This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier

BriskChain: Decentralized Function Composition for High-performance Serverless Computing

KAN WANG[1], JING MA[1], (Member, IEEE) and EDMUND M-K. LAI[1], (Senior Member, IEEE)
[1]School of Engineering, Computer and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand (e-mail: {jing.ma, edmund.lai}@aut.ac.nz)

Corresponding author: Jing Ma (e-mail: jing.ma@aut.ac.nz)

ABSTRACT Serverless computing allows developers to create workflows for complex tasks through the composition of serverless functions. Current serverless workflow engines rely on master-side pattern which do not permit direct interaction between consecutive serverless functions in the workflow. In this paper, a decentralized worker-side pattern is proposed that provides better performance by allowing consecutive functions to flow from the current node to the next without first having to interact with the master controller. It treats the serverless workflow as a whole unit and uses a locality strategy to optimize performance. This approach is implemented in a workflow engine called BriskChain to demonstrate its effectiveness. Experiments using both synthetic and real application workflows show that BriskChain requires significantly reduced runtime overhead compared with OpenWhisk and Apache Composer, two open-source serverless platforms.

INDEX TERMS Serverless workflow; Function as a service; Worker-side pattern

I. INTRODUCTION

Serverless computing is a cloud computing paradigm that offers a promising approach to software development. It is based on functions that are self-contained, stateless, and fine-grained. This approach, also known as Function as a Service (FaaS), is gaining popularity due to its many benefits. Many cloud server companies offer their own proprietary serverless platforms, including Amazon Web Services (AWS) Lambda, Google Cloud Functions, and IBM Cloud Functions, to name just a few. Open-source serverless computing platforms, such as OpenWhisk and OpenFaaS, are also gaining popularity. Some experts predict that serverless computing will dominate the next generation of cloud systems [1].

In serverless computing, developers combine a number of functions into workflows through a workflow engine to create their application logic. A workflow engine manages, schedules, and monitors action tasks according to predefined rules. Currently, most serverless workflow engines rely on a centralized master-side pattern, where a powerful workflow engine acts as the master that schedules and allocates tasks and resources to many workers or serverless function tasks. This pattern has several disadvantages. Most notably, it imposes a high scheduling overhead. Furthermore, it is very inefficient for workflows that involve substantial data movements between serverless functions. This is because Cloud vendors typically impose quotas on the input and output data size for each serverless function, which limits direct data transfer between them. Although this problem could be solved by using external databases for temporary data storage and delivery [2], data transfer is still time-inefficient due to the high latency of database access. Consequently, such workflow engines face challenges in managing intensive interactions between smaller tasks [3].

To address these issues, this article proposes a decentralized worker-side pattern that significantly reduces communication overhead in serverless workflows. With this pattern, each worker task could transmit its state directly to the next worker in the workflow, eliminating the need to route it through the master controller. A decentralized workflow engine called BriskChain has been implemented on the basis of this pattern for latency-sensitive and interactive serverless computing. BriskChain schedules serverless workflows as a whole unit, similar to a single serverless function, which abides by the substitution principle [4]. There is no distinction between serverless function tasks and workflow tasks, which allows the entire workflow be scheduled like a normal serverless function. BriskChain is built on an open-source FaaS system called OpenWhisk. It provides two extensions to OpenWhisk – the BriskChain runtime and the embedded controller. The BriskChain runtime is a containerized sandbox that can handle not only normal serverless function requests, but also functions in the workflow. The embedded

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

controller is responsible for allocating workflow resources. It also implements a close-locality optimization strategy that put sandboxes of the workflow on the same host to remove remote interactions between workflow functions, thus enhancing efficiency.

The contributions of this paper are as follows:

- A decentralized client-side pattern is proposed for serverless workflow that reduces communication overhead and improves data transfer efficiency between workflow nodes.
- A workflow engine, called BriskChain, was implemented based on the worker-side pattern with a runtime and embedded controller as extension to OpenWhisk.
- BriskChain was experimentally tested.

The rest of this paper is organized as follows. A brief review of current practices in serverless computing, focusing on function composition and workflow engines is presented in Section II. It also gives an overview of master-side and worker-side patterns. Section III describes BriskChain, our proposed decentralized serverless workflow in detail. An evaluation of the performance of BriskChain, using both synthetic and real application workflows, is provided in Section IV. This is followed by Section VI that concludes this article with some suggestions for further research.

II. RELATED WORKS

A. SERVERLESS WORKFLOW OPTIMIZATION

Some recent research on serverless computing have focused on optimizing the performance of workflows, particularly in terms of enhancing resource efficiency and reducing latency. Wen et al. presented a system called StepConf that dynamically configures serverless function workflows to meet Service Level Objectives (SLOs) and optimize resource usage [5], [6]. StepConf includes a configuration-aware function prewarming mechanism to reduce cold start overhead. It optimizes memory size for each function step in the workflow and takes inter and intra-function parallelism into consideration. In [7], a retention-aware container caching mechanism is introduced to address the startup latency problem. The proposed Online Retention-aware Distributed Caching (O-RDC) algorithm optimizes both container caching and request distribution across distributed edge nodes, leveraging their heterogeneous resources to reduce overall system costs. By keeping containers in memory for subsequent requests, the approach significantly reduces cold-start latency and enhances the responsiveness of serverless functions.

Heterogeneous (CPU and GPU) batching for multi-SLO Deep Neural Network (DNN) inference has also been explored in complex serverless applications. Chen et al. [8] introduced HarmonyBatch, which reduces the overhead associated with individual function invocations by batching tasks together. This approach balances resource utilization and ensures efficient execution of complex workflows.

These advancements in dynamic configuration, caching mechanisms, resource management, and batching strategies collectively contribute to the optimization of serverless function workflows. By focusing on the entire workflow rather than individual functions, these approaches enhance the efficiency and performance of serverless computing platforms, making them more suitable for complex, multi-function applications. These developments provide a robust foundation for further innovations that aim to fully harness the potential of serverless computing in diverse application domains.

B. MASTER-SIDE AND WORKER-SIDE PATTERNS

Although strategies such as dynamic configuration, caching mechanisms, and resource management play a significant role in serverless workflows, researchers are recently exploring new methods for optimization in this field from the perspective of serverless working patterns. Specifically, in contrast to the traditional centralized master-side workflow engine, they have investigated decentralized worker-side pattern-based workflow engines.

The master-side pattern relies on a centralized, robust workflow engine as the master, responsible for tasks such as scheduling, resource allocation, and task balancing. Consequently, numerous workers (tasks) operate under the management of this master. Currently, the majority of workflow systems adopt the master-side pattern approach [9]–[12]. Related studies include, but are not limited to, works in [10]–[14].

Under the master-side pattern, the workflow engine faces significant scheduling overhead and frequent data interaction pressure from numerous serverless function workers. Each task (worker) within a workflow operates as a serverless function, necessitating frequent communication with its remote master. If a serverless function needs to transfer data to another function, it must first route the data through the master. A study conducted by [15] compared three common communication patterns—broadcast, aggregation, and shuffle—across both VM-based and FaaS solutions. In their research, VM instances process and combine data locally before sending it to another host. In contrast, every serverless function with a payload must interact frequently with its distributed master side. Figure 1 shows the difference in broadcast patterns for both systems. For a VM system with N hosts, the communication complexity is O(N). However, the complexity is $O(N \times K)$ for the FaaS system, where K is the number of functions per host [15].

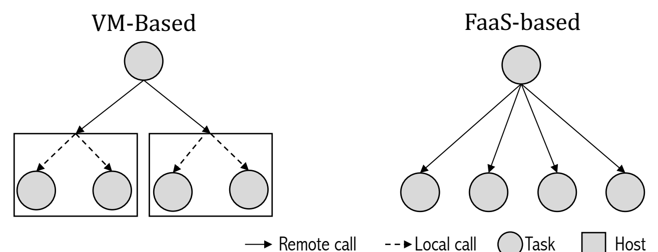

FIGURE 1. The Communication Path of Broadcast Pattern in VM and FaaS Based System

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

Furthermore, the significance of effective communication patterns becomes even more pronounced within intricate, fine-grained serverless systems. A serverless application typically comprises a web of complex interactions among smaller tasks [16]. Within workflows, the interplay between functions necessitates seamless collaboration. However, prevailing master-side patterns tend to impose undue burdens on these fine-grained functions, particularly in terms of internal interactions [3]. This issue is further exacerbated in scenarios involving serverless workflows. In such contexts, the amalgamated functions frequently need to engage with their central controller or workflow engine to facilitate the exchange of internal states. Consequently, the centralized workflow pattern brings substantial connection challenges to the internal functions of serverless workflows.

In contrast, the decentralized worker-side pattern can be utilized to offload the overhead from a central master to the workers [2]. A few studies have emerged that utilized this idea in recent years. Nightcore [17] is an efficient serverless computing framework that orchestrates chained serverless functions of a workflow onto the same host. It relies on a gateway in each worker host to schedule tasks within that host. The gateway acts as a workflow engine proxy, taking over some tasks from the master controller. In this way, the inner calls between chained functions can be processed locally and efficiently because they are located on the same host. Additionally, their internal data can be transmitted and preprocessed by the local gateway without relying on a remote workflow engine.

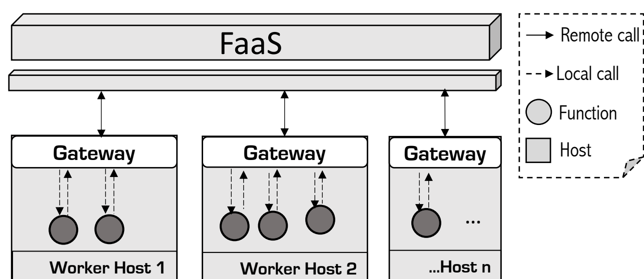

FIGURE 2. Worker-side Pattern Structure in FaaSFlow

While Nightcore seems to utilize worker-side pattern, Li et al. [2] argues that Nightcore is still based on a master-side pattern because its gateway serves as the centralized manager to assign tasks and allocates execution states. They proposed FaaSFlow which is closer to a worker-side design. Figure 2 shows the basic structure of FaaSFlow. There is still a decentralized workflow engine (gateway) located in each worker host similar to Nightcore. The difference is the worker-side engine in FaaSFlow handles more tasks than Nightcore, so the workflow tasks of the master side can be eliminated completely. Thus, FaaSFlow can be considered a worker-side decentralized serverless workflow pattern.

Although both Nightcore and FaaSFlow make use of proxy engine located in each worker host, they are still small regional masters. They rely on gateways to handle internal interactions between serverless functions within the same host, and between gateways to handle remote cross-host interactions. Such worker-side pattern could be further decentralized to have the ability to schedule themselves according to predefined workflow logic without any level of masters. This is the approach proposed in this article which will be described in detail in Section III.

C. OPENWHISK

OpenWhisk, an open-source FaaS system, has a commercial counterpart called IBM Cloud Functions. It enables the scheduling of containerized serverless functions and offers a simple sequential workflow for executing multiple functions. The system utilizes a controller to oversee the scheduling of each serverless function and relies on CouchDB for storing process data. Additionally, OpenWhisk leverages Kafka as a messaging system, essential for decoupling event producers from event consumers. This architectural approach fosters scalable and resilient communication among various components of the system.

The sandbox of OpenWhisk encapsulates serverless functions and is commonly a containerized structure that isolates the functions in serverless applications. Virtualized containers provide an autonomous runtime which includes system libraries and program dependencies. In OpenWhisk, each sandbox only contains the code of a single function due to function isolation concerns. Figure 3 shows the steps from when a sandbox instance is loaded to the time when the function code is injected into the sandbox by the FaaS controller and ready to be invoked. The life of the sandbox ends when it is no longer needed and removed by the controller.

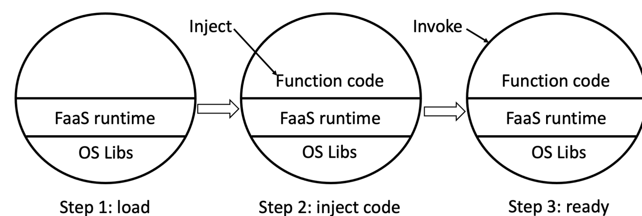

FIGURE 3. The Process Steps of a Sandbox

III. IMPROVING SERVERLESS WORKFLOWS

In Section II, the performance bottleneck of serverless systems that make use of master-side patterns have been outlined. This problem could be alleviated by using a novel worker-side pattern that allows functions to pass control and data from one to another in a workflow without the need for any central controllers. Our implementation of this approach is called BriskChain.

Since BriskChain's implementation requires modifications to the Sandbox's internal structure and the integration of authoritative code into the main controller, an open-sourced serverless platform is required. The underlying architecture

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

of most open-source FaaS platforms is fundamentally similar [18]. We have chosen OpenWhisk, which has been described in Section II-C, as the platform on which to base BriskChain on. An important distinguishing feature of OpenWhisk is its incorporation of a portion of the workflow engine within its structure. This characteristic provides valuable engineering insights that contribute to our implementation strategy. Furthermore, OpenWhisk's Sequence workflow engine is seamlessly integrated into the core FaaS controller. This integration treats workflow tasks and normal serverless functions as interchangeable entities for the controller. This alignment with the substitution principle, as discussed by in [4], harmonizes seamlessly with our concept.

A. ARCHITECTURE OF BRISKCHAIN

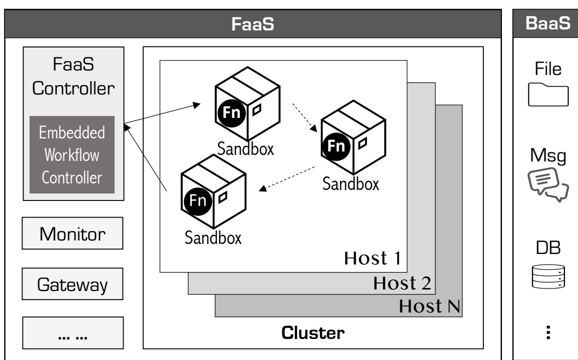

FIGURE 4. Infrastructure of BriskChain

Figure 4 illustrates the infrastructure of BriskChain within the broader FaaS system. The "Embedded Workflow Controller" primarily manages support for serverless workflows. Integrated within the FaaS Controller, this newly incorporated controller enables the entire FaaS system to orchestrate both individual serverless functions and workflows composed of multiple serverless functions. The BriskChain controller can schedule workflow resource allocation and workflow functions. It handles workflow tasks as cohesive units, which, from the perspective of the FaaS controller, are indistinguishable from single functional tasks. This feature ensures there is no operational difference between invoking a workflow and invoking a single function within the BriskChain framework from the user's perspective. Additionally, the new controller supports a locality strategy, orchestrating all the sandboxes in the cloud cluster of the same workflow onto a single host. Consequently, these workflow-related functions can interact with each other through local communication. For a detailed engineer's perspective on the process sequence of the BriskChain workflow system with its new controller, refer to Section III-C.

The Sandbox depicted in Figure 4 is an enhanced version compared to traditional sandboxes used in serverless applications. Its primary feature remains containerizing serverless function code. The "Fn" in the Sandbox in the figure indicates the serverless function encapsulated within the containerized sandbox. The enhanced BriskChain sandboxes can collaborate with the Embedded Workflow Controller to efficiently process serverless workflows. Internally, the enhanced sandbox includes an additional BriskChain runtime layer, represented by the large circles at the bottom of Figure 5. This embedded layer ensures the implementation of serverless workflow features within the sandboxes. These features include reducing reliance on the Controller within workflows, autonomously transmitting workflow context data to subsequent workflow nodes, as indicated by the arrows and dashed arrows between Sandboxes in Figure 4. Additionally, the new Sandbox promptly communicates error information within workflows back to the controller for corrective action or coordination of serverless function scheduling tasks.

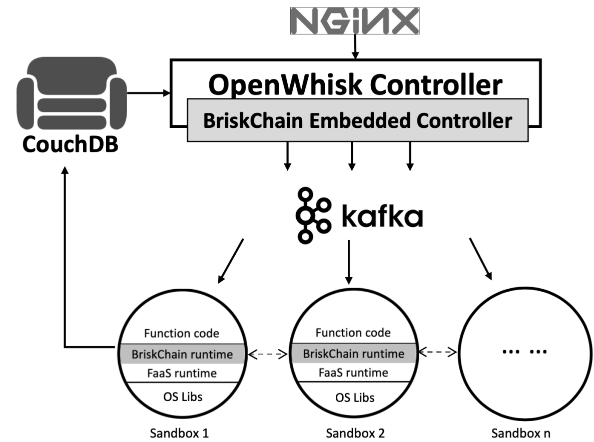

FIGURE 5. OpenWhisk with BriskChain

Other components within the infrastructure of BriskChain, as shown in Figure 4, remain consistent with traditional FaaS systems in terms of functionality and operation. These include, but are not limited to, Monitoring, Gateway, Clusters, and BaaS (Backend as a Service). The Gateway handles both internal and external interactions of the FaaS system. The Monitor focuses on the operation of the sandbox, ensuring that FaaS systems can be observed and detailed logs can be provided. The BaaS offers commonly used external services for FaaS, especially various commonly used storage services.

Figure 5 illustrates the precise software architecture of BriskChain seamlessly integrated within OpenWhisk. The shaded regions in the diagram depict the two fundamental extensions of BriskChain. The BriskChain Embedded Controller functions as an augmentation of the Controller, whereas the BriskChain Runtime enhances Sandbox functionality. Intercommunication among the BriskChain Runtime layers, symbolized by dotted lines with arrows in the figure, across various Sandboxes, facilitates decentralized data exchange to support contextual information within workflow functions. Furthermore, the BriskChain Embedded Controller is equipped with scheduling capabilities tailored for each embedded BriskChain Sandbox.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

B. THE WORKER-SIDE PATTERN IN BRISKCHAIN

The decentralized worker-side pattern means that workers are able to schedule without relying on a master controller. The worker here refers to the sandbox loaded with serverless functions, and the master is a central controller that schedules each worker. The worker-side pattern is mainly supported by the BriskChain runtime which undertakes the function scheduling tasks of the workflows. Specifically, BriskChain sandbox runtime schedule the next function that the current function is connected to, according to the predefined workflow schema. It also forwards the result of the current function to its next function in a workflow directly without having it flow back to the controller or database.

Algorithm 1 BriskChain runtime pseudocode

1: **function** runtime(parameters, schema)
2: $result \leftarrow faas_runtime(parameters)$
3: $node \leftarrow schema.current()$
4: **if** (schema.isNull() or node.isEnd()) **then**
5: $response(result)$ ▷ task end
6: **else if** (sequential node) **then**
7: $next \leftarrow node.child()$
8: $next_sandbox \leftarrow controller.resource(next)$
9: $next_sandbox.msg(result, next)$ ▷ forward to next sandbox
10: **else if** (node is parallel) **then**
11: **for** (i in node.children()) **do**
12: $next \leftarrow node[i]$
13: $next_sandbox \leftarrow controller.resource(next)$
14: $next_sandbox.msg(result, next)$ ▷ forward to a sandbox
15: **end for**
16: **else if** (node is branch) **then**
17: $next \leftarrow node.judge(result)$
18: $next_sandbox \leftarrow controller.resource(next)$
19: $next_sandbox.msg(result, next)$ ▷ forward to next sandbox
20: **else**
21: Throw an exception
22: **end if**
23: **end function**

Algorithm 1 shows the basic processing logic of the BriskChain runtime. There are two main branches in the code logic, one is responsible for normal serverless function processing, and the other is responsible for workflow function processing. If the current function is a function that is not part of a workflow, or it is the last function in the workflow. then the BriskChain runtime will forward this function to the FaaS runtime. Otherwise, it will request new sandbox resources for the next step of the workflow. There are three possibilities in the next step of the workflow. The first one is one single function that follows the current one. In this case, the results of the current sandbox will be transferred asynchronously to the new sandbox. The second case is that it is followed by several parallel sub-functions, then the result will be dispatched to these sub-functions. Finally, if the current workflow node is a conditional branch, then the subsequent route will depend on which condition is satisfied.

The computational complexity of the proposed BriskChain runtime algorithm is determined by the specific structure and flow of the serverless workflow. According to the Algorithm 1, the complexity depends on the number of nodes and their types within the workflow. Sequential nodes contribute to linear complexity, processed one at a time, while parallel nodes increase complexity based on the number of concurrent tasks handled simultaneously. Branch nodes add variability due to conditional evaluations leading to different execution paths. Thus, the overall complexity can be approximated as $O(n+m)$, where n represents the number of sequential nodes and m the number of parallel nodes. This complexity reflects the algorithm's dependency on the total number of nodes, the presence of parallel processing, and the conditions that determine branching paths within the workflow.

C. PROCESSING OF SERVERLESS FUNCTIONS

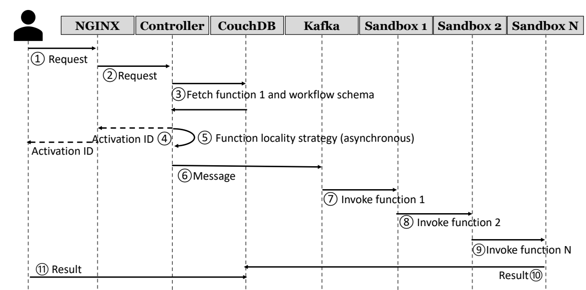

FIGURE 6. **BriskChain Sequence Diagram**

Figure 6 shows the workflow processing sequence diagram using OpenWhisk with BriskChain. The sequence starts from the end-user request to the gateway (NGINX) (①). Specifically, the user invokes the first function of a workflow with a predefined workflow schema. NGINX forwards the requirement to the controller with the function parameters, function ID and the workflow schema ID (②). The controller uses this information to obtain the function source code and detailed workflow definition from CouchDB (③). Then, the user will receive a token (activation ID) after the controller accepts the workflow task (④). Meanwhile, the controller also dispatches the sandbox of the current function to the same location (same host) as other functions in the workflow if possible. The remaining functions of the workflow run one after another without having to communicate with the central controller (⑦, ⑧, and ⑨). This is performed by the embedded controller sending a workflow task to Kafa (⑥). At the end of processing, the result of the workflow comes from the result of the last function of the workflow (⑩). Once this result is stored in CouchDB, the user can use the token (activation ID) to obtain the workflow result.

The BriskChain sandboxes possess the capability to autonomously process interrelated data sequentially, indepen-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

dent of direct interaction with the controller. This autonomy is facilitated by the BriskChain Embedded Controller's predefinition or scheduling of workflows in advance. In scenarios involving sequences, branches, or parallel execution, the BriskChain sandbox automatically routes the output of the current node to the subsequent serverless function within the workflow, as demonstrated in steps 7, 8, and 9 of Figure 6. Moreover, in the special scenario of "fun-in" within a workflow, where multiple nodes contribute output to a single next node, the BriskChain framework relies on an external memory database Redis. This external database temporarily stores output data from multiple preceding nodes before transmitting it to their shared next node.

IV. EVALUATION METHODOLOGY

In order to evaluate the advantage of worker-side pattern that has been implemented in BriskChain, several experiments have been designed to provide answers to the following questions:

(i) What is the scheduling overhead of BriskChain compared with others in different DAG forms?
(ii) What is the performance of BriskChain scheduling serverless functions based on the different payloads of the workflows?
(iii) How does BriskChain perform in real-world applications?

The benchmarks used for evaluation are described in detail in Section IV-A.

	Configuration
Kubernetes Cluster	Google Kubernetes Engine; Zone: us-central1-c; One cluster with three nodes
Cluster Node	301 mCPU requested; 940 mCPU allocatable; 445 MB memory requested; 2.95 gigabyte allocatable
Cluster Software	Container-Optimized OS from Google; Kernel: 5.10.109+; Kubelet: v1.22.11-gke; Operating System: Ubuntu 20.04 LTS
Libraries and Frameworks	OpenWhisk v1.2.0; BriskChain; Composer 0.12.0
Related third-party dependencies	Redis v7.2; Apache Kafka v2.7.0; CouchDB v3.1.1; Nginx v1.19.10
Data	1: Customized simulation data 2: Travis2slack [19]; Vid [20]

TABLE 1. Hardware and Software Setups for the Experiment

The performance of BriskChain will be compared with that of OpenWhisk, and Apache Composer. There are many other open-source serverless platforms such as those reviewed in [18], [21]. However, making fair comparisons between performances of different serverless platforms have been known to be challenging [22]. This is because the optimal performance of such systems often depend on certain configuration parameters which are system dependent. On the other hand, BriskChain, OpenWhisk and Apache Composer share similar runtime environments. Thus, we can more easily configure these systems to ensure a fair comparison.

The hardware and software environment that are used to conduct the experiments are listed in Table 1. A Kubernetes cluster was built on Google Kubernetes Engine (GKE) with three host nodes installed in the same cloud zone. The three severless platforms are installed in the Kubernetes environments.

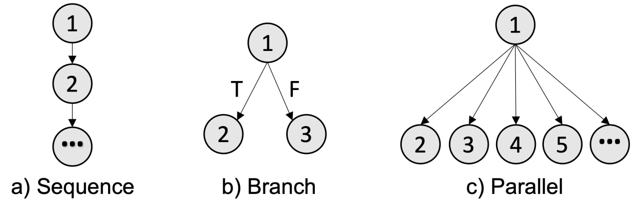

FIGURE 7. Synthetic Benchmarks

Note, however, that OpenWhisk currently only supports sequential function composition. Therefore, for the workflows that involve branching and parallelism, comparisons could only be made between BriskChain and Apache Composer.

A. EXPERIMENTAL BENCHMARKS

Two different kinds of workflows are used as benchmarks in our experiments – synthetic and real applications. The synthetic benchmarks allow us to compare the performances of the systems under sequential, branching, and parallel workflow situations as illustrated in Figure 7. Sequential workflows contain identical functions that are run one after another. The branching workflows contain branching nodes set up in an if-then-else arrangement. Parallel workflows break out from a single function into parallel subtasks. They are also known as fan-out and fan-in workflows because the task breaks out (fan-out) into multiple concurrent sub-functions, and then the results of the sub-functions are collected (fan-in) through a single node.

To evaluate BriskChain under more realistic workflows, two real applications – Travis2slack and video transcoding, are used. They are representative use cases with payloads and external dependencies and involve all three workflows in the synthetic benchmarks.

Travis2slack [19] is a serverless application that responds to notifications from continuous integration and continuous deployment (CI/CD) software projects. It analyses project logs and publishes analysis reports with the original error addresses to the subscribers. The processing steps of Travis2slack are shown in Figure 8. Initially, Travis2slack uses a webhook to receive build notifications from Travis-CI [23] which is a hosted continuous integration service for building and testing software projects hosted on GitHub and Bitbucket. Then it retrieves pull requests and build details. The second task is to fetch and analyze build and test logs. Finally, Travis2slack generates Slack [24] messages for subscribers. This application consists of twelve nodes in its workflow as shown in Figure 9. The descriptions of these twelve nodes are provided in Table 2.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

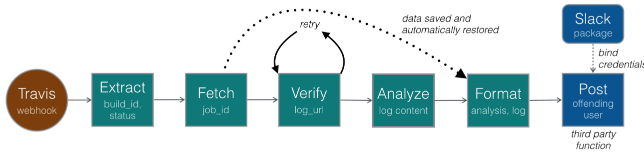

FIGURE 8. Pipeline of Travis CI to Slack

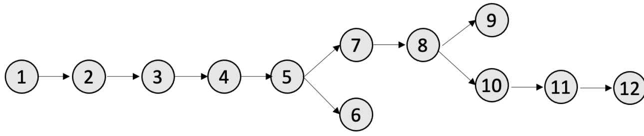

FIGURE 9. Workflow of Travis CI to Slack

	Node	Explanation
1	Webhook	The message comes from Travis.
2	Echo	Data cleaning.
3	Extract	Build notification information.
4	Fetch	Queries Travis CI to determine the job ID.
5	Check	Is a pull request number defined?
6	Return	Record logs, output error and terminating computation.
7	Slack author	Get Slack author.
8	Is subscribed	If author is not subscribed for notifications.
9	Return	Record logs, output error and terminating computation.
10	Analyse	Fetches and analyses the CI logs.
11	Format	This action formats notification message with log analysis.
12	Post	Sends the message to Slack

TABLE 2. Workflow Nodes for the Travis2slack Benchmark

Video Transcoding (Vid) is a serverless application that converts videos from one encoding format to another. It is a resource-intensive task, and traditional transcoding application software consumes a lot of computing resources. Its serverless workflow involves three major steps – splitting, transcoding, and merging. The splitting step segments the video into a series of videos of shorter durations. This step is required to overcome the time limit of serverless functions. For example, OpenWhisk has a default time limit of one minute. These segmented video files are then transcoded into the target format. In the workflow, many segmented videos are transcoded in parallel to increase efficiency. Finally, the transcoded segmented videos are merged into a single video file. This workflow is illustrated in Figure 10 and a description of the nodes can be found in Table 3. In our experiment, a 100-second video file in MOV format is transcoded to MP4 format.

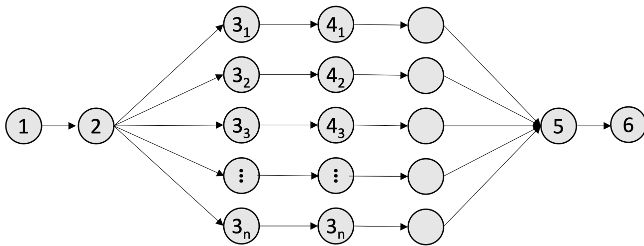

FIGURE 10. Workflow of Video Transcoding

	Node	Explanation
1	Trigger	It triggers the workflow by a newly uploaded video.
2	Split	It splits the video into small video slices according to a defined time interval.
3	Transcode	It transcodes sliced videos in parallel.
4	Other tasks	Other optional tasks such as content security review or video watermarking.
5	Merge	Merge(fan-in) the transcoded videos into one final video.
6	After-process	Update information to database.

TABLE 3. Workflow Nodes for the Video Transcoding Benchmarks

B. EVALUATION METRICS

The effectiveness of BriskChain will be evaluated on the amount of time needed time spend on setting up and getting the serverless functions ready for a workflow. Obviously, this *runtime overhead* depends on the number of functions involved in the workflow and the size of the payload. In our experiments, unless stated otherwise, the overhead includes the total time spent outside of the functions. It typically includes scheduling time, state transition time, and delay time between the workflow functions. Apart from the total overhead time, we will also measure the 95th percentile latency.

V. RESULTS

A. SYNTHETIC BENCHMARKS

1) Sequence Workflow

The sequence workflows consist of identical functions. Furthermore, these functions do not perform any processing and respond to calls immediately. Also, there is no payload transferred between the internal functions of the workflow; only a short string is transmitted to record the execution status of the workflow. The functions will normally encounter a long cold start delay when they are run for the first time. However, the results shown in this experiment do not include any cold start delay. Each workflow case is executed 100 times.

The results in Figure 11 show that BriskChain outperforms OpenWhisk by over 70%. This is direct result of the worker-side pattern in BriskChain. On the other hand, OpenWhisk requires multiple internal interaction loops between the master and the workers when processing the workflow, resulting in much higher overhead. Another desirable effect of BriskChain's worker-side pattern is that the overhead increases linearly with the number of functions in the sequence.

Another reason why BriskChain's overhead is lower is that it omits many unnecessary remote database accesses. OpenWhisk stores the result of each serverless function call into Apache CouchDB, which may be located in a remote host from the functional tasks of the workflow. On the other hand, BriskChain sends the result of the current function directly to the next function of the workflow, without accessing the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

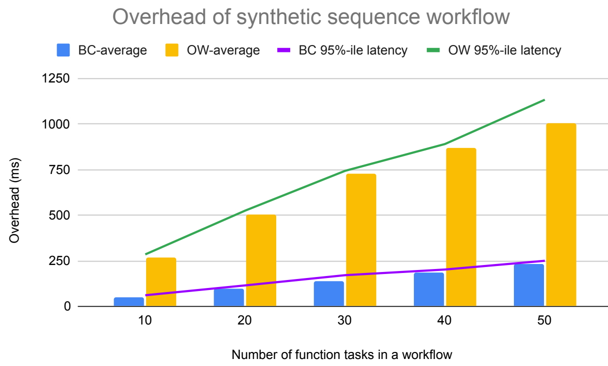

FIGURE 11. Overhead of Synthetic Sequential Workflows

database unless any error occurs.

2) Branch Workflows

The branch workflows each consist of two branches. If the result of the judicial function is true, the result is dispatched to the first branch. Otherwise, it goes to the second branch. Since OpenWhisk does not support branch scheduling, comparisons are made between Apache Composer and BriskChain. Note that this experiment does not examine the workflows with more than two branches.

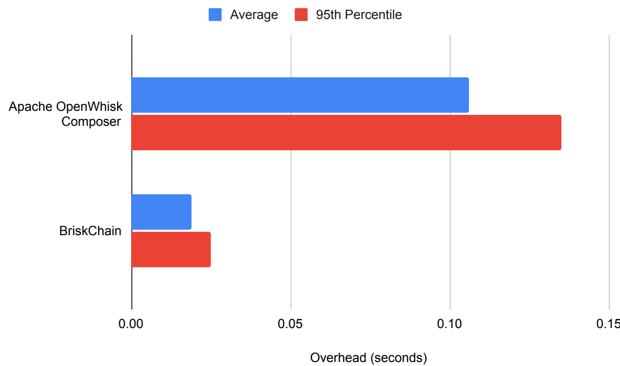

FIGURE 12. Overhead of Synthetic Branch Workflow

The results in Figure 12. shows that BriskChain spent significantly less time in overheads compared with Apache Composer. The 0.023 seconds of overhead on average required by BriskChain is about one-third of that for Apache Composer. Furthermore, the 95th percentile latency is 70% lower. This is because BriskChain dispatches the branch node directly from the condition node without waiting for the scheduling order from the controller. A branch is no different from a sequence, except that there is an additional conditional test before scheduling the next correct workflow node. Therefore, the overhead of BriskChain in the branching benchmark is also low, similar to the sequence workflow experiments.

3) Parallel Workflows

The experiment simulates synthetic parallel workflows composed of various number of sub-functions. A whisker plot of the results for Apache Composer and BriskChain over 1000 runs is shown in Figure 13. Overall, BriskChain requires 50-75% less overhead than Composer across the different number of parallel functions. Also, the overhead of BriskChain grows linearly with the number of parallel functions, but Composer shows exponential growth. For example, in a parallel test of 30 concurrent functions, the overhead of BriskChain is only 1/5 of the overhead of Composer. In addition, BriskChain overhead shows much smaller variance. For example, for 30 parallel tasks, some executions with Composer use nearly 3000 milliseconds which is 50% higher than the lowest value. This is because parallel scheduling in Composer requires complex multisystem dependencies. Delays in any one of these components will lead to delays in the entire workflow. As the number of parallel functions increases, the probability of latency becomes greater, and hence the higher variance.

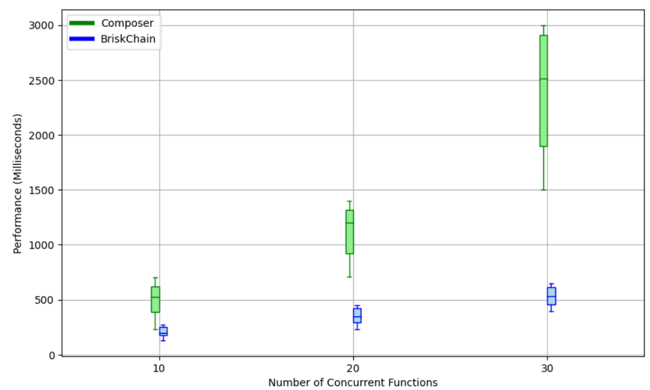

FIGURE 13. Whisker Plot of Parallel Workflow Execution Time

Another reason why BriskChain shows better performance than Composer is resource competition. Large numbers of subtasks in parallel workflow lead to large resource demand spikes, albeit for a short period of time. Such spikes could cause network resource contention. BriskChain's locality strategy enables serverless functions of the same workflow to be exeduted on the same host. Therefore, such network contention issues could be avoided.

4) Payload Evaluation

To assesses how well BriskChain performs with various payload sizes, we constructed a workflow using six functions that operate sequentially. Different payload sizes are used, with each function handing off the payload to the next until the workflow is completed. Figure 14 shows the overhead associated with each payload size. In comparison with BriskChain, OpenWhisk requires roughly three times higher overhead for a 1K byte payload. The overhead of OpenWhisk also increases exponentially while that for BriskChain shows only polynomial increase. This is due to riskChain's locality

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

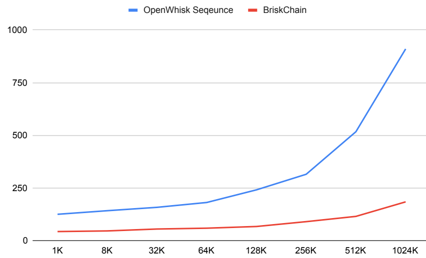

FIGURE 14. Overhead with Different Payloads

mechanism effectively avoiding cross-host internal communications. High overhead for high payload in typical FaaS systems is the reason why many platforms place a limit on the payload size.

B. APPLICATIONS CASE STUDIES

Figure 15 depicts the transcoding performance of the video transcoder workflow. The graph on the left-hand-side of this figure shows the percentage improvements as a 90-second video is divided into videos segments of shorter durations. The percentage is calculated with respect to that of Composer for the 90s video. The graph on the right-hand-side of Figure 15 shows the change in the percentage of time taken to complete the workflow as the video segment duration changes.

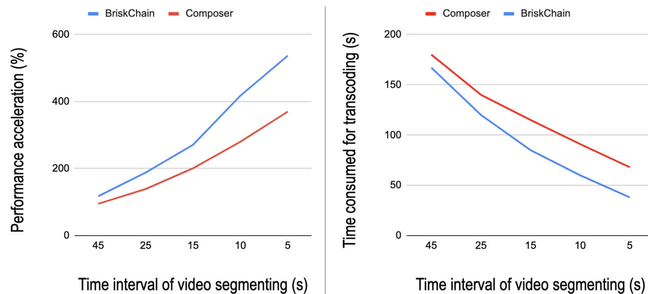

FIGURE 15. The Effect of BriskChain for Video Transcoding

As the original video is split up into a larger number of segments (with shorter segment durations), transcoding is performed by a larger number of distributed functions. Hence, in general, the efficiency increases as the duration becomes shorter. The results show that even with the 45-second segments, the workflow required 180s and 167s to complete with Apache Composer and BriskChain respectively. This correponds to BriskChain having an 18% better performance. With 5-second segments, BriskChain outperforms Apache Composer by a margin of about 35%, with the workflow completed in 38s

The performances of Apache Composer and BriskChain for Travis2slack are shown in Figure 16 as the payload increases. In general, there is a deterioration in performance with increased payload. As the payload quantity is increased

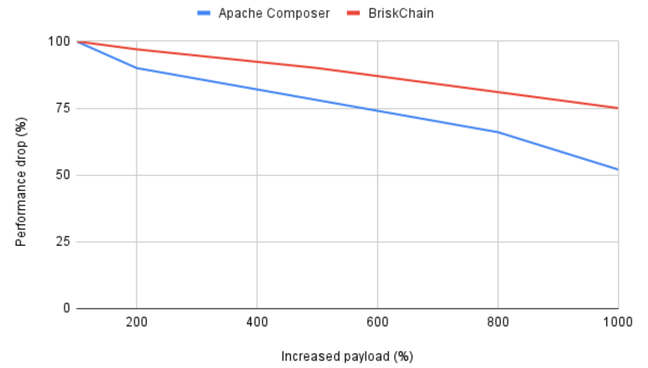

FIGURE 16. The Effect of BriskChain at Travis2slack

ten-fold, Apache Composer experienced a 50% reduction, while BriskChain's performance merely encountered a 25% decline.

It should be noted that data transactions surpassing 1MB between serverless functions on different hosts will introduce bottlenecks. BriskChain always attempts to orchestrate serverless functions from the same workflow onto the same host, effectively circumventing this kind of bottlenecks. Consequently, as the quantum of payload data within the workflow escalates, BriskChain is able to achieve not only a more gradual pace of performance deterioration but also sidesteps the potential hindrance posed by hefty data loads.

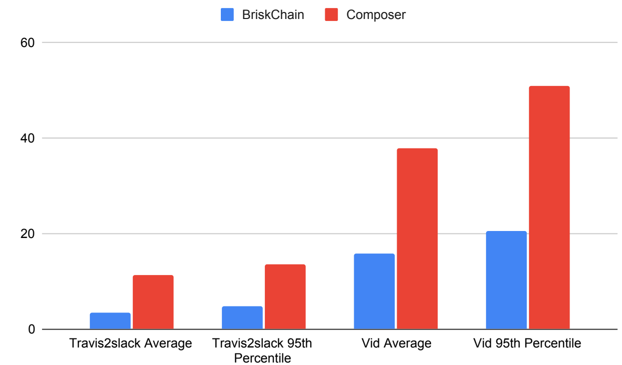

FIGURE 17. Overhead time (seconds) for Travis2slack and Vid

Figure 17 compares the overhead time between BriskChain and Composer for both applications. For Travis2slack with original basic payload, BriskChain's overhead is 3.5 seconds, about 30% of Composer's overhead of 11.3 seconds. In the case of Vid, BriskChain requires 60% less overhead on average than Composer for video segments of five seconds. The superiority of BriskChain for these applications mirrors the results obtained with the synthetic workflows.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

VI. CONCLUSIONS

This article presented the design of a novel worker-side pattern for serverless computing. Its implementation, called BriskChain, is based on the OpenWhisk open-source serverless platform. It shows that by reducing the communication required for each function sandbox with the central controller, computation overhead could be drastically reduced. Furthermore, with an embedded controller in each sandbox, each function could directly transfer payloads to a subsequent function in the workflow, thereby reducing the overhead for data transfer even more. Experimental results using BriskChain showed that In this way, serverless applications are able to run much more efficiently in terms of the runtime overhead required, for both synthetic and real application workflows.

The concept of decentralized architecture holds promise for improving the efficiency of cloud computing, and is worthy of further research. An example of the use of decentralization is to perform auto-scaling. It is an important feature of every virtualized container and is typically supported by the master in an FaaS system. Decentralized containers could be designed to manage their own auto-scaling instead of relying on the master. This implies that each container needs to have its own lifespan, with any inactive container automatically terminated after a predetermined length of time. In this way, the requirement for scaling down could be fulfilled. When service demand increases, each container could spawn more replicas by itself to enable scaling up.

REFERENCES

[1] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[2] 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 *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Feb. 2022, pp. 782–796.

[3] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys*, vol. 54, no. 10, pp. Article no. 220, 34 pages, 2023.

[4] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," in *Proceedings of ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Oct. 2017, pp. 89–103.

[5] 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*. IEEE, 2022, pp. 1868–1877.

[6] Z. Wen, Q. Chen, Y. Niu, Z. Song, Q. Deng, and F. Liu, "Joint optimization of parallelism and resource configuration for serverless function steps," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[7] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 1069–1078.

[8] J. Chen, F. Xu, Y. Gu, L. Chen, F. Liu, and Z. Zhou, "Harmonybatch: Batching multi-slo dnn inference with heterogeneous serverless functions," *arXiv preprint arXiv:2405.05633*, 2024.

[9] M. Adhikari, T. Amgoth, and S. N. Srirama, "A survey on scheduling strategies for workflows in cloud environment and emerging trends," *ACM Computing Surveys*, vol. 52, no. 4, pp. 1–36, 2019.

[10] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*. Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 263–274.

[11] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow:{Low-Latency} video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Symposium, 2017, pp. 363–376.

[12] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions," *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020.

[13] B. Balis, "Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows," *Future Generation Computer Systems*, vol. 55, pp. 147–162, 2016.

[14] A. Mahgoub, L. Wang, K. Shankar, Y. Zhang, H. Tian, S. Mitra, Y. Peng, H. Wang, A. Klimovic, H. Yang *et al.*, "Sonic: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 285–301.

[15] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[16] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proceedings of the 21st International Middleware Conference*. Proceedings of the 21st International Middleware Conference, 2020, pp. 356–370.

[17] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[18] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in *Proceedings of the 5th International Workshop on Serverless Computing*, Dec. 2019, pp. 37–42.

[19] "Travis ci to slack," https://github.com/rabbah/travis-to-slack, 2022. [Online]. Available: https://github.com/rabbah/travis-to-slack

[20] Alibaba Cloud, "Build an elastic and highly available audio and video processing system in a serverless architecture," https://www.alibabacloud.com/help/en/function-compute/latest/build-an-elastic-and-highly-available-audio-and-video-processing-system-in-a-serverless-architecture, 2022. [Online]. Available: https://www.alibabacloud.com/help/en/function-compute/latest/build-an-elastic-and-highly-available-audio-and-video-processing-system-in-a-serverless-architecture

[21] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Analyzing open-source serverless platforms: Characteristics and performance," in *Proceedings of 33rd International Conference on Software Engineering and Knowledge Engineering*, Jul. 2021, pp. 15–20.

[22] J. Decker, P. Kasprzak, and J. M. Kunkel, "Performance evaluation of open-source serverless platforms for kubernetes," *Algorithms*, vol. 15, no. 7, p. 234, 2022.

[23] Idera, Inc., "Travis ci," http://travis-ci.com, 2022. [Online]. Available: http://travis-ci.com

[24] Slack Technologies, LLC, "Slack," https://slack.com, 2022. [Online]. Available: https://slack.com

KAN WANG received his Master degree in Computer and Information Sciences from Auckland University of Technology, Auckland, New Zealand in 2023.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3439695

JING MA (Member, IEEE) received the Ph.D. degree in Computer Sciences from Auckland University of Technology, New Zealand, in 2019. She is currently a lecturer at the department of Computer Sciences and Software Engineering at Auckland University of Technology, Auckland, New Zealand.

EDMUND M-K LAI (Senior Member, IEEE) received his BE(Hons) and PhD from The University of Western Australia, both in Electrical Engineering, in 1982 and 1991 respectively. He is currently Professor of Information Engineering at the Auckland University of Technology, New Zealand. He has over 30 years of academic experience, having previously held faculty positions at universities in Australia, Hong Kong, and Singapore. He has published over 150 international refereed journal and conference papers in signal processing, intelligent control, computational intelligence, and artificial neural networks. Dr Lai is also a Fellow of the Institution of Engineering and Technology (FIET) and a Fellow of Engineers Australia (FIEAust).

• • •

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4