Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

BriskChain: Decentralized Function Composition for High-performance Serverless Computing

Kan Wang[1] | Jing Ma[1,2] | Edmund M-K Lai[1]

[1]School of Engineering, Computer and Mathematical Sciences, Auckland University of Technology, New Zealand
[2]Otago Polytechnic Auckland International Campus, New Zealand

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 patterns which hinder the interaction between serverless functions, causing performance issues. In this paper, a decentralized worker-side pattern is proposed that provides enhanced performance by allowing each workflow function to schedule itself from the current node to the next without having to interact with the master controller. It treats the serverless workflow as a whole unit and uses a locality strategy to optimize performance. A new high-performance workflow engine called BriskChain has been implemented to demonstrate the effectiveness of this approach. Experiments on a real-world application show that BriskChain is able to achieve 2.5 times better performance than a state-of-the-art serverless workflow engine.

KEYWORDS:
Serverless Computing, Serverless Workflow, FaaS

1 | 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 is often referred to as Function as a Service (FaaS), and it 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 multiple function services into workflows through a workflow engine to create their application logic. A workflow engine manages, schedules, or monitors action tasks according to predefined rules. At present, most serverless workflow engines rely on centralized master-side pattern, which involves a powerful workflow engine as the master, scheduling and allocating tasks and resources to many workers or serverless function tasks. This approach has significant performance issues due to high scheduling and data movement overhead. In addition, cloud vendors typically impose quotas on the input and output data size of each function. Users often have to make use of additional database storage services for temporary data storage and delivery [2]. Therefore, this pattern is not very suitable for applications that require intensive interactions between smaller tasks [3].

To address these issues, this article proposes a decentralized work-side pattern that can significantly reduce communication overhead in serverless workflows. In this pattern, each worker node has the ability to route state from itself to the next worker node according to a predefined workflow schema. Each worker task transmits its state directly to the next worker in the workflow,

eliminating the need to route it through the master controller. This will significantly reduce workflow overhead. At the same time, the pressure on the master side is relieved.

A decentralized workflow engine called BriskChain has been implemented based on this pattern for latency-sensitive and interactive serverless computing. BriskChain schedules serverless workflows as a whole unit, just like a single serverless function, which abides by the substitution principle[4]. This approach makes no distinction between serverless function tasks and workflow tasks, allowing BriskChain to schedule the entire workflow like a normal serverless function. BriskChain is built on an open-source FaaS system called OpenWhisk. BrishChain 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 controller is responsible for allocating workflow resources. It also implements a close-locality optimization strategy that schedules sandboxes of the workflow onto the same host to avoid remote interactions between workflow functions, thereby enhancing efficiency.

The rest of this article is organized as follows. Section 2 provides a brief review of current practices in serverless computing, focussing on function composition and workflow engines. It also gives an overview of worker-side patterns. The proposed decentralized serverless workflow is presented in Section 3, together with the detailed design of BriskChain. A comprehensive evaluation of BriskChain, complete with a real-world application example, is provided in Section 4. The last section concludes this article with some suggestions for future work.

2 | RELATED REVIEW

2.1 | Serverless Functions and Workflow Engines

Serverless functions are basic building blocks of serverless applications. Sandboxes, such as Docker containers, provide a lightweight and executable environment for serverless functions. Each serverless function is packaged into a containerized sandbox to become an autonomous and independent entity in a FaaS application. Multiple containerized functions can form more complex serverless applications, where each serverless function handles a small, fine-grained logical service.

To create a serverless application, developers write logic codes that follow the specifications of serverless functions during the application design process. The FaaS system then loads each function into the sandbox, ready for execution. Finally, these containerized functions are scheduled and executed by the FaaS runtime. By integrating multiple serverless functions on a FaaS platform, complex applications can be built.

In addition to serverless functions, there are related components such as gateways, events, and Backend-as-a-Service (BaaS), which together constitute the FaaS System. Figure 1 illustrates the typical structure of a serverless system. Function services can be exposed externally through an API Gateway or Event. Serverless functions can also rely on BaaS for additional cloud support. For example, AWS S3 and Google Cloud Storage are BaaS storage services that can be used for persistent storage services.

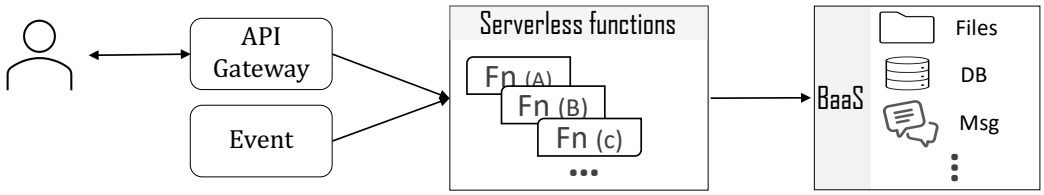

Figure 1 A Typical Structure of FaaS and Related Components

Workflow systems use a centralized scheduler for task assignments and resource allocation[5]. Figure 2 illustrates a typical workflow engine scheduling serverless functions. Most serverless workflow engines work as a separate external system to FaaS. FaaS is responsible for executing serverless functions that can actually be thought of as tasks in a workflow, whereas the workflow engine is in charge of scheduling these serverless functions according to predefined workflow logic.

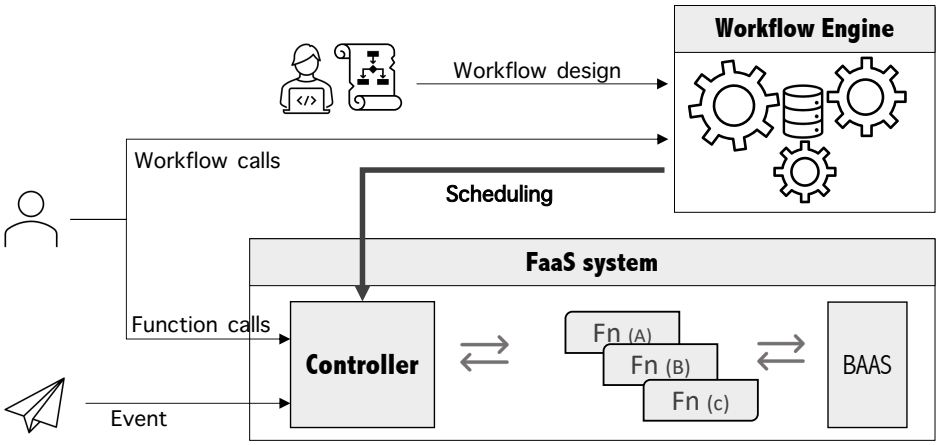

Figure 2 Traditional Workflow Engines with FaaS

The substitution principle of FaaS systems requires that the serverless workflow itself be a serverless function[4]. However, not all FaaS systems obey this principle. For example, Amazon Step Functions and Lambda functions are not the same type of entity and therefore do not obey the substitution principle.

Jonas et al.[6] compared three common communication patterns –broadcast, aggregation and shuffle, for 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 on distributed remote hosts. Figure 3 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[6].

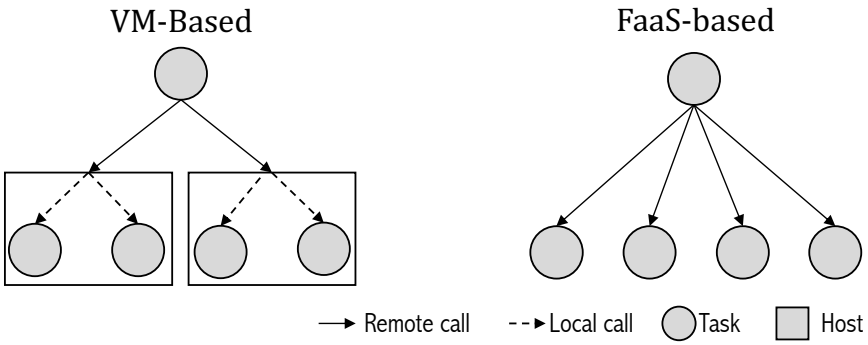

Figure 3 The Communication Path of Broadcast Pattern in VM and FaaS Based System

Most workflow systems adopt the traditional master-side pattern[7,8,9,10]. It relies on a powerful workflow engine to manage tasks and resources. Due to high scheduling overhead and frequent data movement, this pattern does not perform well in serverless environments[2]. This is because each task in a workflow is a serverless function that needs to interact with the remote master frequently. Moreover, if one serverless function needs to transport data to another function, it must move the data to the master side first. This kind of interactions between serverless functions contribute to inefficiency[6].

2.2 | Worker-side Pattern

A decentralized worker-side pattern can be utilized to improve performance. The main idea is 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[11] 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 like a workflow engine proxy that takes 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. Also, their internal data can be transmitted and preprocessed by the local gateway, without relying on a remote workflow engine.

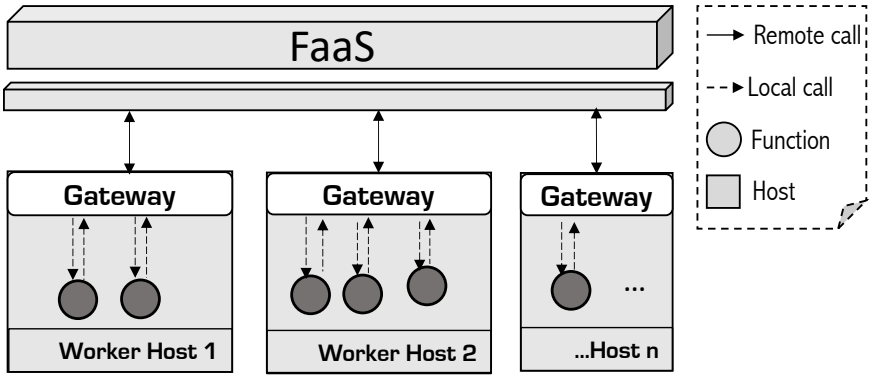

Figure 4 Worker-side Pattern Structure in FaaSFlow

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 4 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.

2.3 | Sandbox Structure

The sandbox encapsulates serverless functions and is commoly a containerized structure that isolates the functions in serverless applications. Virtualized containers provide an autonomous runtime which includes system libraries and program dependencies. In existing FaaS systems, each sandbox typically only contains the code of a single function due to function isolation concerns. Figure 5 shows the steps from when a sandbox instance is loaded from a container image to the time when the function code is injected into the sandbox by the FaaS controller and it is ready to be invoked. The life of the sandbox is ended when it is removed by the controller when it is no longer needed.

2.4 | OpenWhisk

OpenWhisk is an open-source FaaS system although it has a commercial version called IBM Cloud Functions. It is a perfect vehicle to help understand some of the limitations of current FaaS systems.

Figure 6 depicts the processing flow and internal structure of OpenWhisk. NGINX serves as the gateway for OpenWhisk, handling external user requests and exposing function services. CouchDB provides persistent storage of function source code and running states. The controller acts as the central manager, responsible for scheduling and managing serverless functions. Upon receiving a function task from NGINX, the controller fetches the function code from CouchDB, injects it into a sandbox, invokes the sandbox, and retrieves the function result. Kafka functions as a message bus that sends messages or commands to the sandboxes.

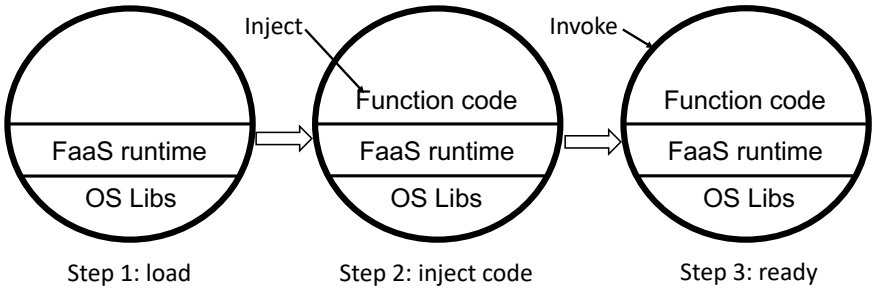

Figure 5 The Process Steps of a Sandbox

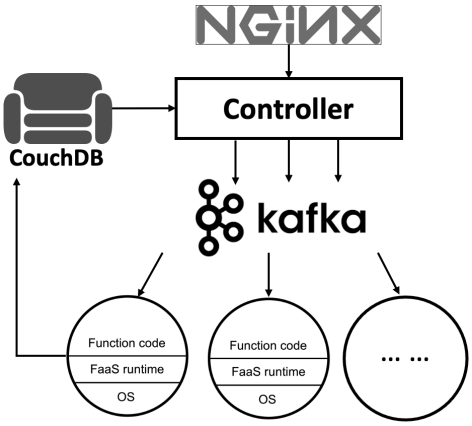

Figure 6 The Internal Structure of OpenWhisk

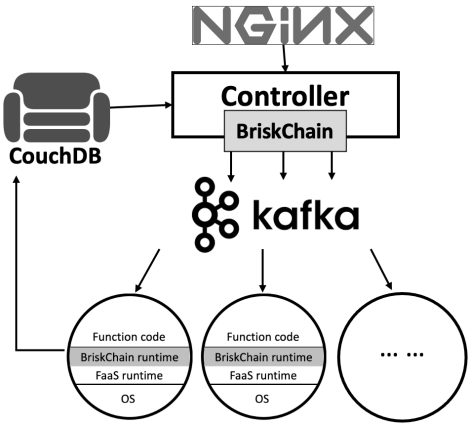

Figure 7 OpenWhisk with BriskChain

Figure 8 illustrates the sequence of actions undertaken from the time a function is requested to the return of the results. The whole process starts from the user initiating a serverless function request to NGINX, which includes function parameters and identities. NGINX then forwards the request to the controller. The controller obtains the program source code of the function from CouchDB and injects it into a loaded sandbox. The controller sends an invocation message to Kafka. Meanwhile, the controller will asynchronously send an activation token to the user because the function may run for a while and the user can request the result later with the token. When Kafka received the innovation message, it transfers the request to the sandbox to invoke the function. The function code will be run in the sandbox and the result will be stored in CouchDB. Finally, the user can fetch the function result from CouchDB.

3 | IMPROVED SERVERLESS WORKFLOW – BRISKCHAIN

The performance bottleneck of current serverless systems have been outlined in Section 2. With the aim to significantly improve performance, we propose 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. It has been implemented as an extension to OpenWhisk which will be known as BriskChain.

Figure 9 shows the basic structure of BriskChain in relation to the rest of the FaaS system. It has two important components – an embedded workflow controller and the BriskChain runtime (sandbox). The BriskChain runtime is a sandbox that serves as the environment for serverless functions. This runtime is endowed with the ability to schedule workflow functions from the current node to the next through the embedded workflow controller. Each sandbox can schedule the next task of the workflow without the support of a central controller. This is key to the implementation of a complete worker-side pattern that satisfies the substitution principle as described in Section 2.1. In BriskChain, workflow tasks and serverless function tasks are treated

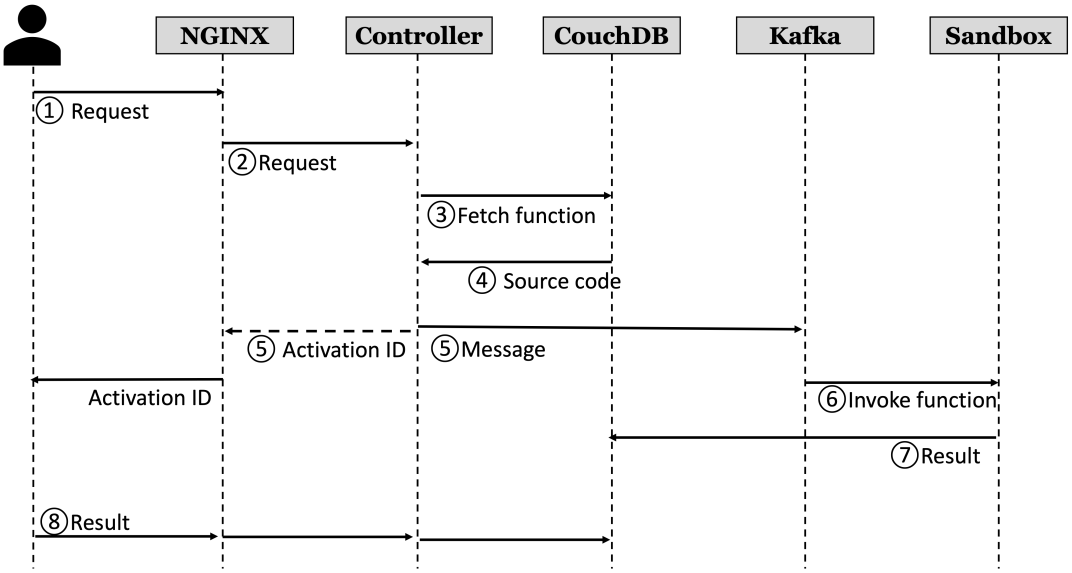

Figure 8 OpenWhisk Sequence Diagram

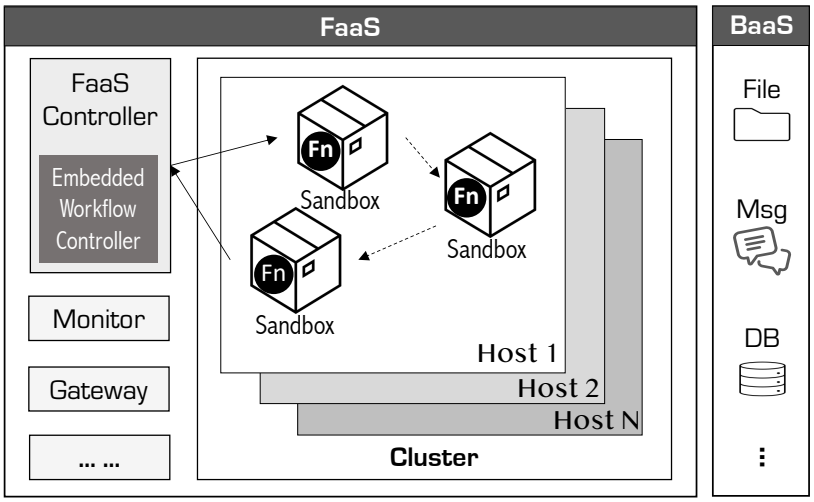

Figure 9 Infrastructure of BriskChain

equally. As a result, serverless workflows have all the attributes and capabilities of normal serverless functions, such as default parameters, limits, blocking invocation, and web export. The benefit is that the system's overall performance can be improved.

The BriskChain sandbox is an extended version of the common FaaS sandbox, as shown in Figure 7. The BriskChain runtime is able to handle both single standalone serverless functions and serverless functions in a workflow. For example, if the function in the sandbox is a normal serverless function (i.e. not a function of the workflow), the BriskChain runtime will forward the task to the FaaS runtime for processing. This ensures that the processing logic of the original serverless function remains unchanged. In contrast, if the function is a serverless function of a workflow, the BriskChain runtime will process the function according to the specified workflow. This will be explained further in Section 3.1.

There is another part of BriskChain that is embedded in the FaaS controller. Figure 7 shows how BriskChain is integrated with OpenWhisk. The database stores the source code and request response status of serverless functions. Each Kubernetes node is a physical host managed by a Kubernetes cluster, which can be thought of as a virtualized computer with unlimited resources. Each serverless function is wrapped in a sandbox, with each sandbox runnubg in a cluster.

3.1 | 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

```
function runtime(parameters, schema){
   result <- faas_runtime(parameters)
   node <- schema.current()
   if(schema.isNull() or node.isEnd()){
      response(result) /* task end */
   }else if(node is sequence){
      next <- node.child()
      next_sandbox <- controller.resource(next)
   /* asynchronously forward to the next sandbox */
      next_sandbox.msg(result, next)
   }else if(node is parallel){
      for(i in node.children()){
         next <- node[i]
         next_sandbox <- controller.resource(next);
         /* asynchronously forward to a sandbox */
         next_sandbox.msg(result,next)
      }
   }else if(node is branch){
      next <- node.judge(result)
      next_sandbox <- controller.resource(next)
   /* asynchronously forward to the next sandbox */
      next_sandbox.msg(result, next)
   }else
      throw an exception
}
```

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.

3.2 | Processing of Serverless Functions

The BriskChain controller has the ability to support function locality strategy. The controller orchestrates all the sandboxes of the same workflow into the same host. Therefore, these functions can interact with each other through local communication.

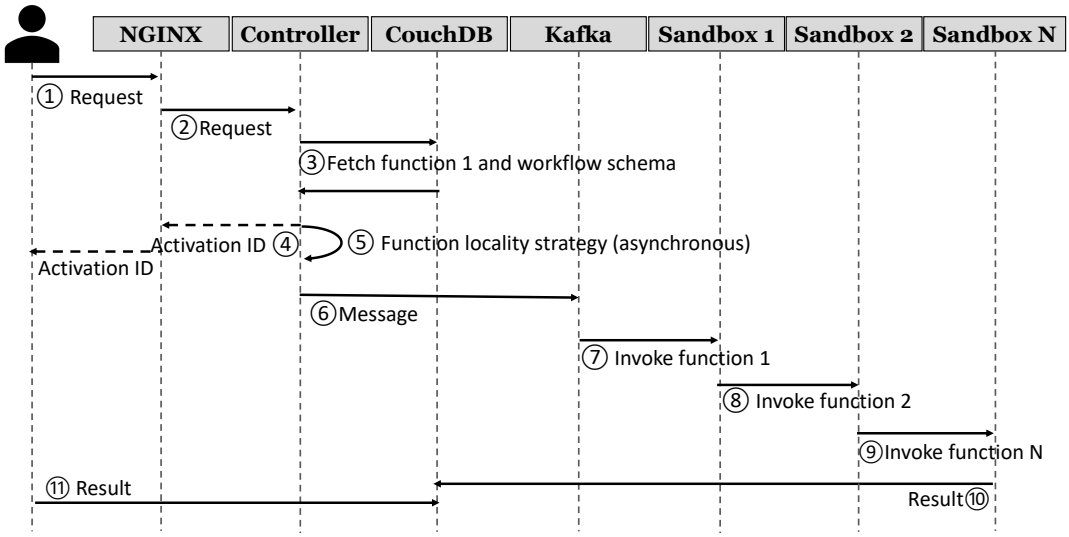

Figure 10 BriskChain Sequence Diagram

Figure 10 shows the workflow processing sequence diagram using OpenWhisk with BriskChain. This could be compared with Figure 8 which is for OpenWhisk without 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.

It can be seen that BriskChain handles workflow tasks as a whole unit, and this workflow unit is no different from a single functional task as far as the FaaS controller is concerned. Also, there is no operational difference between invoking a workflow and a single function.

4 | EVALUATION METHODOLOGY

Several experiments will be conducted to assess the performance of BriskChain in comparison with OpenWhisk[12] and Apache Composer[13]. These systems have similar runtime environments. However, OpenWhisk currently only supports sequential function composition. Therefore, comparison with Apache Composer is needed when the application involves branching and parallelism.

The evaluation seeks to answer 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?

	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 GB allocatable
Cluster Software	Container-Optimized OS from Google; Kernel: 5.10.109+; Kubelet: v1.22.11-gke; OpenWhisk v1.2.0; BriskChain; Composer 0.12.0

Table 1 Hardware and Software Setups in the Experiment

Table 1 shows the hardware and software environment used to conduct the experiments. A Kubernetes cluster was built on Google Kubernetes Engine (GKE) with three host nodes installed in the same cloud zone. OpenWhisk, Apache Composer and BriskChain are installed in the Kubernetes environments.

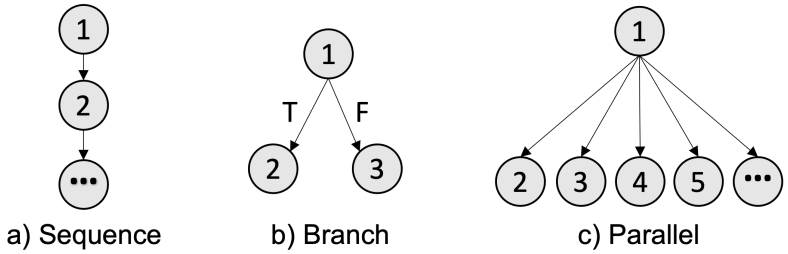

Figure 11 Synthetic Benchmarks

4.1 | Benchmarks

Two different kinds of benchmarks are used in the experiments. They are synthetic micro workflows and real-world applications. The synthetic benchmarks can be divided into three different types: sequence, branch, and parallel. Their workflows are shown in Figure 11. The sequence workflows contain identical functions that are run one after another. The branch workflows contain branching nodes. The 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.

Two real-world applications – Travis2slack and video transcoding, are also used for evaluation. They are representative use cases with payloads and external dependencies and involve all three workflows.

Travis2slack[14] is a serverless application that responds to notifications from continuous integration and continuous deployment (CI/CD) software projects. It automatically analyses the project log and publishes the analysis report and the original error address to subscribers.

The processing steps of Travis2slack are shown in Figure 12. Initially, Travis2slack uses a webhook to receive build notifications from Travis-CI[15] 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[16] messages for subscribers. This application has twelve nodes in its workflow as shown in Figure 13. Description of these twelve nodes are provided in Table 2.

Video Transcoding (Vid) is a serverless application that converts videos from one encoding format to another. Video transcoding 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 divides the video file into

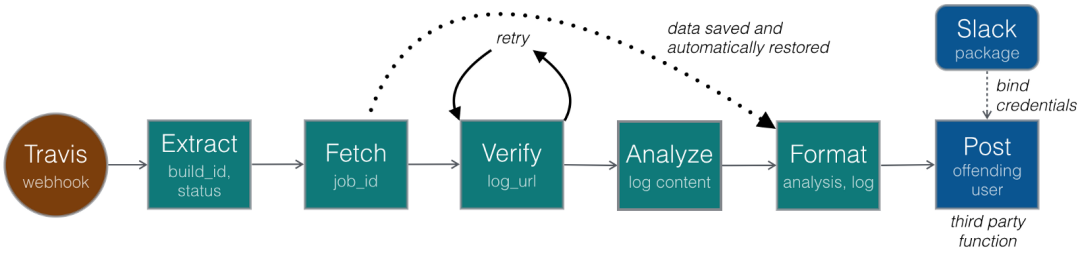

Figure 12 Pipeline of Travis CI to Slack

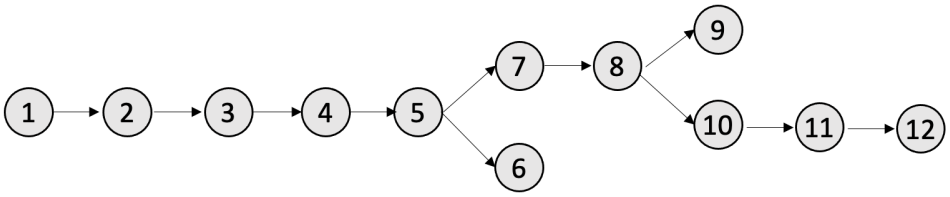

Figure 13 Workflow of Travis CI to Slack

	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

a series of smaller video files of a shorter duration. This step is required to overcome the time limit of serverless functions. For example, OpenWhisk has a default limit of 1 minute. Each segmented video file is 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 14 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.

4.2 | Evaluation Metric

Overhead is a key metric to measure the quality of BriskChain workflow scheduling. The runtime overhead of the workflows depends on the following parameters:

(a) **Number of functions:** This is a significant parameter with the performance difference for different numbers of serverless functions in the workflows.

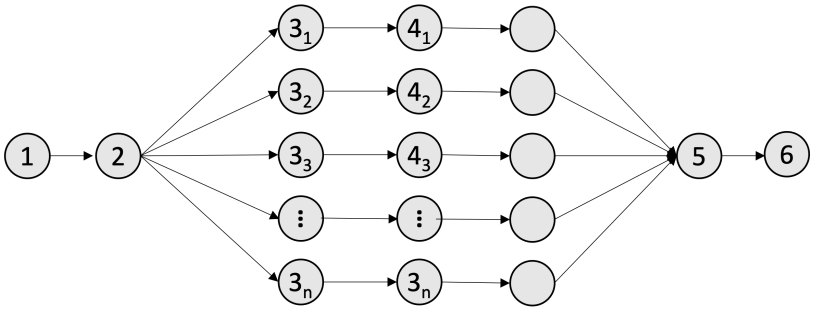

Figure 14 Workflow of Video Transcoding

	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) **Variability and 95th percentile latency:** The 95-th percentile latency and variability of the overhead values are also important in the evaluation.
(c) **Size of payload:** This is an important parameter with the performance difference when BriskChain schedules serverless workflows with payloads of different sizes.

In our benchmarks, 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.

5 | RESULTS

5.1 | Synthetic Benchmarks

5.1.1 | 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.

Figure 15 shows the results for BriskChain and OpenWhisk. They show that BriskChain outperforms OpenWhisk by over 70%. This is directly due to the worker-side pattern of 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

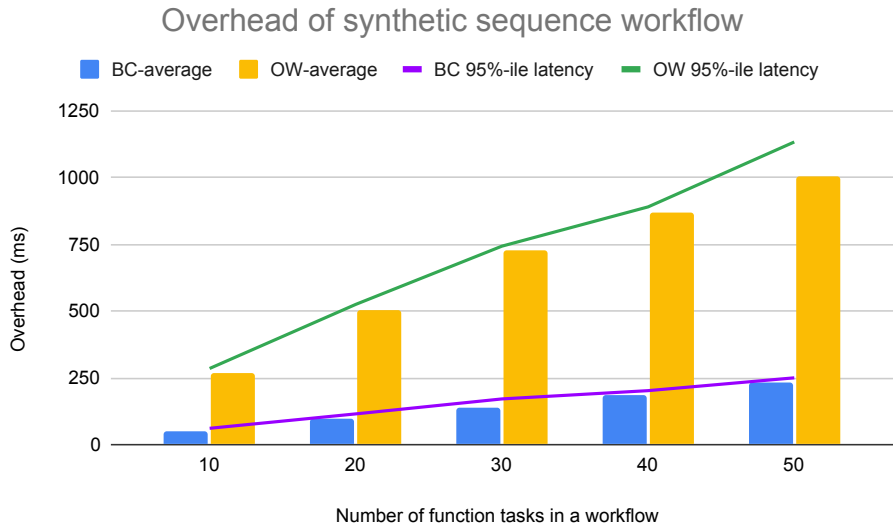

Figure 15 Overhead of Synthesize Sequential Workflows

tasks of the workflow. BriskChain sends the result of the current function directly to the next function of the workflow, without accessing the database unless any errors occur.

5.1.2 | 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.

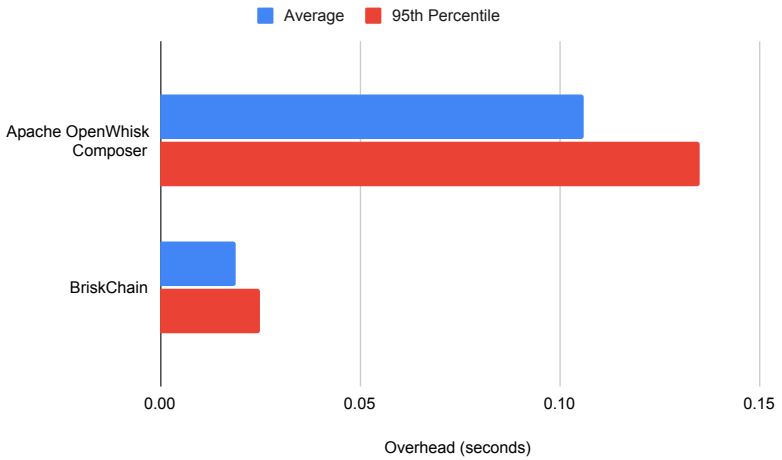

Figure 16 Overhead of Synthetic Branch Workflow

Figure 16 shows the experimental results. For these benchmarks, BriskChain vastly outperforms OpenWhisk Composer. BriskChain requires only 0.023 seconds of overhead on average, which is one-third that of OpenWhisk Composer. Furthermore, the 95-th 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.

5.1.3 | 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 17. 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.

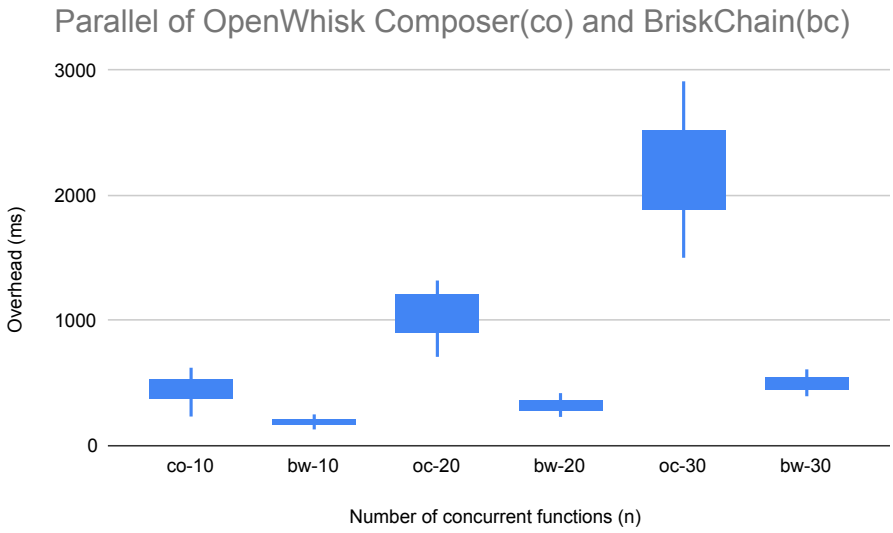

Figure 17 Whisker Plot of Parallel Workflow

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.

5.1.4 | 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 18 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 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.

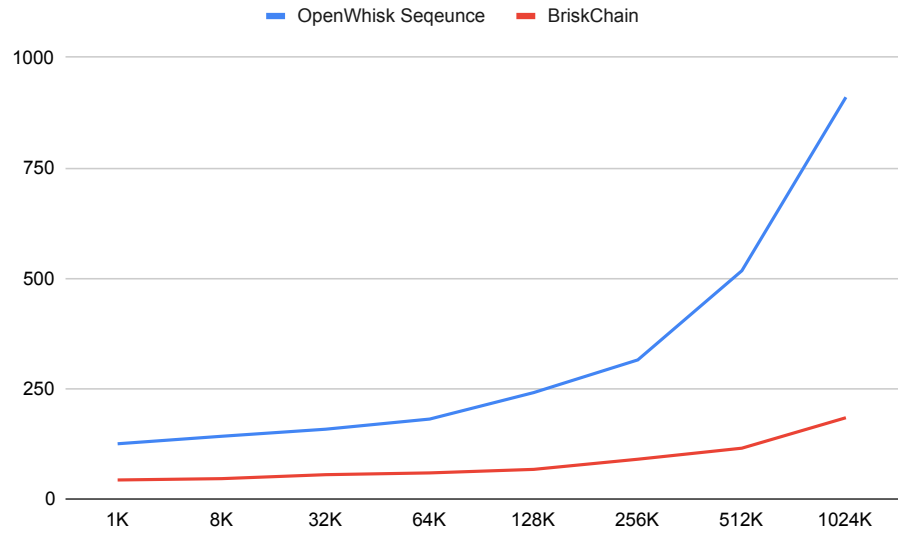

Figure 18 Overhead with Different Payloads

5.2 | Real-world Case Studies

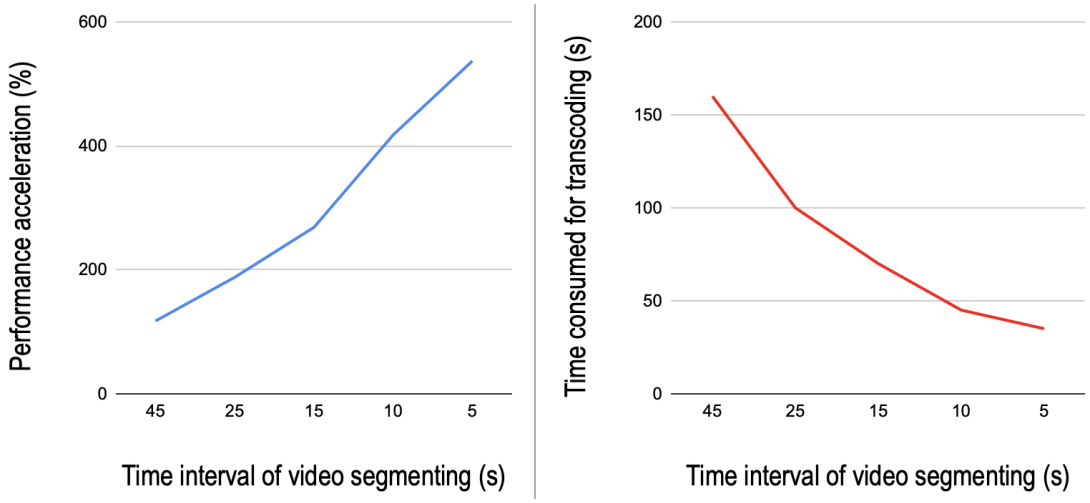

Figure 19 The Effect of Video Transcoding

Figure 19 shows the transcoding performance of the video transcoder workflow. In general, the time required to transcode decreases substantially as the duration of each segment decreases. When the video is split into two video segments of 50-second duration each, the workflow takes about 170 seconds to complete. But this process only takes 38 seconds if the duration of each video segment is 5 seconds.

With both applications, BriskChain is shown to be more efficient. Figure 20 shows a comparison of the overhead between BriskChain and Composer. This mirrors the results obtained through the synthetic workflow experiments. For Travis2slack, 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 5 seconds.

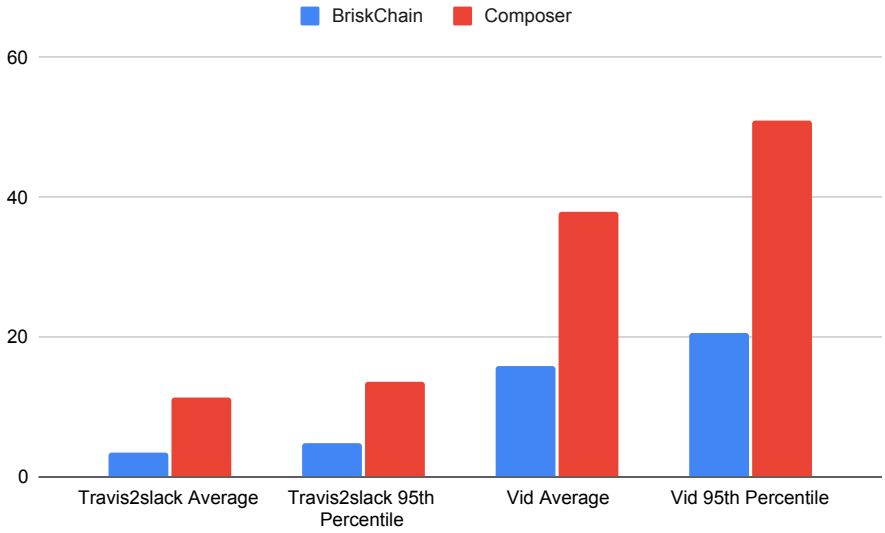

Figure 20 Overhead of the Real-world Applications

6 | CONCLUSIONS

The article presented a new design and implementation of worker-side pattern for serverless computing. It shows that by reducing the requirement for each function sandbox to communicate with a 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. In this way, serverless applications are able to run much more efficiently. Experimental results using our implementation, BriskChain, showed that this is indeed the case, for both synthetic workflows and real-world applications.

The concept of decentralized architecture holds promise for improving the efficiency of cloud computing, and represents a valuable area for further research. For instance, auto-scaling is an important feature of every virtualized container and is typically supported by the master in most FaaS systems. Decentralized containers could manage their own auto-scaling instead of relying on the master. This would involve each container having its own lifespan, with any inactive containers 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. Schleier-Smith J, Sreekanti V, Khandelwal A, et al. What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM* 2021; 64(5): 76–84.

2. Li Z, Liu Y, Guo L, et al. FaaSFlow: enable efficient workflow execution for function-as-a-service. In: ; 2022: 782–796.

3. Li Z, Guo L, Cheng J, Chen Q, He B, Guo M. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Computing Surveys (CSUR)* 2021.

4. Baldini I, Cheng P, Fink SJ, et al. The serverless trilemma: Function composition for serverless computing. In: ACM. ; 2017: 89–103.

5. Carver B, Zhang J, Wang A, Cheng Y. In search of a fast and efficient serverless dag engine. In: IEEE. ; 2019: 1–10.

6. Jonas E, Schleier-Smith J, Sreekanti V, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* 2019.

7. Adhikari M, Amgoth T, Srirama SN. A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Computing Surveys* 2019; 52(4): 1–36.

8. Ao L, Izhikevich L, Voelker GM, Porter G. Sprocket: A serverless video processing framework. In: Proceedings of the ACM Symposium on Cloud Computing. ; 2018: 263–274.

9. Fouladi S, Wahby RS, Shacklett B, et al. Encoding, Fast and Slow:{Low-Latency} Video Processing Using Thousands of Tiny Threads. In: USENIX Symposium. ; 2017: 363–376.

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

11. Jia Z, Witchel E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In: ; 2021: 152–166.

12. Apache Software Foundation . Apache OpenWhisk. http://openwhisk.apache.org; 2022.

13. Apache Software Foundation . Apache OpenWhisk Composer. https://github.com/apache/openwhisk-composer; 2022.

14. Travis CI to Slack. https://github.com/rabbah/travis-to-slack; 2022.

15. Idera, Inc. . Travis CI. http://travis-ci.com; 2022.

16. Slack Technologies, LLC . Slack. https://slack.com; 2022.