It Takes Two to Tango: Serverless Workflow Serving via Bilaterally Engaged Resource Adaptation

Jing Wu[1], Lin Wang[2], Quanfeng Deng[1], Chen Yu[1], Dong Zhang[3], Bingheng Yan[3], Fangming Liu[*1,4]
[1] *National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China*
[2]*Paderborn University, Paderborn, Germany*
[3]*Inspur Data Co., Ltd., Jinan, China*
[4]*Peng Cheng Laboratory, Shenzhen, China*
Email: wujinghust@hust.edu.cn, lin.wang@uni-paderborn.de, quanfengdeng@foxmail.com, yuchen@hust.edu.cn, {zhangdong, yanbh}@inspur.com, fangminghk@gmail.com

***Abstract*—Serverless platforms typically adopt an early-binding approach for function sizing, requiring developers to specify an immutable size for each function within a workflow beforehand. Accounting for potential runtime variability, developers must size functions for worst-case scenarios to ensure service-level objectives (SLOs), resulting in significant resource inefficiency. To address this issue, we propose Janus, a novel resource adaptation framework for serverless platforms. Janus employs a late-binding approach, allowing function sizes to be dynamically adapted based on runtime conditions. The main challenge lies in the information barrier between the developer and the provider: developers lack access to runtime information, while providers lack domain knowledge about the workflow. To bridge this gap, Janus allows developers to provide hints containing rules and options for resource adaptation. Providers then follow these hints to dynamically adjust resource allocation at runtime based on real-time function execution information, ensuring compliance with SLOs. We implement Janus and conduct extensive experiments with real-world serverless workflows. Our results demonstrate that Janus enhances resource efficiency by up to 34.7% compared to the state-of-the-art.**

I. Introduction

Serverless computing has become a popular approach for implementing various cloud applications including web services [1], data processing [2], [3] and more recently machine learning training/inference [4], [5]. Serverless computing allows the developer to offload infrastructure management tasks to the cloud provider and ensures high resource elasticity through horizontal auto-scaling. Applications developed as serverless workflows can be represented by directed acyclic graphs (DAGs) where a node presents a function and an edge represents the data exchange between functions. When triggered by an event (i.e., a request), the functions will be executed according to the data flow specified by the DAG. Moreover, horizontal auto-scaling takes care of the number of function instances based on the real-time request intensity. Yet, the size (e.g., CPU cores and memory size) of each function instance is typically decided with an early-binding approach—the developer sets it according to the service-level objective (SLO), e.g., meeting the end-to-end latency target at the 99th percentile (P99) in the DAG [6], [7].

The early-binding approach shoots for the worst case for SLO guarantee and hence leads to considerable resource over-provisioning. Empirically, we observe that the worst-case execution time can be orders of magnitude larger than that of the best case. For example, the gap between the 95th percentile and the 25th percentile of the workflow execution time of Microsoft Durable Functions can be as high as 80 times on average [6]. Such variability can be attributed to various runtime dynamics including varying input working set size [6]–[14] and performance interference [15]–[22]. When the size of the function is decided based on the worst case, the resource utilization will be low for most requests. For example, production serverless traces from Huawei Cloud reveal that half of the deployed functions have CPU and memory usage at merely 10% and 19.5%, respectively [23].

One promising approach for addressing such resource inefficiency is to allow for runtime resource adaptation at the request level. However, the practicality of such an approach is limited by the information barrier between the application developer and the serverless provider. Specifically, application developers do not have real-time access to runtime information necessary for per-request resource adaptation[1]. Meanwhile, the serverless provider lacks the necessary domain knowledge of the application to fine-tune the resources without violating the SLO. Existing works like Kraken [26] and Xanadu [27] employ proactive and reactive resource scalers simultaneously to provision dynamic DAGs where only a subset of functions are invoked per request. Fifer [28] and BATCH [29] allow adjusting function sizes dynamically to achieve high resource utilization with SLO guarantee. While being effective in addressing resource inefficiency, all of them ignore the aforementioned information barrier in real-world systems, rendering them impractical for the current serverless service model.

We propose Janus, a novel runtime resource adaptation framework for serverless workflows. The goal of Janus is to achieve high resource efficiency while guaranteeing workflow

[1]Monitoring services like Azure Monitor [24] and AWS CloudWatch [25] can report function runtime metrics only at one-minute intervals.

SLOs. To this end, Janus adopts a late-binding approach where the developer synthesizes hints containing rules and options for resource adaptation for the serverless provider to perform runtime adaptation on their behalf following the hints passed to them. During the execution of the serverless workflow, when a function in the application DAG finishes, the serverless platform collects the execution time of that function and derives the time budget for the rest of the workflow. Based on the derived time budget, Janus adjusts the sizes of downstream functions using the hints provided by the developer which are ensured to meet the SLO requirement.

The developer synthesizes the hints through comprehensive profiling. Different from current practice which uses P99 of function execution time to calculate the resource allocation, Janus allows the developer to explore different percentiles and obtain the corresponding resource demands as part of the hints. Such detailed hints allow the serverless platform to perform fine-grained resource adaptation, exploiting runtime information to optimize resource allocation to its maximum potential. However, sharing the detailed profiling information and letting the serverless platform search in a large space at runtime for the best resource configuration come with significant space and time overhead. Janus addresses this issue by condensing the hints while retaining their quality.

Overall, this paper makes the following contributions. After introducing the background and motivating the idea (§II), we

- present the design of Janus—a novel runtime resource adaptation framework for serverless workflows to achieve high resource efficiency following a late-binding approach (§III),
- present effective algorithms for synthesizing, condensing, and utilizing hints to realize resource- and time-efficient runtime resource adaptation (§IV),
- implement Janus and perform extensive experiments with two real-world serverless workflows (§V). Experiment results show that Janus is able to to improves resource efficiency by 29.9% and 34.7% on average respectively, compared with the state-of-the-art serverless system, while guaranteeing latency SLOs.

§VI discusses related work and §VII draws final conclusions.

II. Background and Motivation

We introduce the background on serverless workload serving and motivate the use of runtime resource adaptation to address resource inefficiency in existing serverless platforms.

A. Resource Inefficiency with Early Binding

Current serverless workflow platforms (e.g., AWS Step Functions [30] and Azure Durable Functions [31]) offer the opportunity for developers to build various applications with advanced logic like chaining, branching, and parallel execution. These applications can be defined by JSON-based structured languages (e.g., Amazon States Language) or other programming languages. Meanwhile, developers require to specify resource configurations, including memory size, CPU cores, and scaling options, for individual functions—an early-binding approach. The serverless platform is responsible for monitoring the workload intensity and resource usage at runtime and scaling out/in function instances accordingly. To account for potential runtime variability, developers must size the functions in their application workflow accounting for the worst case in order to provide SLO guarantees over the end-to-end delay of request processing, e.g., the 99th percentile (P99) of the end-to-end delay must be within a given target. After deployment, the function sizes become immutable. The worst case is not representative and over-shoots most of the time, leading to resource inefficiency.

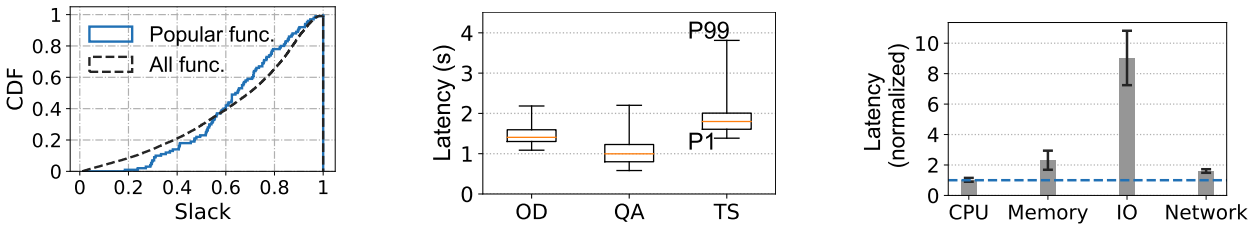
To verify this claim, we conduct a data-driven analysis with a dataset from Microsoft Azure Functions [32] to explicitly demonstrate the resource inefficiency issue. To quantify the inefficiency, we define a metric called *slack*—the margin between the actual execution time and the SLO, which is calculated as $1 - l/T$ with l and T representing end-to-end latency and SLO, respectively. Under certain SLO defined with P99 latency as done by existing works (e.g., [6], [7]), we can see from Figure 1a that more than 60% function invocations have slacks over 60%. Particularly, we analyze slacks of the top 100 most popular functions, whose invocations account for 81.6% of the total function invocations. The result shows that only 20% of the invocations of the popular functions (blue line in Figure 1a) have slacks less than 40%. This means the majority of requests are processed faster than necessary. Notably, in DAG-based workloads (i.e., Azure Durable Functions), the resource inefficiency further deteriorates wherein the ratio between the 95th percentile and 50th percentile is by up to three times [7].

B. Runtime Dynamics

The resource inefficiency caused by the large slack can be mainly attributed to the over-provisioning of resources by the developer. This is to ensure that the SLO is guaranteed even in the worst case (i.e., P99). However, normal cases deviate from the worst case significantly due to runtime dynamics. In particular, we observe that functions face two major dynamic factors at runtime: varying working sets and inevitable performance interference. These two factors contribute significantly to the variance of the function execution time.

Varying working sets. The working set, i.e., input data like videos, audios, and texts, can have varying sizes. Taking Microsoft Azure Function Blobs (storage service) as an example, their data size difference can be as high as nine orders of magnitude [33]. Such a large difference results in substantial variance of the execution time even for the same function [10], [34]. Specifically, we measure the execution time of three functions under different working sets (detailed in §V-A). Figure 1b illustrates the results, where we can observe a variance of up to 3.8 times in function execution caused by varying working set sizes.

Performance interference. For simplicity and security, commercial serverless platforms, such as Alibaba Function Compute, Microsoft Azure, and AWS Lambda, exclusively



(a) (b) (c)

Fig. 1: (a) slacks of function invocations in production traces, (b) function latency variance caused by varying input worksets for functions object detection (OD), question answering (QA), and and text-to-speech (TS), respectively, (c) performance interference attributed to co-location of homogeneous function with different dominant resource demands.

deploy function instances belonging to the same tenant, or even belonging to the same function, in the same virtual machine [35], [36]. For example, the empirical study in [35] shows that in Alibaba Function Compute 65% of the virtual machines exclusively deploy instances of the same function. This co-location of homogeneous function instances, however, can incur severe resource contention on the same resource dimensions, particularly for network bandwidth and memory bandwidth of virtual machines [35]–[38]. To verify this observation, we use a virtual machine to run a function increasing the number of co-located instances from one to six while measuring the execution time of four different functions with resource dominance on different dimensions namely computing, I/O, network, and memory, respectively (detailed in §V-A). As shown in Figure 1c, the co-location of homogeneous functions leads to substantial resource contention and performance interference, prolonging the function execution time up to 8.1 times. The performance interference is often hard to model and predict.

C. Runtime Resource Adaptation

To tackle the aforementioned resource inefficiency issue, we can adopt a late-binding approach through *runtime resource adaptation*, which resizes functions on the fly based on runtime information (e.g., function slacks), achieving higher resource efficiency without violating SLO. For example, given a workflow as a chain of functions, the resource allocation of the downstream functions can be adjusted when the first function finishes execution. This way, the slack from the first function can be exploited to optimize resource efficiency.

The idea sounds straightforward and has been considered in some existing works [2], [26]–[28], [39]. However, most of these works make an unrealistic assumption that either the developer performs the adaptation decision with access to runtime information or the serverless platform provider performs the adaptation with domain knowledge of the application workflow. These assumptions render these solutions impractical to deploy in real-world serverless systems. The information barrier between the developer and the provider calls for a new solution.

We identify the following challenges and opportunities for a full-fledged design for runtime resource adaptation.

Skewed function execution time distribution. Resource allocation for a serverless workflow is typically done by leveraging performance profiles of all the functions in the workflow. During the offline profiling, the execution time distribution for each function is first obtained by running the function with a variety of sample inputs under different resource conditions. Then, given a time budget, existing approaches typically use P99 of the function execution time as a target and calculate the corresponding resource demands. However, due to the high runtime variability, the distribution of the function execution time is highly skewed where the difference between P50 and P99 can be as high as 100 times [23]. This means that if only the function execution time at a single percentile (P50 or P99) is used for resource allocation, there will be significant resource under-provisioning and over-provisioning for most requests at runtime. To address this issue, our idea is to allow for the exploration of the function execution time at diverse percentiles during resource allocation.

Dependencies of adaptation decisions. As the function execution progresses, a sub-workflow will be generated by removing the finished function(s) from the workflow. Within each sub-workflow, the resource adaptation decisions for remaining functions are dependent on each other due to the constraint imposed by the end-to-end latency SLO. For example, under-provisioning a function will result in a reduction of the time budget for executing its downstream functions, thus calling for more resources for these downstream functions to avoid SLO violations. Meanwhile, the selection of the percentile for the execution time of each function dictates resource-latency tradeoff for that function. For example, a higher percentile means that more resources will be allocated to ensure that more requests processed by the function will finish within the given time budget. On the contrary, a lower percentile means that more requests will risk SLO violation, but at the benefits of reduced resource consumption. To address such complex dependencies, we propose the following ideas: (1) We introduce two metrics (i.e., the timeout metric and the resilience metric detailed in §III-B) to balance the resource adaptation decisions of the head function of the current sub-workflow and those of the remaining downstream functions. These metrics help us connect the decision making across sub-workflows and avoids sub-optimal adaptation decisions in each

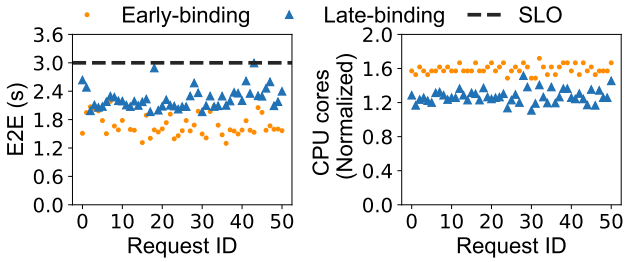


Fig. 2: Performance comparison between early-binding (left) [41] and late-binding (runtime resource adaptation), where the CPU consumption (right) is normalized by the optimal obtained with exhaustive search.

sub-workflow. (2) We explore lower percentiles for the head function and a high percentile (i.e., P99) for other functions in each sub-workflow. Using lower percentiles maximizes the opportunity for resource optimization since any over-time execution of the head function can later be compensated by resource adaptation in the next round. The high percentile ensures that the resource adaptation is not too radical to cause SLO violations.

Tight resource adaptation window. Runtime resource adaptation requires to calculate a new resource allocation decision for the remaining sub-workflow immediately when a function finishes execution. Since serverless functions are typically short-lived (less than 1s on average) [23], [35], [36], [40], the window for resource adaptation is quite tight. Assuming the serverless platform will perform the runtime adaptation on behalf of the developer since the platform has access to full runtime information, the resource adaptation decision making should be fast without involving complex calculations and logic or exploring a large space. As discussed before, the serverless platform provider does not have domain knowledge of the serverless workflow. Hence, the developer must pass the necessary information to the serverless platform for runtime adaptation decision making. Our idea is to let the developer synthesize critical hints containing resource allocation rules and options, which the serverless platform provider utilizes to perform runtime resource adaptation. The hints should be highly condensed so the serverless platform can make adaptation decisions quickly enough.

To demonstrate the potential of runtime resource adaptation incorporating all the above ideas, we take a real-world serverless workflow (explained in §V-A) as an example, and evaluate its end-to-end latency (denoted by E2E) and resource consumption (CPU cores). As illustrated in Figure 2, the late-binding (blue triangle) reduces the resource consumption by up to 42.2% compared with existing early-binding solutions (orange circle), while ensuring SLO guarantees. This highlights the significant gains from runtime resource adaptation.

III. Janus System Design

We present Janus—a novel resource adaptation framework for serving serverless workflows. The goal of Janus is to maximize resource efficiency while limiting SLO violations. Janus achieves this goal via a bilaterally engaged approach to combat the information barrier between the application developer and the serverless platform provider.

A. Overview

Figure 3 depicts an overview of the system architecture. Janus consists of three core components: *profiler*, *synthesizer*, and *adapter*. Specifically, the profiler and synthesizer are deployed on the developer side, which are run offline, while the adapter runs online on the provider side at runtime.

The general procedure of Janus is as follows: First, the profiler interacts with the developer to collect the domain knowledge of the application, such as the workflow structure, constitutional functions execution time under varying CPU cores and concurrency settings (i.e., batch sizes), and SLO requirements. Afterwards, the profiler extracts functions' execution time distribution (to be used as the profiles) from the collected data using different percentiles. Then, the synthesizer takes the profiles and generates the hints table, which contains rules and options for runtime resource adaptation. This table is submitted to the adapter on the serverless platform. During the execution of the serverless workflow, when a function finishes, the serverless platform collects the execution time of that function and derives the time budget for the rest of the workflow. This derived time budget is reported to the adapter, which then searches in the received hints table and notifies the platform about the adaptation decision for downstream functions. In addition, the adapter plays the role as supervisor who carefully monitors the number of table hit/miss rates. If the miss rate exceeds a predefined threshold, the adapter sends feedback to the developer.

Note that the developer and the provider do not generally require online, continuous interaction. The coordination happens mostly only at the beginning of workflow deployment. It is expected that the submitted hints will be effective throughout the execution of the workflow. This is because the hints table contains fine-grained entries for time budgets produced by a comprehensive exploration of the synthesizer (detailed in §IV-A). In very rare cases where hints table misses are severe (i.e., the miss rate exceeds a given threshold), the adapter notifies the developer and proposes re-triggering the profiler and synthesizer to regenerate the hints table. This regeneration process is done *asynchronously* while workflow execution is still in progress, albeit with sub-optimal adaptation decisions from the adapter (explained in §III-D).

Janus performs per-workflow resource adaptation that restricts the exploitation of runtime slack within the same workflow. While this design may miss some cross-workflow optimization opportunities, it allows Janus to easily support complex scenarios, such as those involving highly parallel workflows. In a multi-user scenario, the hints are managed separately for each tenant and each workflow.

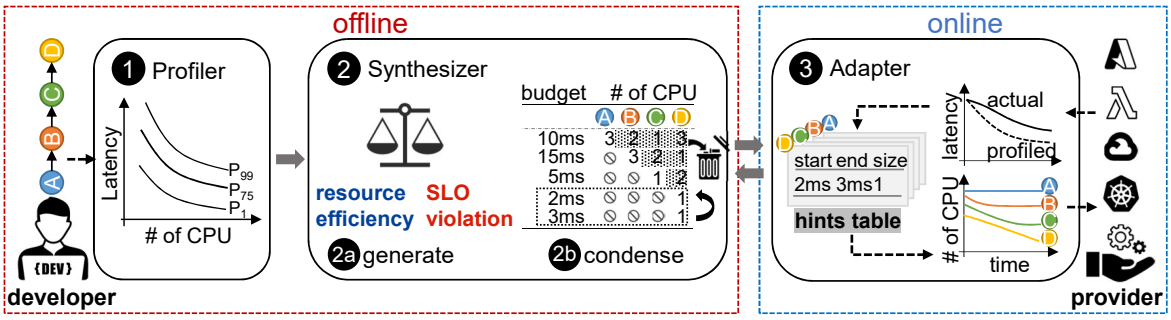


Fig. 3: An overview of the system architecture of Janus. The proposed runtime resource adaptation framework bilaterally engages the application developer and the serverless platform provider, where the developer is responsible for the offline part while the provider is responsible for the online part.

B. Profiler

The profiler is responsible for collecting the execution time of functions under varying resources (i.e., CPU cores) and concurrency levels (i.e., batch sizes) while extracting execution time distribution by using different percentiles. The percentiles can be configured based on SLO requirements. By default, we follow the widely used approach of meeting the end-to-end latency target at the 99th percentile (P99) as latency SLO [6], [7]. Therefore, we use percentiles ranging from 1% to 99% with a step of 5% and the latency profiling of functions is done between P1 and P99. Latency numbers out of the P1-P99 range are not accounted for by Janus for optimization. Janus can accommodate more stringent SLO targets (e.g., at P99.9) by instructing the profiler and synthesizer to use higher percentiles (P99.9).

The diversity in percentiles brings more opportunities to achieve higher resource efficiency but comes at a higher risk of SLO violations. Specifically, when setting percentiles lower than 99%, it may cause under-estimation of function execution time, making functions prone to over-time execution, i.e., their actual execution time exceeds the profiled execution time. To quantify the degree of potential over-time execution, we propose a metric called *timeout*, which is expressed as

$$D(p,k) = L(99,k) - L(p,k), \tag{1}$$

where $L(p,k)$ represents the profiled execution time, with percentiles and CPU cores denoted as k and p, respectively. For preventing SLO violations, Janus must provision more processing resources for downstream functions to absorb such timeouts. To this end, we propose a metric called *resilience* to quantify the absorption capability, which is expressed as

$$R(p,k) = L(p,K_{max}) - L(p,k), \tag{2}$$

where K_{max} denotes the maximum available resources. Any timeout must be restricted within the upper bound of resilience, such that guaranteeing the SLO is still possible.

C. Synthesizer

The synthesizer provides the intelligence of the system by generating and condensing hints in the form of a table. The goal of the synthesizer is to produce hints with high hit rates and maximum resource efficiency. To this end, the synthesizer evaluates potential time budgets across a broad range considering achievable execution time of individual functions. Based on that, the synthesizer explores diverse percentiles for functions to enhance their resource efficiency. Moreover, the synthesizer leverages *timeout* and *resilience*—metrics to quantify the risk of SLO violations as detailed in §III-B—to regulate the above exploration, aiming to provide SLO compliance.

On the other hand, to keep hints tables' efficiency in both space and searching, the synthesizer makes full use of the discreteness in resource adaptation to condense the generated hints (detailed in §IV-B). Finally, the synthesizer provides a highly compact hints table with three simple fields: *start*, *end*, and *size*. This means any workflow with their time budget between *start* and *end* should be provisioned with resource amounts as *size*, which can ensure the maximum resource efficiency without violating the available time budget.

D. Adapter

After a function in the workflow finishes, the adapter derives the available time budget for the remaining functions, and searches the hints table accordingly to figure out the appropriate resource allocation, such that the required time budget can be met with the minimum resource consumption. If the above search results in a miss possibly due to unexpected runtime dynamics (detailed in §II-B), the adapter will scale functions up to the maximum available resources, to prevent SLO violations. Afterwards, the adapter notifies the platform about the adaptation decision. This highly streamlined decision-making process enhances Janus's scalability.

On the other hand, the adapter continuously counts the hits and misses during hint table searches. In rare cases where the miss rate exceeds a predefined threshold, it assumes that the execution time distribution may have changed. In that case, the adapter notifies the developer and suggests triggering the profiler and synthesizer to regenerate hints tables asynchronously. This asynchronous regeneration can strike the trade-off between resource- and time-efficiency in adaptation.

IV. SYNTHESIZER

We now elaborate on the workings of the synthesizer. The hints synthesis process consists of two steps: hints generation and hints condensing.

A. Hints Generation

To generate hints tables with high hit rates and high resource efficiency, the synthesizer requires a twofold effort. First, it must explore all potential runtime time budgets for individual sub-workflows. Second, the synthesizer needs to balance the trade-off between higher resource efficiency and the risk of SLO violation. To this end, we reveal the following insights.

Insight-1: broad time budget range. The time budgets are calculated based on all possibilities between the 1st and 99th percentile (P1-P99) of the function execution time under a wide range of resource allocations, aiming to achieve high hit rates. The range of time budgets therefore are formulated as

$$T_{min} = \sum_{i=1}^{N} L_i(1, K_{max}), T_{max} = \sum_{i=1}^{N} L_i(99, K_{min}), \quad (3)$$

where K_{min}/K_{max} represents the minimum/maximum available resources, and N represents the numbers of functions in the given sub-workflows. Within this range, the synthesizer explores the potential time budgets with finer granularity in milliseconds, while evaluating their corresponding resource allocation. The synthesizer can also be configured with higher percentiles (e.g., P99.9) to meet more stringent SLO targets.

Insight-2: Moderate percentile exploration. Diverse percentiles provide more opportunities for resource optimization, but come with exponentially higher time complexity for runtime resource adaptation. Here, our insight is to only open percentile exploration for the head function of the current sub-workflow while fixing other functions with P99. This moderate percentile exploration benefits the synthesizer with higher resource efficiency, derived from its attempt at lower percentiles for the head function. Meanwhile, it effectively reduces the search space for non-head functions, allowing the synthesizer to achieve high time efficiency.

Insight-3: Resilience-aware. Despite the potential of higher resource efficiency, diverse percentile exploration may put functions at the risk of timeouts, making workflows prone to SLO violations. To address this shortcoming, the synthesizer strictly restricts the timeout within the resilience (the achievable reduction in function execution time by scaling resource up to the maximum possible). Within this "safety zone", the synthesizer tries its best to maximize resource efficiency.

Insight-4: Heavier head. As explained in §II-C, facing substantial variability of execution performance, runtime resource adaptation requires to carry out (head) function by (head) function, so as to keep its high accuracy. This, however, may lead to sub-optimal decisions due to the mismatch between the local objective and the global objective. Specifically, the local objective is to maximize the sub-workflow's resource efficiency, while the global objective is to maximize the whole workflow's resource efficiency. The whole efficiency is determined by that of each sub-workflow's head function, rather than that of sub-workflows. To address this issue, the synthesizer magnifies the local objective's weight for head functions, aiming to calibrate for the mismatch.

As for how to set the weight, our insight is to increase the weight when facing loose SLOs, and vice versa. This is because loose SLOs indicate lower resource requirements, which brings about higher resilience (depicted in Figure 7b). Increasing the weight can better utilize this higher resilience to explore lower percentiles, such that the workflow achieves higher resource efficiency with SLO guarantees.

Hints demonstrates explicit resource allocation that can ensure the sub-workflow with its maximum resource efficiency, i.e., the minimum resource consumption, under given time budgets. This problem thus is formulated as follows:

$$\min \quad Wk_1 + p\sum_{i=2}^{N} k_i + (1-p)(N-1)K_{max} \quad (4)$$

$$\text{subject to} \quad L_1(p, k_1) + \sum_{i=2}^{N} L_i(99, k_i) \le T, \quad (5)$$

$$D_1(p, k_1) \le \sum_{i=2}^{N} R_i(99, k_i), \quad (6)$$

$$1 \le p \le 99,\ p \in \mathbb{Z}, \quad (7)$$

$$K_{min} \le k_i \le K_{max},\ k_i \in \mathbb{R},\ \forall i. \quad (8)$$

where W is the weight for the head function (Insight-4), and T and N denote the time budget and the number of functions in the sub-workflow, respectively. Notably, only the head function can explore lower percentile p (Insight-2). Equation 4 expresses the sub-workflow's expected resource consumption. Specifically, $\sum_{i=2}^{N} k_i$ and $(N-1)K_{max}$ denote non-head functions' resource requirement without and with the head function's timeout, the probability of which is p and $1-p$, respectively. Equation 5 ensures the sub-workflow's execution latency within the time budget. Equation 6 restricts that the possible timeout of the head function can not exceed the total resilience of downstream functions (Insight-3).

The algorithm for generating hints is listed in Algorithm 1. To ensure hints tables with high hit rates, the synthesizer explores all time budgets comprehensively (lines 2–4). Specifically, for a given sub-workflow $\mathbf{F}$, the synthesizer first determines the percentiles $\mathbf{P}$ that can ensure $\mathbf{F}$'s execution time below the required time budget t, with assuming the maximum available CPU cores for each function (lines 8–9). Then, the synthesizer explores the resource allocation for both head and non-head functions, denoted as k and $\mathbf{Z}$, under given percentile p. Its goal is to minimize the expected resource consumption s, while promising timeout $D(p, k)$ restricted within resilience $\sum R(\mathbf{Z}, P_{99})$ (lines 12–17). To accelerate the generation, the synthesizer explores different percentiles concurrently.

B. Hints Condensing

The synthesizer fully utilizes the discreteness in both decision-making and decision-executing to condense hints.

Algorithm 1: Offline hints generation

Input: $\mathbf{F} = \langle f_1, \ldots, f_N \rangle$: (sub-)workflow
Input: $[T_{min}, T_{max}]$: time budget range
Input: $W, \mathbf{P}$: weight and candidate percentiles for head function f_1
Output: $\mathbf{H} = \{\langle t, \{k_1, \ldots, k_N\}\rangle\}$: functions' provisioned CPU cores under given time budget t, i.e., hints table

1 $\mathbf{H} \leftarrow \emptyset$, $\mathbf{P} \leftarrow \emptyset$
2 **foreach** $t \in [T_{min}, T_{max}]$ **do**
3 $\mathbf{H} \leftarrow \mathbf{H} \cup \{\langle t,$ `generate`$(\mathbf{F}, t, \mathbf{P})\rangle\}$
4 return $\mathbf{H}$
5 **Function** `generate`$(\mathbf{F}, t, \mathbf{P})$**:**
6 **if** $|\mathbf{F}| = 1$ **then**
7 **return** `min_resource`(f_1, t)
8 **if** $\mathbf{P} = \emptyset$ **then**
9 $\mathbf{P} =$`explore_percentile`$(\mathbf{F}, t, K_{max})$
10 $s_{min} \leftarrow \infty$, $\mathbf{K} \leftarrow \emptyset$
11 **foreach** $p \in \mathbf{P}$ **do**
12 **foreach** $k \in [K_{min}, K_{max}]$ **do**
13 $\mathbf{Z} \leftarrow$ `generate`$(\mathbf{F} \setminus f_1, t - L_1(p,k), \{P_{99}\})$
14 **if** $\mathbf{Z} \neq \emptyset \wedge D(p,k) \leq \sum R(\mathbf{Z}, P_{99})$ **then**
15 $s \leftarrow Wk + p\sum \mathbf{Z} + (1-p)(|\mathbf{F}| - 1)K_{max}$
16 **if** $s \leq s_{min}$ **then**
17 $s_{min} \leftarrow s, \mathbf{K} \leftarrow \{k\} \cup \mathbf{Z}$
18 return $\mathbf{K}$

Algorithm 2: Offline hints condensing

Input: $\mathbf{H} = \{\langle t, \mathbf{K}\rangle\}$: raw hints table
Output: $\mathbf{U} = [\langle T_{start}, T_{end}, k\rangle]$: condensed hints table

1 **Function** `condense`$(\mathbf{H})$**:**
2 $\mathbf{H} \leftarrow$ `sort`$(\mathbf{H})$
3 $\mathbf{U} \leftarrow \emptyset$, $q, i, j \leftarrow 0$
4 **foreach** $l \in [0, |\mathbf{H}|]$ **do**
5 $t, \langle k_1, \ldots, k_N\rangle \leftarrow \mathbf{H}[l]$
6 **if** $q = 0 \vee k_1 = q$ **then**
7 $j \leftarrow j + 1$
8 **else**
9 $\mathbf{U} \leftarrow \mathbf{U} \cup \{\langle \mathbf{H}[i].t, \mathbf{H}[j].t, q\rangle\}$
10 $i, j \leftarrow l$, $q \leftarrow k_1$
11 return $\mathbf{U}$

Insight-5: Repeated hints. There are various discrete variables, such as batch sizes and CPU cores, involved in resource adaptation. This leads to a significant number of redundant hints that share the same adaptation decisions despite having different time budgets.

Insight-6: Unused fields. The dependencies of adaptation (explained in §II-C) compels Janus to rely solely on the fields for head functions in given hints to maintain adaptation accuracy. Consequently, removing the fields for non-head functions helps compact the hints without compromising accuracy.

The algorithm for condensing hints is listed in Algorithm 2. Specifically, the synthesizer first sorts the given hints $\mathbf{H}$ in descending order by their time budget (line 2). Then, it gradually fuses the hints $\mathbf{H}[l]$ that share the identical size for head function k_1 as shown in line 4–10. Finally, it warps hints into a table with three fields: T_{start}, T_{end}, and k, indicating that the head function of the target sub-workflow should be resized to k when the sub-workflow's time budgets is between T_{start} and T_{end}.

In addition, the weight for head functions impacts the decision-making. Thus, the synthesizer maintains individual hint tables for different weights. We will evaluate the effectiveness of condensing algorithm in §V-F, which suggests a outstanding compression ratio without hurting accuracy.

V. Evaluation

A. Setup and Implementation

Testbed. Our system uses a server equipped with Intel(R) Xeon(R) CPU E5-2678 v3 2.50GHz with 24 physical CPU cores as the local server running Ubuntu 18.04, where Janus synthesizes hints tables. Meanwhile, we use another server equipped with Intel(R) Xeon(R) Platinum 8269CY CPU 2.50GHz with 52 physical CPU cores, running Ubuntu 18.04, as a serverless platform. In this platform, we implement Janus into an open-source framework as Fission [42] (V1.16) for serverless functions on Kubernetes. We use Fission Pool-Manager [43] to spin up function pods, due to its excellent performance against cold starts.

Implementation. Janus has a frontend side and a (remote) backend side. To facilitate seamless coordination, we implement the profiler and synthesizer as two distinct functions on the frontend side, while deploying the adapter as a service on the backend side. Moreover, the adapter can be equipped with automatic horizontal scaling for enhancing Janus's scalability. The frontend interacts with the developer and depends on their domain knowledge (detailed below) to synthesize hints tables. We leverage packages `pandas.DataFrame` to represent hints tables. As for the backend side, we develop a lightweight server using Python Flask [44], Redis [45], Fission APIs [46], and Fission HTTP trigger [47]. The server spawns a process to trace each request's execution. Upon completion of any function in the workflow, this process will re-evaluate the time budget for the remaining functions while accessing hints tables to decide on proper resource adaptation.

Workflows. We evaluate the effectiveness of Janus with two real-world serverless workflows namely Intelligent Assistant (IA) and Video Analyze (VA). Specifically, IA is a chain constituted by three functions: *object detection* (OD, for short) [48], *question answer* (QA) [49], and *text-to-speech* (TS) [50], which analyzes images randomly sampled from COCO2014 [51] and answers questions sampled from SQuAD2.0 [52]; finally, the answers return in the form as audios. VA as another workflow chain includes three functions as *frame extraction* (FE) [53], *image classification* (ICL) [54], and *image compression* (ICO) [55]. Its inputs are YouTube videos with identical duration and resolution, sourced from ORION [6]. Additionally, the four functions in §II-B as *CPU-*, *Memory-*, *Network-*, and *IO-intensive* conduct AES encryption [17], data read (from a Redis based in-memory

database) [56], socket communication [56], and data write (to local disks) [36], respectively.

Runtime dynamics. Our testing workloads contain runtime dynamics, encompassing varying working sets and performance interference. Specifically, the input data for IA, i.e., images and texts from COCO2014 and SQuAD2.0 respectively, are with varying working sets. The empirical study shows that the number of objects per image in COCO2014 ranges from 1 to 15 [57], while the number of words per text in SQuAD2.0 ranges from 35 to 641. As shown in Figure 1b, these varying sets result in significant variance in function execution time. On the other hand, VA extracts frames from videos, followed by image classification and compression. To accelerate processing, VA implements parallelism for each function, incurring cross-function performance interference inevitably. The profiles reveal that, for the three functions in VA, the average ratio of execution time between P99 and P50 is 1.46 times, 1.56 times, and 1.37 times, respectively.

Domain knowledge. We collect the execution time of IA's and VA's functions with respect to CPU cores, ranging from 1000 millicores to 3000 millicores with a step of 100 millicores. After data collection, Janus adopts diverse percentiles, ranging from 1% to 99% with a step as 5%, to profile execution time distribution. To assess Janus's performance over higher loads, we additionally profile IA's execution performance over higher concurrency (i.e., batch size) as two and three. As for VA, we only profile its performance with concurrency as one because FE and ICO cannot process frames in batch form. Here, we exclude memory as a knob. This is because Janus focuses on latency-critical workflows. Our empirical tests show that memory has no impact on execution time.

Baselines. Janus proposes bilaterally engaged resource adaptation to provide efficient serverless workflows serving, aiming to maximize resource efficiency, i.e., minimize resource consumption, with SLO guarantees. Here, we use three early-binding approaches and three late-binding approaches as our baselines. The early-binding approaches include the state-of-the-art serverless workflow serving system ORION [6], GrandSLAM [41] and its enhanced version GrandSLAM+. Specifically, GrandSLAM+ improves GrandSLAM by removing the latter's constraints in identical sizes for all functions. The late-binding approaches include Janus−, Janus+, and Optimal. Optimal represents the best that can be achieved in any late-binding solution.

The differences between Janus, Janus-, and Janus+ are as follows: Janus allows exploring diverse percentiles for the head (first) function in workflows. Janus− disables this exploration and adopts a fixed percentile, P99. Janus+ extends the exploration to both the head function and the next-to-head function. In summary, compared to Janus, Janus− has compromised resource efficiency due to its smaller optimization space. Janus+ can have higher resource efficiency (e.g., 0.6% higher than Janus for IA) owing to a larger optimization space but at the expense of considerable time cost (by up to 107.2 times) in synthesizing hints (§V-C).

Notably, existing late-binding approaches including Fifer, Kraken, Xanadu, and Cypress [9] mostly overlook the information barrier between the developer and provider, raising practicality concerns. Additionally, as highlighted by Cypress, Fifer, Kraken, and Xanadu assume that function execution time does not have large variance, and hence they adopt mean execution time to perform runtime resource adaptation. However, this assumption contradicts our empirical observations from serverless production traces, which exhibit significant variance in execution time (§II). Consequently, these approaches are easily prone to under provisioning and severe SLO violations. Thus, we exclude them as the baselines.

TABLE I: Overall resource reduction (normalized by Optimal) by Janus compared to baselines when serving IA and VA, respectively.

	ORION	**GrandSLAM+**	**GrandSLAM**	**Janus−**	**Janus+**
IA(%)	22.6	31.3	31.3	2.9	0
VA(%)	26.9	35.2	32.4	4.7	-0.2

Setup. Considering our testbed's capacity and the short-lived nature of functions [23], [40], we set SLOs for IA and VA as 3s and 1.5s, respectively. We set the weight for each function as one unless otherwise specified. We explore Janus's performance under varying SLOs and weights in §V-G and §V-E, respectively. When a hints table miss occurs, we scale functions up to 3000 millicores to prevent SLO violations. The miss rate threshold is set as 1% by default. To ensure experimental results' statistical significance, we evaluate the performance of Janus and baselines over 1000 requests.

B. Overall Performance

End-to-end latency distribution. Figure 4 shows the end-to-end latency (E2E) distribution of IA and VA under the concurrency as one, as well as that of IA under the concurrency as two and three respectively. We observe that Janus can fulfill the SLO requirements in all cases despite relatively higher E2E. This is because Janus aims to improve resource efficiency while meeting latency SLOs. Under the premise of fulfilling SLOs, Janus trades in time for resource efficiency.

Resource consumption. We compare the resource consumption of Janus and baselines when serving IA and VA given SLO as 3s and 1.5s respectively, with the concurrency as one. Table I shows the average resource reduction of Janus, normalized by Optimal, when compared with baselines, and Figure 5a illustrates the detailed comparison. We can see that Janus outperforms GrandSLAM+, GrandSLAM, and ORION significantly. This is because Janus fully uses slacks at runtime to improve resource efficiency. Compared with Janus−, Janus achieves further resource reduction as 2.9% and 4.7% for IV and VA respectively, due to Janus's exploration of lower percentiles for head functions. Additionally, Janus incurs a negligible increase in resource consumption, i.e., 0.2%, compared to Janus+.

We also assess Janus's performance under higher loads. Here, we increase IA's concurrency up to two and three. For a

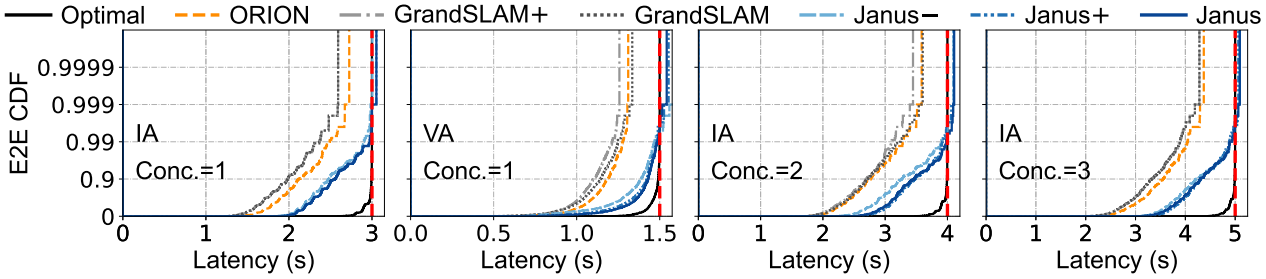


Fig. 4: End-to-end latency distribution of IA under the concurrency (i.e., batch size) as one, two and three respectively, with different SLOs (red dashed line). The concurrency of VA is limited to one due to its non-batchable functions (i.e., FE and ICO).

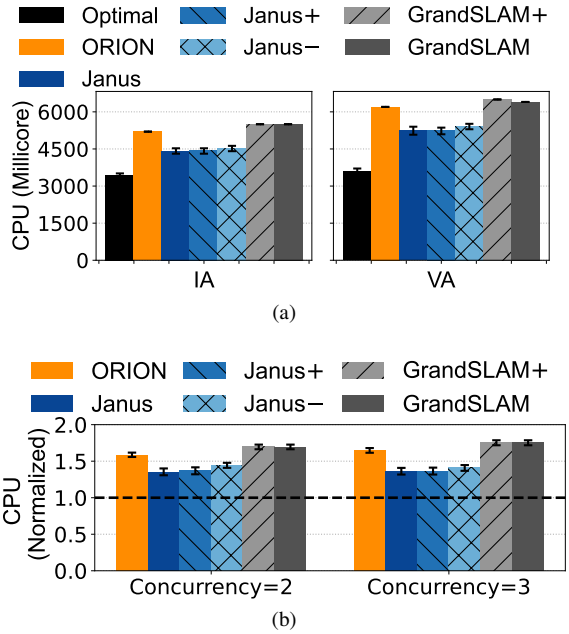


Fig. 5: Resource consumption of (a) IA (left) and VA (right) under the concurrency as one, respectively, and of (b) IA under the concurrency as two (left) and three (right), respectively.

fair comparison, we increase SLOs to 4s and 5s respectively, to promise GrandSLAM and GrandSLAM+ with feasible function sizes. Figure 5b shows the resource consumption normalized by Optimal. We find that the three early-binding systems, i.e., GrandSLAM, GrandSLAM+, and ORION, suffer over-allocation by up to 1.75 times. This is because the increase of the concurrency further enlarges the runtime variability. For example, the gap between P99 and P50 of QA (the second function in IA) increases from 2.17 times to 2.32 times on average. This higher variability magnifies the early-binding's over-provisioning. As a contrast, Janus relies on its runtime adaptation to capture the variance and resize functions correspondingly, thus reaping higher resource efficiency.

C. Effectiveness of Moderate Percentile Exploration

Here, we assess the effectiveness of the moderate exploration approach (§IV-A) adopted by Janus, which restricts

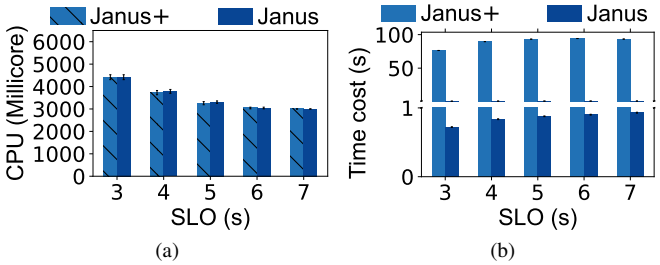


Fig. 6: (a) Workflow sizes and (b) time costs of Janus+ and Janus respectively, with SLOs ranging from 3s to 7s.

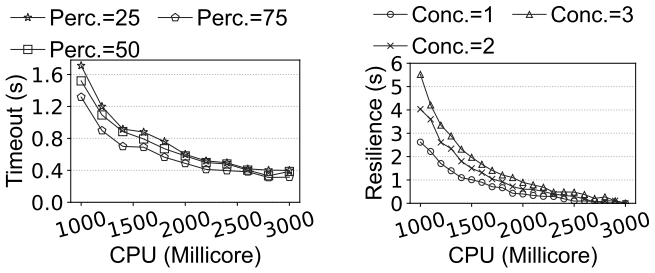
the exploration of lower percentiles (below P99) to head functions within sub-workflows. This strategy aims to balance the trade-off between the time- and resource-efficiency of resource adaptation. We compare Janus with Janus+, which extends percentile exploration to both the head function and the next-to-head function. While this expansion of percentile exploration can yield higher resource efficiency, it comes at the expense of significant time cost in synthesizing hints tables.

Taking IA as an instance, we observe from Figure 6a that compared with Janus, Janus+ decreases resource consumption merely by 0.6% on average. This means the optimization space of wider percentile exploration for IA is limited. However, this limited reduction in resource comes at a significant time cost in synthesizing hints, by up to 107.2 times higher than Janus, as depicted in Figure 6b.

Additionally, Janus's time costs increases marginally as SLO grows. This is because higher SLOs brings in more candidate adaptation plans. Janus needs to efficiently evaluate these plans, and figure out the one with the minimum resource consumption, thus incurring higher time costs. Notably, the above time costs only happen during hints generation. When coming to online adaptation, its overhead is merely less than 3ms (explained in §V-H).

D. Timeout and Resilience

We propose timeout and resilience to quantify the risk of SLO violations (detailed in §IV-A). Owing to space constraints, we use TS from IA as an example, and other



(a) (b)

Fig. 7: (a) Timeout and (b) resilience of TS under varying CPU cores.

TABLE II: Resource consumption and percentiles for the head function of IA with the weight as one and three, respectively.

	weight=1	**weight=3**
CPU (Millicore)	1442.9	1228.6
Percentile (%)	94.4	91.3

functions exhibit similar patterns. We observe from Figure 7a that timeout decreases as either percentiles or available CPU cores increase. This is because additional resources enhance functions' capability to handle both runtime interference and variability of working sets [35], thus reaping lower timeout.

As for resilience, Figure 7b shows a marginal reduction as the number of provisioned CPU cores increase. This is attributed to non-parallelizable operations within functions, leading to diminishing returns on execution time despite the addition of more resources. Additionally, higher concurrency enhances higher resilience. This is due to the increased computing load, which heightens functions' sensitivity to resources, thereby boosting resilience.

E. Impact of Weight

Higher weights for head functions is introduced to further improve the resource efficiency of Janus (detailed in §IV-A). Taking IA as an example, we evaluate its resource consumption with SLOs ranging from 4s to 10s under the weight as one and three, respectively. The results show that when the SLO is less than 8s the moderate weight consumes less resources by 2.9% on average. Conversely, as the SLO becomes relaxed, the higher weight allows to further reduce resource by 1% owing to its larger optimization space.

We also examine the impact of weights on resource consumption and percentile selection for head functions. Table II shows that Janus tends to decrease both resource allocation and percentiles under higher weights. This is because with higher weights the objective focuses more on decreasing the size of head functions, rather than that of sub-workflows. Lower percentiles typically indicate that fewer requests need to be completed within the specified time budget, thus requiring fewer resource consumption. This aligns well with the objective with higher weights. Yet, lower percentiles may expose

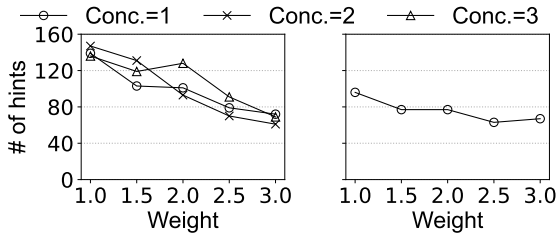


Fig. 8: Total numbers of hints synthesized for IA (left) and VA (right) under different weights.

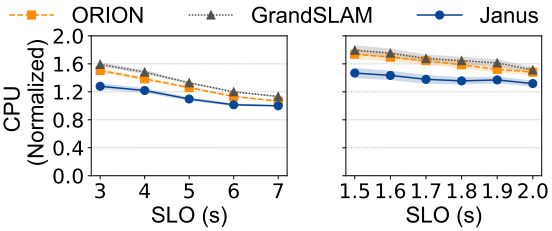


Fig. 9: IA's (left) and VA's (right) resource consumption (normalized by Optimal) under different SLOs.

sub-workflows at the risk of timeouts, particularly under strict SLOs. To prevent SLO violations, non-head functions may compensate by requesting additional resources, potentially hurting the overall resource efficiency.

F. Effectiveness of Hints Condensing

We assess hints table sizes, i.e., numbers of hints, with and without condensing. As explained in §IV-A we depend on our testbed's capacity to configure the range of time budgets explored during hints generation. Specifically, for IA the range is from 2s to 7s, from 3s to 7s, and from 4s to 10s, with a fine-grain step of 1ms, under the three different concurrency, respectively. For VA this range is from 1.5s to 2s with a step of 1ms. Additionally, weight, as a hyper-parameter involving hint generation, also influences hints table sizes. Therefore, we assess the two workflows' hints table sizes under different weights ranging from 1 to 3 with a step as 0.5, respectively.

Figure 8 illustrates the detailed number of hints for IA (left) and VA (right), respectively. After effective condensing, the overall hints for IA and VA are less than 147 and 96 respectively, achieving compression ratios of up to 99.6% and 98.2%. In addition, the size of hints tables decreases as the weight increases. The reason is that higher weights focus more on minimizing the size of the head function, which may lead to the sub-workflow's over-allocation. This over-allocation increases hints' applicability across different runtime conditions, thus benefiting hints tables with smaller sizes.

G. Impact of SLO

We compare Janus's resource consumption with baselines when serving IA and VA under varying SLOs. For clarity, we normalize the results by Optimal. To ensure readability, we illustrate only the results of ORION, GrandSLAM, and Janus.

As shown in Figure 9, for IA Janus outperforms ORION and GrandSLAM by 16.1% and 24.1% on average, respectively. In terms of VA, Janus outperforms the baselines by 22.2% and 27.7%, respectively. Notably, as SLOs increase, Janus's performance gains decrease marginally. This is due to our testbed's limitation of CPU cores, i.e., 1000 millicores at least per function, restricting Janus's further improvement. For instance, under given SLOs as 6s and 7s, IA's resource consumption reduces to 3043.6 millicores, approaching that of Optimal (i.e., 3000 millicores). As for other baselines, GrandSLAM+ exhibits performance comparable to GrandSLAM, with a marginal gap of less than 0.6%. Janus+ achieves a resource reduction of up to 1.8% compared to Janus. Janus− incurs higher resource consumption, exceeding Janus by 3.2% and 4.3% on average, when serving IA and VA respectively.

H. System Overhead

We evaluate Janus's time cost for online resource adaptation serving IA and VA respectively, under varying SLOs from 2s to 7s with the weight as one and three, respectively. The results show that the time cost remains under 3ms. This suggests that Janus maintains high time-efficiency unaffected by either SLOs or weights.

We measure the memory footprint of Janus during online adaptation and offline hints generation. As for online adaption, Janus consumes negligible memory less than 12.1MB and 10.9MB for IA and VA, respectively. In terms of offline hints generation, the average memory consumption is less than 12.4MB and 10.9MB for IA and VA, respectively.

VI. Related Work

We summarize related work covering both early-binding and late-binding approaches for serverless resource management.

A. Early Binding

COSE [58], Sizeless [59], and Parrotfish [8] adopt machine learning to learn the cost/performance of functions with respect to different sizes, and then select "best" function sizes, such that overall costs can be minimized without violating SLOs. FA2 [60] fully considers the dependency of functions within a workflow and their uncertain execution paths to periodically adapt resources, aiming to minimize resource consumption while promising SLAs. Aquatope [15] considers runtime performance interference to decide function sizes. ORION [6] and WISEFUSE [7] observe skewed function execution latency and develop a distribution-based performance modeling to provision serverless DAGs. GrandSLAM [41] provisions functions with fixed and identical sizes, while dynamically batching and reordering requests within each function by considering runtime slacks, aiming to achieve higher throughput without violating SLOs. Additionally, Morhpling [4] and INFaaS [2] focus on resource auto-configuration for ML-inference specific systems. On the other hand, there exists research from the industry that helps developers decide function sizes, such as AWS Lambda Power Tuning [61] and AWS Compute Optimizer [62].

B. Late Binding

Cirrus [63] proposes a serverless framework to boost the performance of best-efforts tasks (i.e., ML training), by integrating a client-side to monitor the remote execution while adjusting its resource allocation. Fifer [28] leverages the slacks, generated at each stage, to adjust batch sizes and scale out/in containers for higher resource utilization with SLO guarantees. Atoll [64] enables proactive resource scaling as well as deadline-aware scheduling to minimize SLO violations. BATCH [29] fully considers serverless workload burstiness (the intensity of arrival requests) to dynamically adjust function size (memory size) and batching parameters, for the sake of minimizing monetary cost without violating SLOs. Kraken [26] and Xanadu [27] employ proactive and reactive resource scalers simultaneously to provision dynamic DAG workloads, which have uncertain execution paths, aiming to minimize resource consumption without SLO violations. Cypress [9] enables input size-aware request batching and resource provisioning. Llama [2] focuses on auto-tuning video analytics pipelines under heterogeneous serverless environments. Erms [65], FIRM [16], and Sinan [66] focus on improving resource efficiency without violating SLOs, for shared microservices. Apart from auto-scaling, there are works focusing on serverless workflows scheduling [67]–[74] and over-commit [35], [75].

VII. Conclusion

In this paper, we identified the resource inefficiency lying in the early-binding based resource allocation for serverless workflows, and proposed a late-binding approach to address it by promoting bilateral runtime resource adaptation engaging both the developer and the provider. Based on this concept, we proposed Janus—a novel resource adaptation framework for serverless workflows. We identified the challenges in building Janus and proposed efficient algorithms for fine-grained resource allocation for Janus. Experiments based on a system prototype show that Janus achieves significant resource savings while providing latency SLO guarantee. Future work includes adding support for more complex workflows and exploring the impact of the runtime resource adaptation on function caching strategies.

Acknowledgment

This work was supported in part by National Science Foundation of China under grant 62232012, in part by National Key Research & Development (R&D) Plan under grant 2022YFB4501703, in part by the Major Key Project of PCL under Grant PCL2024A06 and PCL2022A05, and in part by the Shenzhen Science and Technology Program under Grant RCJC20231211085918010.

References

[1] Z. Zhao, M. Wu, J. Tang, B. Zang, Z. Wang, and H. Chen, "Beehive: Sub-second elasticity for web services with semi-faas execution," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 2023, pp. 74–87.

[2] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*. ACM, 2021, pp. 1–17.

[3] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 2023, pp. 1489–1504.

[4] L. Wang, L. Yang, Y. Yu, W. Wang, B. Li, X. Sun, J. He, and L. Zhang, "Morphling: Fast, near-optimal auto-configuration for cloud-native model serving," in *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*. ACM, 2021, pp. 639–653.

[5] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 2021, pp. 397–411.

[6] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless dags," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, M. K. Aguilera and H. Weatherspoon, Eds. USENIX Association, 2022, pp. 303–320. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/mahgoub

[7] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "WISEFUSE: workload characterization and DAG transformation for serverless workflows," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, pp. 26:1–26:28, 2022. [Online]. Available: https://doi.org/10.1145/3530892

[8] A. Moghimi, J. Hattori, A. Li, M. B. Chikha, and M. Shahrad, "Parrotfish: Parametric regression for optimizing serverless functions," in *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*. ACM, 2023, pp. 177–192.

[9] V. M. Bhasi, J. R. Gunasekaran, A. Sharma, M. T. Kandemir, and C. R. Das, "Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms," in *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, A. Gavrilovska, D. Altinbüken, and C. Binnig, Eds. ACM, 2022, pp. 257–272. [Online]. Available: https://doi.org/10.1145/3542929.3563464

[10] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. D. Palma, B. Batchakui, and A. Tchana, "OFC: an opportunistic caching system for faas platforms," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 2021, pp. 228–244.

[11] W. Xiao, Y. Hao, J. Liang, L. Hu, S. A. AlQahtani, and M. Chen, "Adaptive compression offloading and resource allocation for edge vision computing," *IEEE Transaction. Cogn. Commun. Netw.*, vol. 10, no. 6, pp. 2357–2369, 2024.

[12] S. Chen, L. Wang, and F. Liu, "Optimal admission control mechanism design for time-sensitive services in edge computing," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, London, United Kingdom, May 2-5, 2022*. IEEE, 2022, pp. 1169–1178.

[13] B. Xia, C. Wong, Q. Peng, W. Yuan, and X. You, "Cscnet: Contextual semantic consistency network for trajectory prediction in crowded spaces," *Pattern Recognition*, vol. 126, p. 108552, 2022.

[14] C. Wong, B. Xia, Z. Zou, Y. Wang, and X. You, "Socialcircle: Learning the angle-based social interaction representation for pedestrian trajectory prediction," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*. IEEE, 2024, pp. 19 005–19 015.

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

[16] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: an intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 805–825. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/qiu

[17] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: characterization and optimization," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, V. Salapura, M. Zahran, F. Chong, and L. Tang, Eds. ACM, 2022, pp. 757–770. [Online]. Available: https://doi.org/10.1145/3470496.3527390

[18] A. Suresh and A. Gandhi, "Servermore: Opportunistic execution of serverless functions in the cloud," in *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds. ACM, 2021, pp. 570–584. [Online]. Available: https://doi.org/10.1145/3472883.3486979

[19] R. B. Roy, T. Patel, R. Liew, Y. N. Babuji, R. Chard, and D. Tiwari, "Propack: Executing concurrent serverless functions faster and cheaper," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2023, Orlando, FL, USA, June 16-23, 2023*, A. R. Butt, N. Mi, and K. Chard, Eds. ACM, 2023, pp. 211–224. [Online]. Available: https://doi.org/10.1145/3588195.3592988

[20] W. Xiao, X. Ling, M. Chen, J. Liang, S. A. Alqahtani, and M. Chen, "Mvpoa: A learning-based vehicle proposal offloading for cloud-edge-vehicle networks," *IEEE Internet of Things Journal*, 2024.

[21] S. Ginzburg and M. J. Freedman, "Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms," in *WoSC@Middleware 2020: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Virtual Event / Delft, The Netherlands, December 7-11, 2020*. ACM, 2020, pp. 43–48. [Online]. Available: https://doi.org/10.1145/3429880.3430099

[22] R. Cordingly, S. Xu, and W. Lloyd, "Function memory optimization for heterogeneous serverless platforms with CPU time accounting," in *IEEE International Conference on Cloud Engineering, IC2E 2022, Pacific Grove, CA, USA, September 26-30, 2022*. IEEE, 2022, pp. 104–115. [Online]. Available: https://doi.org/10.1109/IC2E55432.2022.00019

[23] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. N. Darlow, J. Wang, and A. Barker, "How does it function?: Characterizing long-term trends in production serverless workloads," in *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*. ACM, 2023, pp. 443–458. [Online]. Available: https://doi.org/10.1145/3620678.3624783

[24] "Azure monitor," https://learn.microsoft.com/en-us/azure/azure-monitor/overview, 2024.

[25] "Aws cloudwatch," https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html, 2024.

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

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

[28] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. ACM, 2020, pp. 280–295.

[29] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 69. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00073

[30] "Aws step functions," https://aws.amazon.com/step-functions/, 2022.

[31] "Microsoft durable functions," https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc, 2023.

[32] "Azure dataset," https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md, 2023.

[33] "Azure function blob dataset," https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsBlobDataset2020.md, 2023.

[34] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*. ACM, 2021, pp. 122–137. [Online]. Available: https://doi.org/10.1145/3472883.3486974

[35] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, A. Gavrilovska, D. Altinbüken, and C. Binnig, Eds. ACM, 2022, pp. 78–93. [Online]. Available: https://doi.org/10.1145/3542929.3563470

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

[37] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, predicting and scheduling serverless workloads under partial interference," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 2021, p. 22.

[38] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 1063–1075.

[39] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, London, United Kingdom, May 2-5, 2022*. IEEE, 2022, pp. 1868–1877. [Online]. Available: https://doi.org/10.1109/INFOCOM48880.2022.9796962

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

[41] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*. ACM, 2019, pp. 34:1–34:16.

[42] "Fission," https://fission.io/, 2023.

[43] "Fission executor," https://fission.io/docs/architecture/executor/, 2023.

[44] "Python flask," https://flask.palletsprojects.com/en/3.0.x/, 2024.

[45] "Redis," https://redis.io/, 2024.

[46] "Fission cli reference," https://fission.io/docs/reference/fission-cli/, 2024.

[47] "Fission http triggers," https://fission.io/docs/reference/fission-cli/fission_httptrigger/, 2024.

[48] "Object detection," https://pytorch.org/vision/stable/models/generated/torchvision.models.detection.fasterrcnn_mobilenet_v3_large_320_fpn.html, 2024.

[49] "Question answer," https://huggingface.co/distilbert/distilbert-base-uncased-distilled-squad, 2024.

[50] "Text-to-speech," https://huggingface.co/facebook/mms-tts-hat, 2024.

[51] "Coco dataset," https://www.kaggle.com/datasets/jeffaudi/coco-2014-dataset-for-yolov3 , 2024.

[52] "The stanford question answering dataset," https://rajpurkar.github.io/SQuAD-explorer/ , 2024.

[53] "Extract frame," https://ffmpeg.org/, 2024.

[54] "Image classification," https://pytorch.org/vision/main/models/generated/torchvision.models.squeezenet1_1.html, 2024.

[55] "Image compression," https://docs.python.org/3/library/shutil.html, 2023.

[56] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: a serverless benchmark suite for function-as-a-service computing," in *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, K. Zhang, A. Gherbi, N. Venkatasubramanian, and L. Veiga, Eds. ACM, 2021, pp. 64–78. [Online]. Available: https://doi.org/10.1145/3464298.3476133

[57] T. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," in *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V*, ser. Lecture Notes in Computer Science, vol. 8693, 2014, pp. 740–755.

[58] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "COSE: configuring serverless functions using statistical learning," in *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*. IEEE, 2020, pp. 129–138.

[59] S. Eismann, L. Bui, J. Grohmann, C. L. Abad, N. Herbst, and S. Kounev, "Sizeless: predicting the optimal size of serverless functions," in *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*. ACM, 2021, pp. 248–259.

[60] K. Razavi, M. Luthra, B. Koldehofe, M. Mühlhäuser, and L. Wang, "FA2: fast, accurate autoscaling for serving deep learning inference with SLA guarantees," in *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2022, Milano, Italy, May 4-6, 2022*. IEEE, 2022, pp. 146–159. [Online]. Available: https://doi.org/10.1109/RTAS54340.2022.00020

[61] "Aws lambda power tuning," https://github.com/alexcasalboni/aws-lambda-power-tuning, 2022.

[62] "Aws lambda compute optimizer," https://docs.aws.amazon.com/compute-optimizer/latest/ug/requirements.html, 2023.

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

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

[65] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: Efficient resource management for shared microservices with SLA guarantees," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 62–77. [Online]. Available: https://doi.org/10.1145/3567955.3567964

[66] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 167–181. [Online]. Available: https://doi.org/10.1145/3445814.3446693

[67] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," ser. SoCC '20, 2020.

[68] Z. Guo, Z. Blanco, M. Shahrad, Z. Wei, B. Dong, J. Li, I. Pota, H. Xu, and Y. Zhang, "Decomposing and executing serverless applications as resource graphs," 2022.

[69] T. Schirmer, J. Scheuner, T. Pfandzelter, and D. Bermbach, "Fusionize: Improving serverless application performance through feedback-driven function fusion," in *IEEE International Conference on Cloud Engineering, IC2E 2022, Pacific Grove, CA, USA, September 26-30, 2022*. IEEE, 2022, pp. 85–95. [Online]. Available: https://doi.org/10.1109/IC2E55432.2022.00017

[70] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: a scalable and locality-enhanced framework for serverless parallel computing," in *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds. ACM, 2020, pp. 1–15. [Online]. Available: https://doi.org/10.1145/3419111.3421286

[71] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference,*

USENIX ATC 2021, July 14-16, 2021, I. Calciu and G. Kuenning, Eds. USENIX Association, 2021, pp. 285–301. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/mahgoub

[72] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 2021, pp. 805–820.

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

[74] Q. Chen, J. Qian, Y. Che, Z. Lin, J. Wang, J. Zhou, L. Song, Y. Liang, J. Wu, W. Zheng, W. Liu, L. Li, F. Liu, and K. Tan, "Yuanrong: A production general-purpose serverless system for distributed applications in the cloud," in *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*. ACM, 2024, pp. 843–859.

[75] S. Li, W. Wang, J. Yang, G. Chen, and D. Lu, "Golgi: Performance-aware, resource-efficient function scheduling for serverless computing," in *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*. ACM, 2023, pp. 32–47.