. The queueing delay experienced by $r_{k,j}$ so far is $t{-}t_{r_{k,j}}$, where we recall that $t_{r_{k,j}}$ is the request queue admission time. Additionally, $r_{k,j}$ will stay for Δt_j amount of time in the queue before being served. To get served within the target delay, the revised CPU speed of $r_{k,j}$ is calculated as: $\mu_{r_{k,j}}{=}w_k/[D_k{-}(t{-}t_{r_{k,j}}{+}\Delta t_j)]$. When this container starts serving $r_{k,j}$, the container's residual time to finish the execution will be equal to $r_{k,j}$'s processing time. Let $\mathcal{M}_k(\Delta t_j)$ be the set of updated residual times of the running and transiting containers at Δt_j, which is given by

$$\mathcal{M}_k(\Delta t_1){=}\Big\{m_k(t){-}\Delta t_1,\\ \forall m_k(t) \in \mathcal{M}_k(t)\setminus\min(\mathcal{M}_k(t))\Big\}{\cup}\Big\{\frac{w_k}{\mu_{r_{k,1}}}\Big\} \quad (3)$$

where $m_k(t)$ is the generic element of set $\mathcal{M}_k(t)$.

Similarly, with respect to the current time t, the second request to handle, $r_{k,j+1}$, will be served at $\Delta t_{j+1}{=}\Delta t_j{+}\min(\mathcal{M}_k(\Delta t_j))$. The revised CPU speed of $r_{k,j+1}$ is then: $\mu_{r_{k,j+1}}{=}w_k/[D_k{-}(t{-}t_{r_{k,j+1}}{+}\Delta t_{j+1})]$. The set, $\mathcal{M}_k(\Delta t_{j+1})$, containing the time relative to Δt_{j+1} at which running and transiting containers can handle the third request in Q_k is,

$$\mathcal{M}_k(\Delta t_{j+1}){=}\Big\{m_k(\Delta t_j){-}(\Delta t_{j+1}{-}\Delta t_j),\\ \forall m_k(\Delta t_j) \in \mathcal{M}_k(\Delta t_j)\setminus\min(\mathcal{M}_k(\Delta t_j))\Big\}{\cup}\Big\{\frac{w_k}{\mu_{r_{k,j+1}}}\Big\}.$$

Generalizing the above equation, the n-th request to handle in the queue, $r_{k,j+n}$, will be served at $\Delta t_{j+n}{=}\Delta t_{j+n-1}{+}\min(\mathcal{M}_k(\Delta t_{j+n-1}))$. By revising the CPU speeds, the time at which running and transiting containers can handle such request relative to Δt_{j+n} is:

$$\mathcal{M}_k(\Delta t_{j+n}){=}\Big\{m_k(\Delta t_{j+n-1}){-}(\Delta t_{j+n}{-}\Delta t_{j+n-1}),\\ \forall m_k(\Delta t_{j+n-1}){\in}\mathcal{M}_k(\Delta t_{j+n-1})\setminus\min(\mathcal{M}_k(\Delta t_{j+n-1}))\Big\}\\ \cup\Big\{\frac{w_k}{\mu_{r_{k,j+n}}}\Big\}. \quad (4)$$

Thus, the residual queueing delay of the n-th queued request $r_{k,j+n}$ is given by Δt_{j+n}, where $n \in \{0, 1, ..., |Q_k| - 1\}$.

The queueing delay of the newly arrived request $r_{k,i}$ will be the sum of the time at which the last request in the queue will start being executed, and the updated remaining time of the running and transiting containers at $\Delta t_{j+|Q_k|-1}$, i.e.,

$$d_{r_{k,i}}(\mathcal{M}_k(t)){=}\Delta t_{j+|Q_k|-1}{+}\min(\mathcal{M}_k(\Delta t_{j+|Q_k|-1})). \quad (5)$$

To clarify the queueing delay calculations, we provide an example below.

Example 1 (Queueing delay of the newly arrived request r_k)**.** *As depicted in Fig. 4, consider a new request, $r_{k,3}$, arriving at the orchestrator when two requests, $r_{k,1}$ and $r_{k,2}$, are already queued Q_k. Assume the workload requirement w_k of service k is 6 CPU cycles, and let the CPU speed allocated to $r_{k,1}$ and $r_{k,2}$ be $\mu_{r_{k,1}}{=}2$ CPU cycles/s, and $\mu_{r_{k,2}}{=}3$ CPU cycles/s (resp.). Also, we have two containers for MS k, A and B, currently running the requests $r_{k,x}$ and $r_{k,y}$ (resp.). The time (relative to t) at which these containers can run the first request waiting in the queue is $\mathcal{M}_k(t){=}\{4,2\}$. Let us assume that the orchestrator does not create a new container from cold/warm state to serve the queued requests. Then $r_{k,1}$ will be served at Δt_1 on a container which can serve $r_{k,1}$ at the earliest, i.e., on B. Hence, $\Delta t_1{=}\min(\mathcal{M}_k(t)){=}2\,s$. Now, the container B requires a processing time of $\frac{w_k}{\mu_{r_{k,1}}}$, which is 3 s. Thus, the time at which the two containers can handle the next requests relative to Δt_1 is given by $\mathcal{M}_k(\Delta t_1){=}\{2,3\}$. At $\Delta t_2{=}\Delta t_1{+}\min\{\mathcal{M}_k(\Delta t_1)\}{=}4\,s$, the container A will serve the request $r_{k,2}$, hence the queueing delay that $r_{k,2}$ will experience is 4 s, and $\mathcal{M}_k(\Delta t_2){=}\{2,1\}$. The worst-case queueing delay of the newly arrived request, $r_{k,3}$, is then $\Delta t_2{+}\min\{\mathcal{M}_k(\Delta t_2)\}{=}5\,s$.*

We use the above estimated worst-case queueing delays in Sec. V-D for making decisions on starting a new container from cold/warm state, and in Secs. V-D and V-E to revise the allocated CPU speeds.

C. Pre-computation of power consumption

Upon the arrival of a new request, $r_{k,i}$, the orchestrator has to make a decision that minimizes the power consumption of the servers in the data center. The optimal policy for this problem is non-trivial. On one hand, keeping the requests longer in the queue may reduce the total number of running and transiting containers in the system, but it may lead to a higher CPU speed allocation to such running containers and, hence, higher overall power consumption. On the other hand, if we keep the requests for a shorter duration in the queue, we may have a higher number of transiting and running containers with lower CPU speed allocated, again resulting in a higher power consumption. Moreover, this latter policy may reduce the reusability of the same container to serve other requests. Thus, to minimize power consumption, considering just the CPU cycles assigned to currently running and transiting containers is not enough. Instead, it is critical to also account for the impact of queued requests on the power consumption. To this end, during every request arrival, we pre-compute the possible power consumption according to the various decisions the orchestrator can make for the first request in the queue and the newly arrived request.

Given the time t_1 of the current arrival, the orchestrator knows the request execution end and transition complete events for the existing transiting and running containers. Let $\mathcal{L} = \{t_1, t_2, ...\}$ be the set of the time instants at which these known events will occur. Based on the decisions made for the new request and the first request in the queue, the orchestrator updates $\mathcal{L}$ to include the occurrence time of execution end for the newly arrived request and the queued requests as well.

Let $\mathcal{L}_q$ be the auxiliary set containing the time at which each of the known events will occur if the orchestrator decides to queue the newly arrived request and not to serve the first request in the queue (i.e., $q_{r_{k,i}}$=1). Then, the power consumption P_q across all the servers in the data center is $P_q = \sum_{\hat{s}\in\mathcal{S}} \sum_{i=1}^{|\mathcal{L}_q|-1} \int_{t_i}^{t_{i+1}} (P_{\text{idle}} + P(\lambda_{\hat{s},t_i,q}))dt$ where $\lambda_{\hat{s},t_i,q}$ is the load of server $\hat{s}$ at t_i. Similarly, the orchestrator pre-computes the power consumption for the other decisions. If the orchestrator decides to queue the new request and start a warm container on server s to serve the first request in the queue, the power consumption $P_{x,s}$ across all the servers in the data center is $P_{x,s} = \sum_{\hat{s}\in\mathcal{S}} \sum_{i=1}^{|\mathcal{L}_{x,s}|-1} \int_{t_i}^{t_{i+1}} (P_{\text{idle}} + P(\lambda_{\hat{s},t_i,x,s}))dt$, where $\lambda_{\hat{s},t_i,x,s}$ is the load of server $\hat{s}$ at t_i and $\mathcal{L}_{x,s}$ is the auxilliary set containing the time of occurrence of known events. If the decision is to use a cold container on server s to serve the first request in the queue, then the pre-computed power consumption is represented by $P_{z,s} = \sum_{\hat{s}\in\mathcal{S}} \sum_{i=1}^{|\mathcal{L}_{z,s}|-1} \int_{t_i}^{t_{i+1}} (P_{\text{idle}} + P(\lambda_{\hat{s},t_i,z,s}))dt$. Finally, if the new request is not queued, the pre-computed power consumption is $P = \sum_{\hat{s}\in\mathcal{S}} \sum_{i=1}^{|\mathcal{L}|-1} \int_{t_i}^{t_{i+1}} (P_{\text{idle}} + P(\lambda_{\hat{s},t_i}))dt$.

D. Objective function for arrival event

Consider the arrival at the orchestrator of a new request for MS k, $r_{k,i}$. The objective is to minimize power consumption across all the servers as well as minimize the expected power consumption in the future, based on the currently known events (i.e., request execution end and transition complete), while maximizing the number of served requests. To avoid further increasing the complexity of the problem formulation, we do not assume to know the future arrival events. We maximize the number of requests served by adding a high penalty, F, if the orchestrator decides to drop the newly arrived request (i.e., $q_{r_{k,i}}$=0).

Recall that $\mathcal{M}_k(t)$ represents the residual times of the currently running and transiting containers in state $T_{C\to R}$, to finish their assigned requests relative to current time t. Let $\mathcal{M}_{k,x}(t)$ be an auxilliary set representing the updated version of $\mathcal{M}_k(t)$, if the orchestrator decides to use an existing warm container to serve HoL request in the queue (i.e., $x_{r'_k,c_{k,W}}$=1). Since the latency associated with a warm container transiting to the running state is negligible, we have: $\mathcal{M}_{k,x}(t) = \mathcal{M}_k(t) \cup \{0\}$. Similarly, let $\mathcal{M}_{k,z}(t)$ be another auxilliary set denoting the updated version of $\mathcal{M}_k(t)$, if the orchestrator decides to start a new container (cold-start) to serve HoL request (i.e., $z_{r'_k,c_{k,C}}$=1). Given the startup latency for a cold container ($\delta_{k,T_{C\to R}}$), we have: $\mathcal{M}_{k,z}(t) = \mathcal{M}_k(t) \cup \{\delta_{k,T_{C\to R}}\}$.

Next, based on the sets $\mathcal{M}_k(t)$, $\mathcal{M}_{k,x}(t)$, and $\mathcal{M}_{k,z}(t)$, the orchestrator can estimate the possible residual queueing delays of the currently queued requests and that of the new request as described in Sec. V-B.

Thus, upon the arrival of $r_{k,i}$, the orchestrator should solve the following problem:

$$\begin{aligned}\min_{\{y,x,q,z,\{\mu\}\}} \Big[& q_{r_{k,i}} \cdot (1 - x_{r_{k,j},c_{k,W}}) \cdot (1 - z_{r_{k,j},c_{k,C}}) \cdot P_q \\ & + q_{r_{k,i}} \cdot x_{r_{k,j},c_{k,W}} \cdot (1 - z_{r_{k,j},c_{k,C}}) \cdot \sum_{s\in\mathcal{S}} y_{r_{k,j},s} \cdot P_{x,s} \\ & + q_{r_{k,i}} \cdot (1 - x_{r_{k,j},c_{k,W}}) \cdot z_{r_{k,j},c_{k,C}} \sum_{s\in\mathcal{S}} y_{r_{k,j},s} \cdot P_{z,s} \\ & + (1 - q_{r_{k,i}}) \cdot P \Big] + (1 - q_{r_{k,i}}) \cdot F \end{aligned} \quad (6)$$

subject to the constraints below:

$$\lambda_{s,t} + y_{r_{k,j},s} \cdot (x_{r_{k,j},c_{k,W}} \cdot \mu_{r_{k,j}} + z_{r_{k,j},c_{k,C}} \cdot \mu_{k,T_{C\to R}}) \le \gamma_{s,t} \hat{\mu}_s \ \forall s \quad (7)$$

$$\nu_{s,t} + y_{r_{k,j},s} \cdot (x_{r_{k,j},c_{k,W}} \cdot \tau_k + z_{r_{k,j},c_{k,C}} \cdot \tau_{k,T_{C\to R}}) \le \gamma_{s,t} \hat{\tau}_s \ \forall s \quad (8)$$

$$x_{r_{k,j},c_{k,W}} \cdot \left(t - t_{r_{k,j}} + \frac{w_k}{\mu_{r_{k,j}}} \right) \le D_k \quad (9)$$

$$z_{r_{k,j},c_{k,C}} \cdot \left(t - t_{r_{k,j}} + \delta_{k,C} + \frac{w_k}{\mu_{r_{k,j}}} \right) \le D_k \quad (10)$$

$$\begin{aligned} & q_{r_{k,i}} \cdot \Big[(1 - x_{r_{k,j},c_{k,W}}) \cdot (1 - z_{r_{k,j},c_{k,C}}) \cdot d_{r_{k,i}}(\mathcal{M}_k(t)) + \\ & (1 - x_{r'_k,c_{k,W}}) \cdot z_{r'_k,c_{k,C}} \cdot d_{r_{k,i}}(\mathcal{M}_{k,z}(t)) + \\ & (1 - z_{r'_k,c_{k,C}}) \cdot x_{r'_k,c_{k,W}} \cdot d_{r_{k,i}}(\mathcal{M}_{k,x}(t)) + \frac{w_k}{\mu_{r_{k,i}}} \Big] \le D_k \end{aligned} \quad (11)$$

$$t-t_{r_{k,j+n}}+(1-x_{r_{k,j},c_{k,W}})(1-z_{r_{k,j},c_{k,C}})d_{r_{k,j+n}}(\mathcal{M}_k(t))$$
$$+(1-x_{r_{k,j},c_{k,W}})z_{r_{k,j},c_{k,C}}\cdot d_{r_{k,j+n}}(\mathcal{M}_{k,z}(t))+(1-z_{r_{k,j},c_{k,C}})$$
$$\cdot x_{r_{k,j},c_{k,W}}\cdot d_{r_{k,j+n}}(\mathcal{M}_{k,x}(t))+\frac{w_k}{\mu_{r_{k,j+n}}}\leq D_k$$
$$\forall n\in\{0,1...|Q_k|-1\} \tag{12}$$

$$\sum_{s\in\mathcal{S}} y_{r_{k,j},s}=x_{r_{k,j},c_{k,W}}+z_{r_{k,j},c_{k,C}} \tag{13}$$

$$x_{r_{k,j},c_{k,W}}\cdot y_{r_{k,j},s}\leq|\Omega^s_{k,W}(t)|\ \forall s\in\mathcal{S} \tag{14}$$

$$x_{r_{k,j},c_{k,W}}+z_{r_{k,j},c_{k,C}}\leq 1 \tag{15}$$

$$x_{r_{k,j},c_{k,W}},\,q_{r_{k,i}},\,y_{r_{k,j},s},\,z_{r_{k,j},c_{k,C}}\in\{0,1\}\,. \tag{16}$$

Constraint (7) ensures that, at any t, the CPU cycles used by the pre-existing running and transiting containers as well as by the newly scheduled container do not exceed the server capability. Constraint (8) guarantees that the memory allocated to pre-existing containers in warm, running, and transiting states, and the newly scheduled container cannot exceed the server capability. If the decision is to run the HoL request, $r_{k,j}$, on a warm container, the total delay experienced by $r_{k,j}$ cannot exceed MS k's target delay (see (9)). If the decision is instead to run $r_{k,j}$ on a cold container, the total delay experienced by $r_{k,j}$ cannot exceed MS k's target delay (see (10)). If the decision is to queue the new request, the total delay the request experiences in the worst case cannot exceed D_k (see (11)). Note that the queueing delay the new request is going to experience depends on the decisions made on the HoL request, and the CPU speed allocated to all queued requests must be revised accordingly ((12) ensures that the MS target delay is met). If $x_{r_{k,j},c_{k,W}}=1$ or $z_{r_{k,j},c_{k,C}}=1$, we must specify on which server the warm or cold container has started its transition, as ensured by (13). Further, when $x_{r_{k,j},c_{k,W}}=1$, $y_{r_{k,j},s}$ can be set only for that server which has a warm container constraint in (14). Constraint (15) ensures that $r_{k,j}$ is scheduled in a warm container, or in a cold container, or later. Finally, (16) specifies that $x_{r_{k,j}}$, $c_{k,W}$, $q_{r_{k,i}}$, $y_{r_{k,j},s}$, and $z_{r_{k,j},c_{k,C}}$ are binary decision variables.

E. Problem formulation for request execution end

Consider now that at time t a container on server $\hat{s}\in\mathcal{S}$ running MS k, $c_{k,i,R}$, finishes its current execution; then, the orchestrator makes one of the following decisions:
1) To schedule HoL request in the queue, $r_{k,j}$, on the container $c_{k,i,R}(t)$ (Case 1 in Fig. 5) . Note that this decision, denoted by setting $\hat{x}_{r_{k,j},c_{k,i,R}}=1$ and $e_{c_{k,i,R}}=2$, is possible since we assume that the transition time of a container from the state R to the state W is negligible. At this stage, $c_{k,i,R}(t)$ is referred to as $c_{k,j,R}(t)$, and the CPU speed allocated to $c_{k,j,R}(t)$ is updated to $\mu_{r_{k,j}}$.
2) Not to schedule HoL request in the queue $r_{k,j}$ on the container $c_{k,i,R}(t)$ denoted by setting $\hat{x}_{r_{k,j},c_{k,i,R}}=0$. Further, to set $\hat{x}_{r_{k,j},c_{k,i,R}}=0$, the orchestrator needs to revise the CPU speed allocated to the queued requests such that they are served within the target delay of the service k. For the n-th queued request, let $\mu_{r_{k,j+n}}$ be its revised CPU speed, where $n\in\{0,1,...,|Q_k|-1\}$. Additionally, the orchestrator makes one of the following decisions regarding $c_{k,i,R}(t)$:

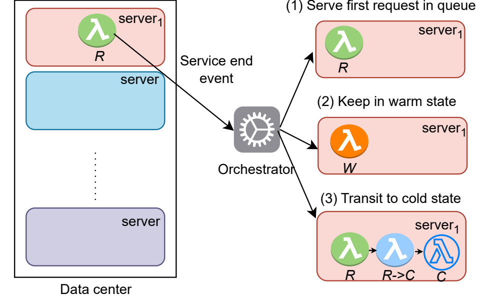

Fig. 5. Decisions the orchestrator can make upon a request execution end.

(2a) To keep $c_{k,i,R}(t)$ in state W, denoted by $e_{c_{k,i,R}}=1$ (Case 2 in Fig. 5). This is useful to reduce the startup latency of future requests and serve them with low CPU speed.

(2b) To delete the container $c_{k,i,R}(t)$, denoted by $e_{c_{k,i,R}}=0$. The container $c_{k,i,R}(t)$ then enters state $T_{R\to C}$, and it will be removed from the system once it reaches state C (Case 3 in Fig. 5). Without knowing the future arrival pattern, deciding whether to keep the container in the state W or C is challenging. To make this decision, we consider probability p_0 of no arrivals within t and $t+\delta_{k,T_{C\to R}}$, and added this as a penalty in the objective function. The orchestrator decides to set $\hat{x}_{r_{k,j},c_{k,i,R}}=0$, only if the target delay of the queued requests can be met with the existing running and transiting containers of MS k excluding $c_{k,i,R}(t)$. To represent the set of residual time to complete the execution for existing running and transiting container, except for $c_{k,i,R}(t)$, we define an auxiliary set $\mathcal{M}_{k,\hat{x}}(t)=\mathcal{M}_k(t)\setminus\{0\}$.

For the request execution end event, through the same procedure as in Sec. V-C, we pre-compute: $P_{\hat{x}}$ i.e., the power consumption if the decision is to set $\hat{x}_{r_{k,j},c_{k,i,R}}=1$, and P_{e_1} and P_{e_0}, i.e., the power consumption if the orchestrator decides to keep $c_{k,i,R}(t)$ in warm and cold states (resp.). Then, the objective function is given by

$$\min_{\{x,e,\{\mu\}\}}\ \hat{x}_{r_{k,j},c_{k,i,R}}\cdot P_{\hat{x}}+(1-\hat{x}_{r_{k,j},c_{k,i,R}})e_{c_{k,i,R}}\cdot P_{e_1}+$$
$$(1-\hat{x}_{r_{k,j},c_{k,i,R}})(1-e_{c_{k,i,R}})P_{e_0}-$$
$$(1-\hat{x}_{r_{k,j},c_{k,i,R}})\Big[e_{c_{k,i,R}}(1-p_0)+(1-e_{c_{k,i,R}})p_0\Big] \tag{17}$$

subject to the following constraints:

$$\lambda_{\hat{s},t}+\hat{x}_{r_{k,j},c_{k,i,R}}\cdot\mu_{r_{k,j}}+(1-\hat{x}_{r_{k,j},c_{k,i,R}})(1-e_{c_{k,i,R}})\mu_{k,T_{R\to C}}\leq \gamma_{s,t}\hat{\mu}_{\hat{s}} \tag{18}$$

$$\nu_{\hat{s},t}+\hat{x}_{r_{k,j},c_{k,i,R}}\cdot\tau_{k,T}+(1-\hat{x}_{r_{k,j},c_{k,i,R}})(1-e_{c_{k,i,R}})\tau_{k,T_{R\to C}}\leq \gamma_{s,t}\hat{\tau}_{\hat{s}} \tag{19}$$

$$(1-\hat{x}_{r_{k,j},c_{k,i,R}})d_{r_{k,j+n}}(\mathcal{M}_{k,\hat{x}}(t))$$
$$+\frac{w_k}{\mu_{r_{k,j+n}}}\leq D_k\ \forall n\in\{0,1,...,|Q_k|\} \tag{20}$$

$$e_{c_{k,i,R}}\geq 2\hat{x}_{r_{k,j},c_{k,i,R}} \tag{21}$$

$$\hat{x}_{r_{k,j},c_{k,i,R}},\in\{0,1\},\ e_{c_{k,i,R}}\in\{0,1,2\}\,. \tag{22}$$

Constraint (18) imposes that the CPU cycles used by currently running, transiting, and the newly scheduled con-

tainer does not exceed the server CPU capacity. Likewise, (19) guarantees that the memory allocated to pre-existing containers in warm, running, and transiting states, and newly scheduled container cannot exceed the server memory. Constraint (20) ensures that the revised CPU speeds allow for meeting the deadline of queued requests, if the orchestrator sets $\hat{x}_{r_{k,j},c_{k,i,R}}$=0. If $\hat{x}_{r_{k,j},c_{k,i,R}}$=0, either the container is kept in W state ($e_{c_{k,i,R}}$=1) or it is destroyed ($e_{c_{k,i,R}}$=0). If $\hat{x}_{r_{k,j},c_{k,i,R}}$=1, the container starts serving request $r_{k,j}$ once it enters state R, i.e., $e_{c_{k,i,R}}$=2 (as per (21)). Finally, (22) specifies that the decision variables $\hat{x}_{r_{k,j},c_{k,i,R}}$ and $e_{c_{k,i,R}}$ are binary and integer variables (resp.).

F. Problem complexity

Theorem 1. *The optimization problem for new request arrival and request execution end events is in the form of a Mixed Integer Non-Linear Problem (MINLP) and is NP-hard.*

Proof. Consider the optimization problem for new request arrival. The non-linearity therein stems from the ratios in the constraints (e.g., w_k/μ in (9)). Let us first linearize these ratios by introducing the new decision variable l=$1/\mu$, and replacing the ratios with $w_k{\cdot}l$; this linearization transforms our problem into an instance of a Mixed Integer Linear Problem (MILP). We prove the NP-hardness by showing that the multi-choice vector bin packing (MVBP) problem [23], which is NP-hard, can be reduced to an instance of our MILP in polynomial time. Let us assume that, when a new request arrives, we always queue it, and serve the HoL request in a warm container. This reduced form is an instance of MVBP whose statement is the following: Given a set of items (requests), each with multiple choices (possible CPU speed allocations), for each choice of an item, there exists a vector representing its size or characteristics (resource demand and experienced delay). The goal is to pack the items in the bins (servers) to minimize the total volume (energy minimization) such that the capacity of the bins (available resources and target delay) is not exceeded. Similarly, the optimization problem for the request execution end event can be proved to be NP-hard. □

G. Model scope and extensions

Application complexity. This work considers serverless applications consisting of single MS executions to simplify the presentation of the system model and optimization problem and to focus on container provisioning decisions. Extending our model to real-world serverless applications involving linear chains of MSs or more complex structures such as Directed Acyclic Graphs (DAGs) introduces additional challenges. Serverless applications often have end-to-end SLA requirements, necessitating consideration of each MS's contribution to overall application latency. To address this, we can decompose the end-to-end SLA into per-MS SLA requirements using techniques [24], [25] based on the mean and covariance of MS response times. Once determined, we can apply the existing model to minimize energy consumption and cold-start occurrences while meeting these end-to-end SLAs. Another challenge arises when a single MS is shared across multiple applications, each with diverse workloads and different end-to-end SLA requirements. We can extend our model to account for various SLA requirements that the shared MS must satisfy simultaneously. While our current model provides a foundation for these extensions, implementing them is beyond the scope of this paper.

Deterministic execution times of MSs. We assume that the workload requirement of each MS, in terms of total CPU cycles, is known and the orchestrator allocates CPU cycles/s to containers, resulting in deterministic execution times. The assumption on the deterministic execution time allows us to simplify the initial model and focus on the core aspects of the energy-efficient container provisioning. However, this simplification has limitations. In real-world scenarios, processing time of the requests can vary due to factors like workload fluctuations, resource contention, and performance interference, even if a suitable number of CPU cycles is allocated to the containers.

Non-deterministic processing time presents significant challenges in resource allocation and in meeting SLA requirements for requests. One primary implication is the potential for SLA violations. When processing times are unpredictable, it becomes increasingly difficult to ensure that requests are served within their target delays. This uncertainty can also lead to suboptimal resource provisioning; if the predicted processing time diverges significantly from the actual time, it may result in either under-provisioning or over-provisioning of resources. Additionally, suboptimal CPU speed allocations can negatively impact overall energy consumption. Thus, although non-deterministic processing time is an important aspect, a detailed analysis of these issues is beyond the scope of this work.

VI. AiW Orchestration Algorithm

In light of the complexity of the above problems, the orchestrator processes the requests in an event-driven manner using the AiW algorithm of COME framework, which consists of `RequestArrival()` and `RequestExecutionEnd()` procedures. When a new request for MS k, $r_{k,i}$, arrives, the orchestrator uses the `RequestArrival()` procedure to determine the possible actions for $r_{k,i}$. Instead, when an existing container of MS k, $c_{k,i,R}$, finishes its execution, the orchestrator uses the `RequestExecutionEnd()` procedure to determine the possible actions for HoL request of Q_k. After scheduling HoL request, `RequestExecutionEnd()` also assesses whether any warm container(s) needs to be transited back to cold state based on the current system load and estimated average arrival rate.

Recall that containers (in W, T, or R states) of one MS cannot be used to serve the requests of other MSs. To take actions on the newly arrived request or the reuse of the container of a specific MS, it is sufficient to consider the waiting requests and containers associated with that particular MS. Thus, AiW opts for one queue per MS, instead of a single global queue operated with scheduling policies like Earliest Deadline First (EDF) or FIFO.

We now describe `RequestArrival()`, outlined in Algorithm 1, in further detail; the related flowchart is depicted in

Fig. 6. The algorithm adopts a "lazy scheduling" policy, i.e., it calculates just the right CPU speed for a warm container to serve $r_{k,i}$ within its target delay (Lines 2–7). Since in this scenario, startup latency and queueing delay of $r_{k,i}$ is 0, the entire service time is dedicated to its processing. The algorithm then identifies a set of eligible servers, $\hat{\mathcal{S}}$, with warm containers of MS k and enough computing and memory resources to serve $r_{k,i}$. If $\hat{\mathcal{S}}\neq\emptyset$, then we select the server with minimum remaining computing capacity (best fit) to execute $r_{k,i}$ in a warm container. If $\hat{\mathcal{S}}$ is empty, the algorithm advances to stage 2 and decides whether to enqueue $r_{k,i}$ or cold-start a container to serve it.

Algorithm 1 AiW orchestration: New request arrival

1: **procedure** `RequestArrival`(t, $r_{k,i}$)
2: $\quad \mu_{r_{k,i}} \leftarrow w_k/D_k$ ▷ Decide CPU speed for warm-start
3: $\quad \hat{\mathcal{S}} \leftarrow \{s \in S | \mu_{r_{k,i}} \leq \gamma_{s,t} \cdot (\hat{\mu}_s - \lambda_{s,t}) \wedge \tau_{\hat{k}} \leq \gamma_{s,t} \cdot (\hat{\tau}_s - \nu_{s,t}) \wedge \Omega^s_{k,W}(t)\}$ ▷ Set of all servers with warm container of service k with sufficient computing and memory resources
4: $\quad$ **if** $\hat{\mathcal{S}} \neq \emptyset$ **then**
5: $\quad\quad$ sort $\hat{\mathcal{S}}$ in increasing order of remaining cpu
6: $\quad\quad$ $\hat{s} \leftarrow \hat{\mathcal{S}}$.pop() ▷ get the server with least remaining CPU speed to serve $r_{\hat{k},j}$
7: $\quad\quad$ start a warm container on $\hat{s}$ to serve $r_{k,i}$
8: $\quad$ **else** ▷ If warm-start is not possible
9: $\quad\quad$ estimate $d_{r_{k,i}}$ as described in Sec. V.B
10: $\quad\quad$ $\mu_{r_{k,i}} \leftarrow w_k/(D_k - d_{r_{k,i}})$ ▷ required cpu speed if request is enqueued
11: $\quad\quad$ **if** $0 < \mu_{r_{k,i}} \leq \hat{\mu}$ **then** ▷ If enqueueing is possible
12: $\quad\quad\quad$ $Q_k \leftarrow Q_k$.append($r_{k,i}$) ▷ Enqueue $r_{k,i}$
13: $\quad\quad$ **else** ▷ Check for cold-start
14: $\quad\quad\quad$ $\mu_{r_{k,i}} \leftarrow w_k/(D_k - \delta_{k,T_{C\to R}})$ ▷ cpu required for cold-start
15: $\quad\quad\quad$ $\mathcal{S}' \leftarrow \{s \in S | \mu_{r_{k,i}} \leq \gamma_{s,t} \cdot (\hat{\mu}_s - \lambda_{s,t}) \wedge \tau_k \leq \gamma_{s,t} \cdot (\hat{\tau}_s - \nu_{s,t})\}$ ▷ Set of all servers with sufficient computing and memory resources for cold-start
16: $\quad\quad\quad$ **if** $\mathcal{S}' \neq \emptyset$ **then**
17: $\quad\quad\quad\quad$ sort $\mathcal{S}'$ in increasing order of remaining cpu
18: $\quad\quad\quad\quad$ $s' \leftarrow \mathcal{S}'$.pop() ▷ get the server with least remaining CPU speed to serve $r_{\hat{k},j}$
19: $\quad\quad\quad\quad$ start a cold container on s' to serve $r_{k,i}$
20: $\quad\quad\quad$ **else** ▷ Neither queueing nor cold-start is possible
21: $\quad\quad\quad\quad$ drop $r_{k,i}$
22: **end procedure**

We remark that enqueueing a newly arrived request avoids container overprovisioning and promotes the reusability of the those running, transiting, or in warm state. Cold-starting a new container promptly is however necessary to ensure that requests are served within their target delay, possibly at a lower CPU speed than if they were enqueued. In stage 2 (Lines 8–12), the orchestrator makes this decision by estimating the queueing delay for $r_{k,i}$, $d_{r_{k,i}}$, based on the residual processing time of the currently running and transiting containers, and queueing delays of already queued requests (see Sec. V-B). If the decision is to enqueue it, it then calculates the CPU speed needed for $r_{k,i}$, $\mu_{r_{k,i}}$, based on $d_{r_{k,i}}$.

Ideally, a currently running container will start serving $r_{k,i}$ with a CPU speed of $\mu_{r_{k,i}}$ at $t+d_{r_{k,i}}$. Nonetheless, if $\mu_{r_{k,i}}$ is very large, the server might not have enough computing resources to serve $r_{k,i}$ at $t+d_{r_{k,i}}$. To get an upper limit on $\mu_{r_{k,i}}$, let us define state variable $\hat{\mu}=\hat{\mu}_{\text{max}}-\mu_{\text{alloc}}$ where μ_{alloc} is

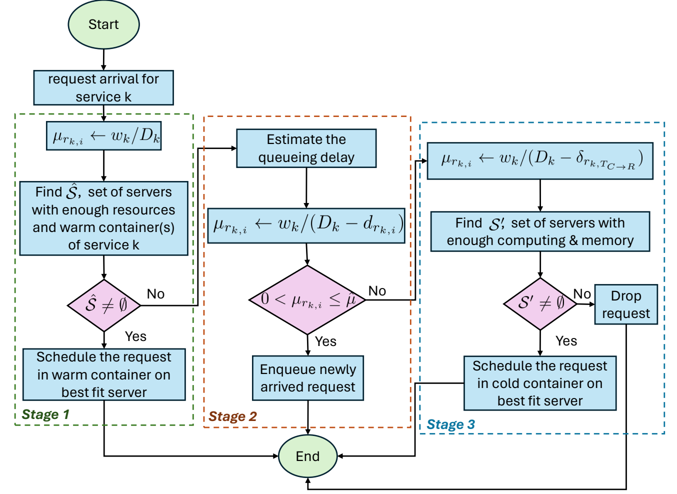

Fig. 6. Flowchart of RequestArrival() procedure.

the maximum of CPU speeds decided for the waiting requests across all the queues and $\hat{\mu}_{\text{max}}$ is the maximum possible CPU speed allocation in the system. If $0<\mu_{r_{k,i}}\leq\hat{\mu}$, then enqueueing $r_{k,i}$ is a feasible decision; otherwise, the algorithm proceeds to stage 3.

In stage 3 (Lines 13–21), it is better to cold-start a new container rather than prolonged waiting in the queue. The algorithm then identifies the set of servers, $\mathcal{S}'$ having enough computing and memory resources to cold-start a container. If $\mathcal{S}'\neq\emptyset$, we cold-start a container on the best fit server and associate $r_{k,i}$ with it. In this case, the processing time available for $r_{k,i}$ is $D_k - \delta_{k,T_{C\to R}}$; otherwise, neither cold/warm-start nor enqueueing is possible for $r_{k,i}$, hence we drop it.

The `RequestExecutionEnd()` of AiW is described in Algorithm 2 and the flowchart is presented in Fig. 7. The procedure begins with keeping the container that has completed its execution, $c_{k,i,R}$, in warm state and subsequently enters stage 1 (Lines 4–11), where it evaluates the possibility of warm-start for HoL request of Q_k. Like before, this decision involves calculating the CPU speed for warm-start. Since in this case startup latency is 0 and queueing delay is $t-t_{r_{k,j}}$, available processing time for $r_{k,j}$ is $D_k-(t-t_{r_{k,j}})$. Next, the procedure identifies the set of eligible servers for warm-start and serves $r_{k,j}$ on the best fit server. If a warm-start is not feasible for $r_{k,j}$, the procedure transitions to stage 2 (Lines 12–19) where it assesses the possibility of a cold-start for $r_{k,j}$. In this stage, the available processing time for $r_{k,j}$ is determined by $D_k-(t-t_{r_{k,j}}+\delta_{k,T_{C\to R}})$, accounting for both cold-start latency and queueing delay. The algorithm serves $r_{k,j}$ on the best-fit server if it exists. Finally, if neither warm nor cold-start is possible for $r_{k,j}$, the procedure drops $r_{k,j}$ and proceeds to handle the next request in Q_k.

We reach stage 3 (Lines 23–24) of `RequestExecutionEnd()` if Q_k is empty or HoL request of Q_k is served on cold or warm container. To turn off unused warm containers whenever possible and reduce the unnecessary memory occupancy, it is important to assess the computing ability available for processing incoming requests. Let $\pi_k(t)$ be the weighted average of the arrival rates

Algorithm 2 AiW orchestration: End of request execution

1: **procedure** RequestExecutionEnd(t,$c_{k,i,R}$,s)
2: keep $c_{k,i,R}$ in warm state on s
3: **while** $|Q_k| > 0$ **do** ▷ keep iterating over Q_k until we serve a request or Q_k becomes empty
4: $r_{k,j} \leftarrow Q_k[0]$ ▷ HoL request of Q_k
5: $\mu_{r_{k,j}} \leftarrow w_k/(D_k - (t - t_{r_{k,j}}))$ ▷ Calculate cpu speed for warm-start
6: $\hat{\mathcal{S}} \leftarrow \{s \in S | \mu_{r_{k,j}} \le \gamma_{s,t} \cdot (\hat{\mu}_s - \lambda_{s,t}) \wedge \tau_{\hat{k}} \le \gamma_{s,t} \cdot (\hat{\tau}_s - \nu_{s,t}) \wedge \Omega^s_{k,W}(t)\}$ ▷ Set of all servers with warm container of MS k with sufficient computing and memory resources
7: **if** $\hat{\mathcal{S}} \neq \emptyset$ **then**
8: sort $\hat{\mathcal{S}}$ in increasing order of remaining cpu
9: $\hat{s} \leftarrow \hat{\mathcal{S}}$.pop() ▷ get the server with least remaining CPU speed to serve $r_{k,j}$
10: start a warm container on $\hat{s}$ to serve $r_{k,j}$
11: break
12: **else** ▷ If warm-start is not possible
13: $\mu_{r_{k,j}} \leftarrow w_k/(D_k - (t - t_{r_{k,j}} + \delta_{k,T_{C\to R}}))$ ▷ cpu required for cold-start
14: $\mathcal{S}' \leftarrow \{s \in S | \mu_{r_{k,j}} \le \gamma_{s,t} \cdot (\hat{\mu}_s - \lambda_{s,t}) \wedge \tau_k \le \gamma_{s,t} \cdot (\hat{\tau}_s - \nu_{s,t})\}$ ▷ Set of all servers with sufficient computing and memory resources for cold-start
15: **if** $\mathcal{S}' \neq \emptyset$ **then** ▷ If cold-start is possible
16: sort $\mathcal{S}'$ in increasing order of remaining cpu
17: $s' \leftarrow \mathcal{S}'$.pop() ▷ get the server with least remaining CPU speed to serve $r_{k,j}$
18: start a cold container on s' to serve $r_{k,j}$
19: break
20: **else** ▷ If cold-start is not possible
21: drop $r_{k,j}$
22: continue ▷ Continue to process next request in Q_k
23: **if** $|\Omega_{k,W}(t)| > \psi_k$ **then**
24: select $|\Omega_{k,W}(t)| - \psi_k$ warm containers at random and transit them to cold state
25: **end procedure**

observed in the past, and $\omega_k(t)$ be the average processing time of the currently running and transiting containers of MS k. Thus, the estimated number of required warm containers at time t is given by $\pi_k(t) \cdot \omega_k(t)$. Acknowledging potential variations in the actual arrival rate compared to $\pi_k(t)$, we slightly overprovision the warm containers by including an overprovisioning factor given by $\frac{\delta_{k,T_{C\to R}}}{D_k - \alpha_k(t)}$, where $\alpha_k(t)$ is the average waiting time of the requests that are currently being processed. Finally, estimated number of required warm containers at t is given by $\Psi_k = \lambda_k \cdot \omega_k(t) \cdot \frac{\delta_{k,T_{C\to R}}}{D_k - \alpha_k(t)}$. If $\Omega_{k,W}(t) > \Psi_k$, we randomly select $\Omega_{k,W}(t) - \Psi_k$ number of warm containers, and transit them back to cold state.

It is worth underlining that the orchestrator may start serving the newly arrived request in a cold container before serving an already queued request. This may occur when a queued request is waiting for one of the running or transiting containers, while it might be impossible to serve a newly arrived request on existing running or transiting containers within its target delay. Nevertheless, our "lazy" CPU allocation policy guarantees that the request arrived first will finish the execution first.

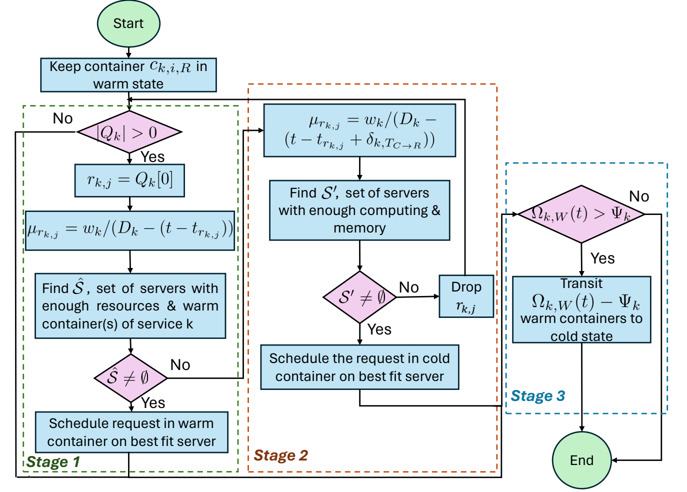

Fig. 7. Flowchart of RequestExecutionEnd() procedure.

VII. Dynamic Server Provisioner (DSP)

In this section, we describe DSP module of the COME framework. DSP is an enhanced version of OpenStack energy saving strategy [26]. DSP reduces the power consumption of the edge data center by keeping just the right number of servers in running state. Conversely, OpenStack keeps the right number of servers active, but some of them may be in idle state. DSP achieves this through two simple strategies: (i) turning on/off servers in response to the current CPU load of the active servers, and (ii) using hysteresis.

Algorithm 3 Dynamic Server Provisioner (DSP)

1: **procedure** ServerProvisioner($\mathcal{S}$)
2: $\eta \leftarrow 0$, $\zeta \leftarrow 0$ ▷ Initialize used/available CPU cycles
3: **for** every s in $\mathcal{S}$ **do**
4: $\eta \leftarrow \eta + \lambda_{s,t}$ ▷ Update used CPU cycles
5: $\zeta \leftarrow \zeta + \gamma_{s,t} \cdot \hat{\mu}_s$ ▷ Update available CPU cycles
6: **if** $\frac{\eta}{\zeta} \ge \theta + A$ **then** ▷ If average load is high
7: **for** every s in $\mathcal{S}$ **do** ▷ Find an inactive server
8: **if** $\gamma_{s,t} = 0$ **then** ▷ Found
9: $\hat{s} \leftarrow s$ ▷ $\hat{s}$ is the server for activation
10: break
11: Activate $\hat{s}$ ▷ Activation of server $\hat{s}$ has started
12: **else** ▷ If average load is low
13: **for** every s in $\mathcal{S}$ **do** ▷ Find an idle server to deactivate
14: **if** $\gamma_{s,t}{=}1 \wedge \lambda_{s,t}{=}0$ **then** ▷ Found an idle server
15: $s' \leftarrow s$ ▷ s' server can be deactivated
16: break
17: $\eta \leftarrow 0$, $\zeta \leftarrow 0$ ▷ Initialize used/available CPU cycles
18: **for** every s in $\mathcal{S} \setminus s'$ **do** ▷ Find the average CPU load excluding idle server s'
19: $\eta \leftarrow \eta + \lambda_{s,t}$
20: $\zeta \leftarrow \zeta + \gamma_{s,t} \cdot \hat{\mu}_s$
21: **if** $\frac{\eta}{\zeta} < \theta - A$ **then** ▷ Deactivate s' only if average CPU load excluding s' is still less than threshold
22: Deactivate s' ▷ Deactivation of s' has started
23: $\gamma_{s',t} \leftarrow 0$ ▷ Update the status of s'
24: **end procedure**

DSP decides on server activation/deactivation after the orchestrator handles a new request arrival or the end of a request execution. To make these decisions, it proceeds as described in Algorithm 3. It first calculates the overall CPU cycles used by

currently running and transiting containers on active servers, which is given by $\eta{=}\sum_{s\in\mathcal{S}}\lambda_{s,t}$, where $\lambda_{s,t}$ is the used CPU cycles of server s at time t. Similarly, the total number of CPU cycles across the active servers is given by $\zeta{=}\sum_{s\in\mathcal{S}}\gamma_{s,t}\hat{\mu}_s$, where $\gamma_{s,t}$ indicates whether server s is currently switched on or off (Lines 2–5). The average CPU load across the active servers is the ratio η/ζ.

If η/ζ is greater than the pre-defined threshold θ (Line 6), DSP goes through the set of servers till it finds one in inactive state ($\gamma_{s,t}{=}0$) and switches it on. Since the server activation involves a delay, the requests can only be scheduled on such a server after its activation (Lines 7–9). Conversely, if DSP detects that the average load of the active servers falls below θ, it goes through the set of servers searching for an idle one, i.e., s.t. $\gamma_{s,t}{=}1$ and $\lambda_{s,t}{=}0$ (Lines 13–15). Let s' denote such a server; DSP decides to deactivate s' only if doing so still maintains the average load below the threshold (Lines 17–22). If the decision is to deactivate s', $\gamma_{s',t}$ is changed to 0.

During activation or deactivation, a server consumes maximum energy because its CPU usage reaches 100%. To avoid energy and delay overheads due to frequent server activation/deactivation, DSP works conservatively implementing an hysteresis scheme with by following two simple strategies. Firstly, DSP cannot activate or deactivate any server while one is currently being activated/deactivated. Secondly, it introduces two different thresholds to prevent frequent switching between the active and inactive states of the server. Specifically, instead of using just one threshold θ for both server activation/deactivation, DSP activates a new server when the average CPU utilization of the data center exceeds $\theta{+}A$ (Line 6), and it deactivates an idle server when the average utilization falls below $\theta{-}A$ (Line 21). The value of A is a design parameter that we optimize experimentally.

VIII. Performance Evaluation

To evaluate the performance of COME, we initially compare AiW against the optimum, in a small-scale scenario without DSP. We then extend our analysis to a larger-scale scenario consisting of MSs with diverse resource requirements and experimentally measured cold-start latency, and we compare COME against state-of-the-art alternatives. In both scenarios, we employ a power model adapted from [14], which assumes an idle power consumption of $P_{\text{idle}}{=}0.121$ kW when the server's CPU load is 0. The power consumption varies linearly as the CPU load increases, reaching 0.750 kW at peak load.

A. Small-scale scenario

To compare AiW against the optimum, and owing to the complexity of the optimization problem at hand, we consider a small-scale scenario including only one type of MS, namely, `img_ocr`, with target delay equal to 5 s. The MS requests arrive according to a Poisson process with varying rate. We set the startup latency of the container running `img_ocr` to be equal to $\delta_{k,C}{=}4.5$ s, and the MS workload requirement is fixed to $w_k{=}8$ G clock cycles. Also, we set the number of active servers in the data center to 2, and we pre-create 3 warm containers per server. The optimal solution is derived using Gurobi. Since the scale of this experiment is small, we also compare AiW and the optimal approach against a trivial algorithm, No-Queue-Always-Warm (NQ-AW). The NQ-AW approach schedules requests immediately in a warm or cold container without queuing and always keeps the container warm after request departure.

Fig. 8(left most) presents the measured average memory consumption across all the servers in the data center for various arrival rates. Notice that all warm, running, and transiting containers contribute to the server's memory utilization. NQ-AW has significantly higher memory consumption due to a large number of active and warm containers, as illustrated in Fig. 8(left). In contrast, memory utilization of AiW is only slightly higher than that of the optimal solution. This is because, in the optimal solution, a large number of requests are queued and a few containers operate at a high CPU speed to serve them. In AiW, thanks to the prompt cold-starts, there are fewer requests in the queue and a slightly higher number of containers operating at a lower CPU speed.

Moving to the power consumption, Fig. 8 (right) shows that the power consumption of AiW and NQ-AW algorithms is the same as that of the optimal solution. Additionally, the energy consumption per request for all three approaches is identical, as illustrated in Fig. 8(right most). It is important to note that, while all three approaches have identical power consumptions, the significantly higher memory utilization by the NQ-AW approach indicates its lack of scalability when dealing with multiple service types in resource-constrained servers. This difference in memory utilization clearly highlights AiW as a more scalable and efficient solution.

B. Simulator design

We develop an event-driven simulator in Python that takes actions when one of the following events occurs: (i) `new_request_arrival`, (ii) `request_execution_completion`, (iii) `container_transition_completion`, (iv) `server_activation`, and (v) `server_deactivation`. The simulator maintains an event queue to track these events.

Initially, we add a `new_request_arrival` event to the queue. Upon the arrival of a new request, the simulator first invokes the `RequestArrival()` procedure of the AiW algorithm, which determines how to handle the request. Based on the decisions of the AiW algorithm, the simulator may add new events to the event queue, such as `container_transition_completion` or `request_execution_completion`. At this point, we also generate a `new_request_arrival` event based on the Poisson arrival rate.

When a `request_execution_completion` event occurs, the simulator calls the `RequestExecutionEnd()` procedure of the AiW algorithm. Depending on the algorithm's output, this may generate further events, such as `request_execution_completion` or `container_transition_completion`. Upon the occurrence of a `container_transition_completion` event, the simulator updates the state of the transitioning container.

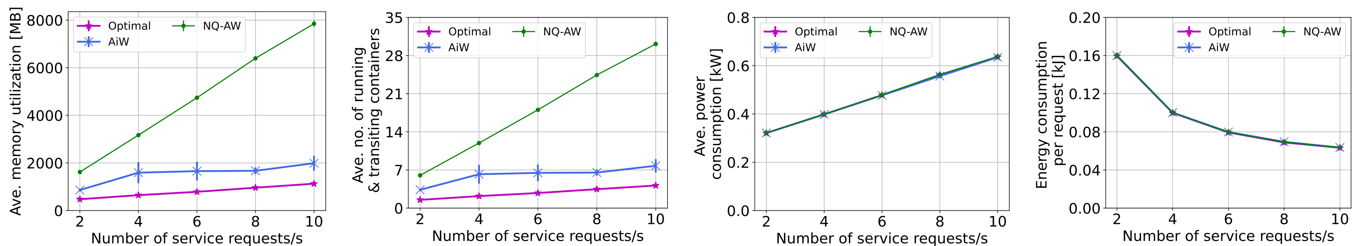

Fig. 8. Performance of AiW vs. optimal, as the arrival rate of service requests varies: average memory utilization (left most); average number of running and transiting containers (left); average power consumption (right); energy consumption per request (right most).

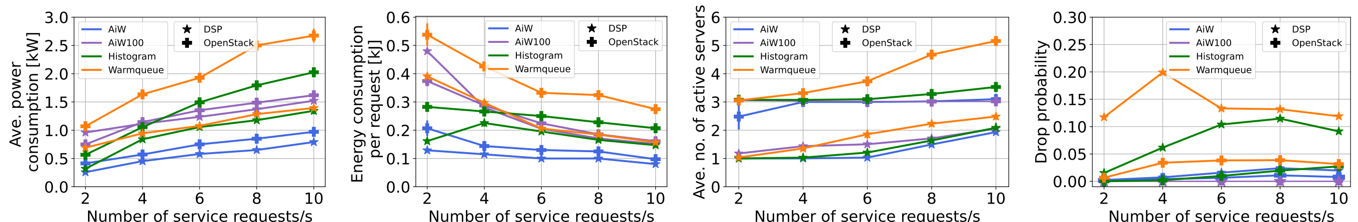

Fig. 9. COME vs. benchmarks: average power consumption (left most); average energy consumption per request (left); average active servers (right); drop probability (right most).

After handling these events, the simulator also invokes the DSP algorithm to check whether a new server needs to be activated or an existing active server should be deactivated. This call may generate additional events, such as `server_activation`, or `server_deactivation`. Finally, when a `server_activation`, or `server_deactivation` event occurs, the simulator marks the state of the server accordingly.

C. Large-scale scenario

We consider now four MSs with different startup latency, workload requirements, and target delays. We begin with setting one server activated all the time and DSP is responsible for activating/deactivating servers in response to the short-timescale decision made by AiW. For comparing performance, we exploit the following alternative methodologies for short-timescale decisions.

- *AiW100*: This scheme uses a threshold on the number of waiting requests in the queue. When queue size exceeds the threshold, it selects HoL request that has the earliest deadline for execution. AiW100 allocates to a container 100% of servers CPU cycles; as a result, only one request at a time per server can be executed.
- *Histogram* [8]: As soon as a request arrives, it is either served or dropped, i.e., there is no queue to hold the waiting requests. Based on the arrival rate, this technique specifies pre-warming and keep-alive periods for warm containers. The CPU is allocated to the container in the same way as in AiW.
- *Warmqueue* [27]: again, requests are served or dropped as soon as they arrive, but, for each MS, it implements a queue for warm containers. Assuming that there can be a maximum number of requests per MS arriving at the system in a given time period (in our simulation, set to 1 s), it sets the size of the warm queue equal to this maximum number of requests.

For larger timescale decisions, we consider the well-known OpenStack energy-saving strategy [26] as the state-of-the-art alternative. This strategy, driven by two input parameters, `standby_nodes_int` and `standby_used_percent`, computes the required number of idle servers as max(`standby_nodes_int`, `standby_used_percent` × `number_of_servers_with_containers`), where `number_of_servers_with_containers` refers to the number of active servers with running or transiting containers. If the current number of idle servers is less than the required one, new servers are activated; otherwise, existing idle servers are deactivated to match the required number of idle servers.

In summary, we compare the performance of COME framework, consisting of AiW for short timescale decisions and DSP for longer scale decisions, against seven different frameworks obtained by pairing each short timescale approach with DSP and the OpenStack energy-saving strategy. Notice that the set of alternatives also includes the scheme obtained by combining AiW and the OpenStack energy-saving strategy. Further, in the following simulations, the time taken to activate/deactivate a server is set to 30 s [28].

1) Fixed arrival pattern: We start by modelling the service request arrivals according to a Poisson process and vary the request arrival rate from 2 to 10 arrivals/s. The results, presented in Fig. 9, reveal that COME exhibits the lowest average power consumption (left most) and energy consumption per request (left) among the considered solutions. When compared to its alternatives, COME minimizes the power and energy consumption per request by 22-64% and 11-66%, respectively.

Further, our findings emphasize that the frameworks employing AiW as the short timescale algorithm (both AiW along with OpenStack and COME), perform better than the considered benchmarks. Despite AiW100's effort to allocate maximum CPU cycles per container, resulting in lowest processing times, its power consumption is higher than that of AiW. This behavior can be attributed to two key observations: (i) AiW100 incurs higher number of active servers than AiW, as shown in Fig. 9(right), and (ii) servers in AiW100 remain idle while waiting for new request arrivals,

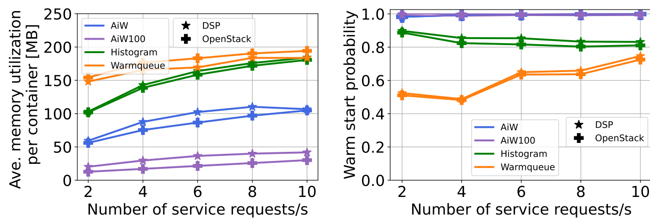

Fig. 10. COME vs. benchmarks: average memory utilization per container (left); warm-start probability (right).

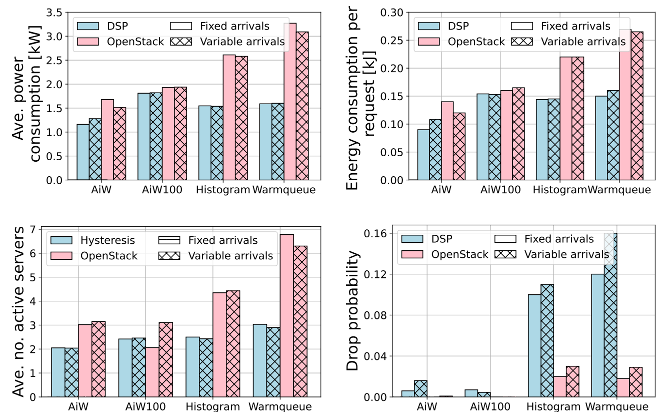

Fig. 11. COME vs. benchmarks with fixed and variable arrival pattern: average power consumption (top left); average energy consumption per request (top right); average number of active servers (bottom left); drop probability (bottom right).

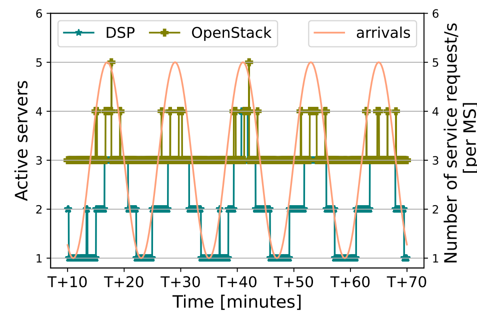

Fig. 12. Dynamic server activations/deactivations as number of service requests varies over time.

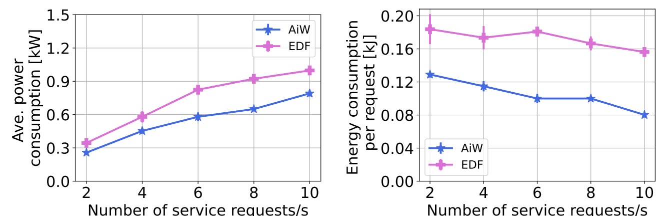

Fig. 13. AiW vs. single-queue EDF: average power consumption (left); energy consumption per request (right).

after serving the existing requests. In contrast, AiW leverages "lazy scheduling", allocating the minimum necessary CPU cycles to the containers to meet their target delays. This approach is enacted by AiW, as it can process multiple requests simultaneously with lower power consumption. Similarly, the higher number of active servers contributes to increased power consumption under the histogram and warmqueue frameworks. Additionally, this behavior is influenced by a decrease in the warm-start probability, as shown in Fig. 10(right).

Fig. 9 also demonstrates that DSP-based frameworks incur lower power consumption than the ones based on OpenStack, owing to the smaller number of activated servers. Notice that, we optimized the input parameters of the OpenStack approach for our scenario, which requires at least one active server in idle state. For this reason, OpenStack requires, in general, a higher number of active servers. Interestingly, despite the advantages in lower power consumption, DSP-based frameworks indicate that there exists a tradeoff with drop probability, which tends to increase (right most in Fig. 9) relatively to the OpenStack approaches.

Fig. 10(left) presents the memory utilization per container, for COME and its alternatives. Recall that memory utilization stems from running, transiting, and warm containers. Notably, AiW100 exhibits the lowest memory utilization per container, attributed to its approach of running a single container per server at any given time. Although AiW requires more memory than AiW100, it still outperforms other state-of-the-art frameworks in terms of memory efficiency. As depicted in Fig. 10(right), both AiW and AiW100 reduce the need for cold-start occurrences, thanks to their queueing approach. Conversely, frameworks employing warmqueue and histogram exhibit lower warm-start probabilities. This behavior underlines the effectiveness of AiW and AiW100 in optimizing memory utilization and in minimizing cold-start occurrences.

2) Variable arrival pattern: Next, we are interested in assessing the robustness of COME under dynamic and fluctuating service request arrival pattern. To this end, we consider a modulated Poisson arrival process, wherein the arrival rate follows a sinusoidal function. Specifically, the average arrival rate is set at 3 arrivals/s per MS, oscillating between a minimum of 1 arrival/s and 5 arrivals/s per MS. The results, depicted in Fig. 11, demonstrate that there are not much variations in the performance of the considered frameworks, thereby emphasizing their robustness. Notably, COME maintains its superior performance across both fixed and variable arrival patterns, again outperforming its state-of-the-art alternatives.

Fig. 12 shows how DSP and OpenStack (combined with AiW as short timescale approach) dynamically activate/deactivate the servers as the request arrival rate varies over the time. Consistently with our intuition, as the arrival rate increases, more servers are activated and, as the rate decreases, servers get deactivated. More interestingly, the number of servers used by OpenStack is higher than DSP, since OpenStack determines an idle number of active servers and maintains them even though unused.

3) Single vs. multi-queue approach: To highlight the advantages of a multi-queue approach over a single queue, we compared the performance of AiW with a single-queue approach operated under the EDF policy. In the EDF policy, when a new request for MS k arrives, it is enqueued. After enqueuing the request, the EDF approach identifies the request with the earliest deadline in the queue, denoted with $r_{k,i}$. If $r_{k,i}$ has an expired deadline, EDF drops it and continues to find next request with earliest deadline. Otherwise, if there are warm containers of MS k available, $r_{k,i}$ is processed in the

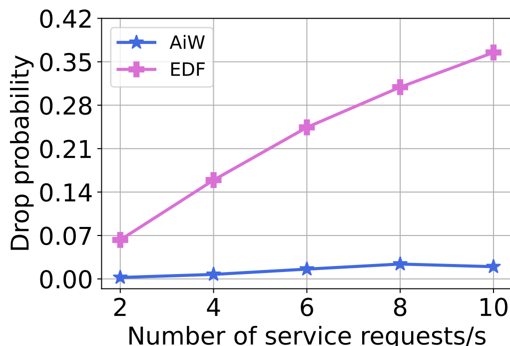

Fig. 14. Drop probability of AiW and single-queue EDF.

warm container. If no warm container is available, similar to AiW, the EDF approach decides between initiating a cold-start or keeping the request waiting in the queue. Upon an execution completion event, we first keep the container that finished its execution in warm state and then try to process the request with the earliest deadline in the warm container. If serving the EDF request in the warm container is not possible due to unavailability of resources on the server where the warm container is, then the orchestrator serves the request in a cold container in the bestfit server.

The results demonstrate that EDF has higher power consumption and energy consumption per request than AiW (Fig. 13), and EDF has a high request drop probability (Fig. 14). This is because EDF prioritizes requests with the earliest deadlines, regardless of which MS the request belongs to. As long as the request with the earliest deadline remains in the queue, requests from other MSs are delayed, waiting for their turn. Since multiple requests with different deadlines are queued, some requests may wait too long in the queue, eventually missing their deadlines, which increases the drop probability. In contrast, AiW employs a multi-queue approach, making independent decisions for each MS. This reduces the probability that queueing delays exceed deadlines and, consequently, the probability of requests being dropped. In summary, these findings further emphasize the importance of employing a multi-queue approach, such as AiW.

D. Discussions on deployment

In this work, we rely on the dynamic allocation of CPU cycles to containers to meet target delay constraints. Our system model and the AiW algorithm assume that CPU allocations can be updated at runtime without restarting containers. In Docker, a popular containerized environment, updating the CPU and/or memory allocation of a running container can be done relatively easily using the `docker update` command without requiring a restart. Additionally, starting from Kubernetes v1.27, the `InPlacePodVerticalScaling` feature enables CPU and/or memory resizing for containers within a running pod without restarting the containers.

We conducted experiments in a Minikube environment (a single-node Kubernetes deployment) using a Python Kubernetes API script, which showed that it takes 30 ms to update the CPU allocation for the running container, which is negligible in our scenarios.

IX. Related Work

The most relevant and recent approaches to ours can be broadly classified into two categories: (i) those exclusively dedicated to reducing the cold-start latency of the containers by redesigning the containers sandboxes and maintaining the warm containers for faster startup by exploiting keep-alive time; ii) approaches that optimize request scheduling for serverless services with multiple scopes and objectives. In this section, we provide a brief summary of these research efforts.

Reducing cold-starts. Major serverless platforms like AWS Lambda [29] and Azure Functions [30], along with open-source platforms like OpenWhisk, minimize cold-starts by employing a fixed keep-alive time for warm containers. However, this simple policy is vulnerable to abuse, as tenants can breach it by sending frequent dummy requests. The histogram approach [8] addresses this issue by defining pre-warm and keep-alive times for warm containers based on the historical data. Additionally, PCPM [9] identifies the creation and initialization of the container network as the most latency-consuming process in container creation. To mitigate this, PCPM pre-creates networking stacks and dynamically binds them to containers, thus reducing cold-start latency. A checkpoint and restore based strategy is proposed in [31]. Instead of starting the containers from scratch, this strategy restores the snapshots of previously running containers. Within the same context, Catalyzer [32] proposes a sandbox design for serverless prioritizing isolation and fast startup by leveraging well formed checkpoint images, bypassing the initialization steps of container startup. All the above approaches have been originally proposed for the cloud, and hence do not take into account the QoS requirements of the edge services. Further, they do not discuss the CPU allocation policies to the containers, which has an impact on the energy footprint of the edge servers.

Request scheduling for serverless services. Many existing frameworks use one-to-one request-to-container mapping [33], which results in an excessive number of containers being provisioned than required to achieve SLAs, increasing the CPU load (hence energy consumption) of the servers. Some existing techniques set a fixed number of container instances and queue requests for this static pool of containers [30], [34]. However, determining the optimal number of container instances is challenging, and doing so in a MS-agnostic manner can result in SLA violations.

The work in [5] investigates retention-aware container caching problem in serverless edge computing. It formulates an optimization problem with the goal of enhancing system efficiency by leveraging container caching jointly with request distribution. This approach takes advantage of the distributed and heterogeneous nature of edge nodes, acknowledging that the cold-start latency and retention cost of a container can vary across nodes based on their computing capacity. The problem is then mapped to the classic ski-rental problem and the study proposes an online algorithm to solve it by incorporating resource constraints and network latency. Although our work shares initial motivations with this study, we diverge in terms of objectives. Specifically, our focus is on improving energy

efficiency in the context of serverless edge computing by exploiting container and server provisioning decisions at two different timescales.

The study in [35] presents a novel approach to enhancing the efficiency of serverless services by introducing dynamic CPU resource management to containers. The primary goal of [35] is to optimize resource utilization while ensuring that the deadline requirements of the services are met. To achieve this objective, it proposes the Dynamic Resource Alteration (DRA) algorithm, which dynamically adjusts CPU allocations to container instances as they approach their deadlines during runtime. Also, DRA considers resource contentions within VMs. In cases where such contentions are detected, DRA takes action by evicting recently initiated requests from constrained VMs and reallocating them to VMs with sufficient free resources. Unlike our work, [35] does not consider the advantages of warm containers for reducing response time. An adaptive resource management framework for serverless platforms is introduced in [36], which optimizes service chains by efficiently bin-packing requests into fewer containers through service-aware container scaling and request batching. To meet SLA requirements, the approach proactively creates containers to prevent cold-starts. Both [36] and AiW aim at reducing cold-starts and avoiding container overprovisioning while meeting SLA requirements. However, they differ in the adopted CPU allocation strategy. AiW employs "lazy scheduling", allocating CPU speed only as needed to meet deadlines, thereby minimizing server energy consumption – an aspect not discussed in [36].

Finally, in our conference paper [1], we proposed a more naive version of AiW. Building upon this foundation, in this work we propose COME, which includes a more effective version of AiW and combines it with DSP to work at two different timescales. Further, we substantially extended our performance evaluation considering a large scale-dynamic scenario and a much richer set of state-of-the-art alternatives.

X. Conclusions

We proposed COME, a framework for dynamically allocating resources to short-lived MSs with strict SLA requirements, in a serverless computing scenario. We specifically designed COME to work in resource-constrained environments, like edge data centers and private clouds. COME minimizes the energy footprint of edge data centers by working at multiple timescales, so that it can act upon both containers and servers provisioning. First, COME minimizes the energy consumption of active servers in a data center by utilizing cold, warm, and running containers over a finite time horizon. It does so by using a low-complexity, yet effective, solution, named AiW, which closely matches the optimum in terms of overall energy and memory consumption of the data center. Then, to reduce power consumption due to idle servers, COME incorporates DSP scheme for dynamically activating/deactivating servers in response to the container provisioning decisions made by AiW. Our results show that COME can reduce the energy consumption by 22-64% when compared to state-of-the-art benchmarks. Furthermore, we demonstrated the robustness of COME under variable arrival patterns of MSs requests.

Acknowledgment

This work was supported by the EC through the SEMANTIC project (Grant No. 861165) and by the DFG through the SWAVES project.

References

[1] M. Adeppady, A. Conte, H. Karl, P. Giaccone, and C. F. Chiasserini, "Energy-aware provisioning of microservices for serverless edge computing," in *IEEE GLOBECOM*, December 2023.

[2] Y. C. Hu *et al.*, "Mobile edge computing a key technology towards 5G," tech. rep., ETSI, 2015.

[3] Z. Tao *et al.*, "A survey of virtual machine management in edge computing," *Proc. of the IEEE*, vol. 107, no. 8, pp. 1482–1499, 2019.

[4] E. Jonas *et al.*, "Cloud programming simplified: A Berkeley view on serverless computing," *arXiv:1902.03383*, 2019.

[5] L. Pan *et al.*, "Retention-aware container caching for serverless edge computing," in *IEEE INFOCOM*, 2022.

[6] R. Xie *et al.*, "Workflow scheduling in serverless edge computing for the industrial internet of things: A learning approach," *IEEE Trans. on Ind. Informatics*, pp. 1–10, 2022.

[7] I. E. Akkus *et al.*, "SAND: Towards high-performance serverless computing," in *USENIX ATC*, 2018.

[8] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC*, 2020.

[9] A. Mohan *et al.*, "Agile cold starts for scalable serverless," in *HotCloud*, 2019.

[10] J. Cadden *et al.*, "SEUSS: skip redundant paths to make serverless fast," in *ACM EuroSys*, 2020.

[11] A. Agache *et al.*, "Firecracker: Lightweight virtualization for serverless applications," in *USENIX NSDI*, 2020.

[12] D. Saxena *et al.*, "Navigating performance-efficiency tradeoffs in serverless computing: Deduplication to the rescue!," *ACM SIGOPS Oper. Syst. Rev.*, 2023.

[13] "Memory and computing power at Amazon Lambda." https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html. Accessed on Mar. 6, 2024.

[14] P. Ruiu *et al.*, "On the energy-proportionality of data center networks," *IEEE Trans. on Sustainable Comp.*, vol. 2, no. 2, pp. 197–210, 2017.

[15] Z. Li *et al.*, "Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing," in *USENIX ATC*, 2022.

[16] "Apache OpenWhisk." https://openwhisk.apache.org/. Accessed on Mar. 6, 2024.

[17] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *IEEE/ACM MICRO*, ACM, 2019.

[18] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 502–504, 2019.

[19] E. Oakes *et al.*, "SOCK: Rapid task provisioning with Serverless-Optimized containers," in *USENIX ATC*, 2018.

[20] "perf: Linux profiling with performance counters." https://perf.wiki.kernel.org/index.php/Main_Page. Accessed on Mar. 6, 2024.

[21] "vmstat – report virtual memory statistics." https://linux.die.net/man/8/vmstat. Accessed on Mar. 6, 2024.

[22] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," *IEEE Trans. on Cloud Computing*, pp. 2834–2847, 2022.

[23] B. Patt-Shamir and D. Rawitz, "Vector bin packing with multiple-choice," *Discrete Applied Mathematics*, vol. 160, no. 10, pp. 1591–1600, 2012.

[24] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *ACM EuroSys*, 2019.

[25] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao, "Rhythm: component-distinguishable workload deployment in datacenters," in *ACM EuroSys*, 2020.

[26] "Energy saving strategy." https://specs.openstack.org/openstack/watcher-specs/specs/pike/implemented/energy-saving-strategy.html. Accessed on Mar. 6, 2024.

[27] G. McGrath *et al.*, "Serverless computing: Design, implementation, and performance," in *IEEE ICDCSW*, 2017.

[28] C. Liu *et al.*, "Minimal cost server configuration for meeting time-varying resource demands in cloud centers," *IEEE Trans. on Parallel and Distributed Systems*, pp. 2503–2513, 2018.
[29] "Amazon Lambda." https://aws.amazon.com/lambda/. Accessed on Mar. 6, 2024.
[30] "Azure functions." https://azure.microsoft.com/en-us/products/functions. Accessed on Mar. 6, 2024.
[31] P. Silva *et al.*, "Prebaking functions to warm the serverless cold start," in *ACM Middleware*, 2020.
[32] D. Du *et al.*, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ACM ASPLOS*, 2020.
[33] L. Wang *et al.*, "Peeking behind the curtains of serverless platforms," in *USENIX ATC*, 2018.
[34] "Cloud functions." https://cloud.google.com/functions. Accessed on Mar. 6, 2024.
[35] A. Mampage *et al.*, "Deadline-aware dynamic resource management in serverless computing environments," in *ACM CCGrid*, 2021.
[36] J. R. Gunasekaran *et al.*, "Fifer: Tackling resource underutilization in the serverless era," in *ACM Middleware*, 2020.

Madhura Adeppady received her Ph.D. in 2024 from Politecnico di Torino, Italy, where she now is a researcher.

Alberto Conte is a senior researcher and project manager at Nokia Bell Labs, in the NSSR lab.

Paolo Giaccone (SM'16) is a Professor at Politecnico di Torino.

Holger Karl is a Professor at the Hasso Plattner Institute, University Potsdam, Germany.

Carla Fabiana Chiasserini (F'18) is a Professor at Politecnico di Torino, Italy, and a WASP Guest Professor at Chalmers University, Sweden.