BRISKCHAIN: TOWARDS HIGH-PERFORMANCE SERVERLESS COMPUTING WITH DECENTRALIZED FUNCTION COMPOSITION

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER AND INFORMATION SCIENCES

Supervisors
Prof. Edmund M-K Lai
Dr. Jing Ma

Feb 2023

By
Kan Wang
School of Engineering, Computer and Mathematical Sciences

Abstract

Serverless computing (FaaS) is a promising cloud software development paradigm that consists primarily of serverless functions. Multiple serverless functions can be composited together as a workflow (function composition) to accomplish more complex tasks. As the serverless application grows, the serverless workflow becomes more and more important. However, many specificities of serverless functions are not considered by current workflow engines, which results in poor performance of workflows in scheduling serverless functions. This thesis studied performance issues of serverless workflow and the corresponding performance improvement strategies and proposed a decentralized worker-side pattern for high-performance serverless workflows.

Critical performance issues and associated performance improvement strategies for serverless workflows are investigated in detail. Firstly, due to the containerized isolation of each serverless function and the zero scaling capability of FaaS, serverless workflow suffers from a long cold start by the function's first run. Currently, warm-up strategies for standalone serverless functions are common, but performance optimizations for function warm-up in serverless workflows are lacking. The running paths of static workflows are predetermined in advance, so serverless function tasks in the workflow that can be warmed up are recommended according to the predefined workflow paths. Secondly, every serverless function is distributed in the cloud, and even functions within the same workflow interact across hosts. However, the functions of the same workflow are logically related to each other, and they interact with each other one by one. As a result, a close-locality strategy was proposed that enables local communication between functions of the same workflow. Thirdly, current workflow engines are master-side patterns, relying on a central controller to schedule serverless functions. However, in this pattern, each workflow function requires frequent remote interaction with its master resulting in performance degradation. Therefore, a decentralized worker model is proposed, where each workflow function can schedule itself from the current workflow node to the next workflow node without interacting with the master controller.

Based on the existing performance strategies, a new high-performance serverless workflow framework BriskChain is designed. BriskChain treats a serverless workflow as a whole unit the same as an independent serverless function. Therefore, workflows and functions can work together in the BriskChain system, and the performance of the workflow as a whole is optimized rather than focusing on the performance of individual serverless functions like other systems. Additionally, BriskChain orchestrates serverless functions based on workflow logic. For example, schedule functions of the same

workflow on the same host to avoid remote communication, or pre-warm serverless functions based on the workflow logic to avoid long cold-start issues. Moreover, BriskChain introduces an innovative worker-side pattern, a decentralized approach that allows each serverless function to schedule itself without a central controller.

BriskChain shows a high-performance result based on our evaluation. There are different synthetic micro-workflows and real-world application benchmarks with detailed comparative analysis of the overhead of BriskChain, OpenWhisk and Apache OpenWhisk Composer (Composer). Our experimental results show that BriskChain has 70% less overhead than its competitors when using sequential and branching DAG forms, and 50% to 70% less overhead than its competitors when using parallel DAG forms. Also, when the payload increases, the BriskChain overhead increases modestly compared to other workflow engines. In the real-world application experiments, BriskChain used only 30% to 40% of the total overhead of its competitors.

Contents

List of Tables

List of Figures

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.

Signature of student

Acknowledgements

I am grateful for the constant guidance and invaluable support from my two supervisors. They are Professor Edmund M-K Lai and Dr Jing (Julia) Ma. During the research process of the thesis, they helped me a lot, such as determining the topic and research methods, and the final review. Their enthusiastic help enabled me to learn a lot of techniques and research methods. It can be said that without their help, I could not successfully complete my research.

I would also like to thank my family for their understanding and encouragement. Research is long and hard work that can make a family's life even busier. I am extremely grateful to my family for their support and understanding so that my research can be completed on time.

Chapter 1

Introduction

This chapter starts with an introduction to serverless computing and composed serverless functions. A research topic on high-performance optimization for composing serverless functions is presented, and the motivation for this research is identified. Finally, the aims and objectives of the thesis are introduced.

1.1 Background and Motivation

1.1.1 Serverless Computing (FaaS)

Serverless computing is a promising software paradigm in the cloud field. Many cloud servers companies provide their serverless platforms such as Amazon Web Services (AWS) Lambda, Google Cloud Functions and IBM Cloud Functions. There are also plenty of popular open-source serverless computing platforms such as OpenWhisk and OpenFaaS. Serverless means not having to care about any servers in the application. Instead, serverless platforms are responsible for managing infrastructure and servers, so developers only need to pay attention to their own code and do not need to care about the corresponding running environments. Additionally, users are able to save a lot of

money and improve code quality due to the elasticity and auto-scaling capabilities of serverless computing.

Serverless computing is a function-based, stateless and fine grained software paradigm known as Function as a Service (FaaS). Each serverless function is an independent (self-contained) unit that performs a single atomic task. Also, FaaS is stateless since handling the function state is difficult for scaling purposes. In addition, Backend as a Service (BaaS) is commonly work with serverless functions together in serverless applications. For example, due to the stateless nature of FaaS, it mainly relies on BaaS storage services to maintain users' business data. Because of the close relationship between BaaS and FaaS, Jonas et al. (2019) define serverless computing as FaaS plus BaaS. However, there is still no standard definition of serverless computing, and some researchers may regard serverless computing as just FaaS.

FaaS has many advantages that some researchers even predict will dominate the next generation of cloud systems soon in (Schleier-Smith et al., 2021). Its benefits include, but are not limited to: (1) FaaS is cloud-based and has unlimited computing resources. (2) Serverless applications eliminate upfront commitments to cloud servers; pay as you go, even for short-lived resources. (3) Serverless has higher hardware utilization and avoids wasting idle resources (Jonas et al., 2019). (4) It is cost-effective because customers are not charged for idle resources. (5) It has a powerful auto-scaling capability and automatically adjusts resources. (6) It offloads much server administration work for developers and operators (DevOps) such as increasing or decreasing resources, so they only need to focus on their logic code. (7) Developers can use any program languages they like in serverless applications.

1.1.2 Serverless Function Composition (Workflow)

Function composition chains together multiple serverless functions into a new powerful service, which is important for complex serverless applications. The way to compose any two serverless functions is to connect the output of the previous function to the input of the next function. The advantage of this is to reuse existing fine-grained serverless functions and build more complex application logic. For example, Baldini et al. (2017) illustrates a workflow with two existing serverless functions: there is a serverless function that monitors the Travis continuous integration tool for build failures (Meyer, 2014), and another serverless function that can post notifications via Slack messaging service (Lin, Zagalsky, Storey & Serebrenik, 2016); then, a new simple workflow can be made using two already existed services, and it can notify those developers who break the build; so, new powerful services can be made using server-side glue logic and existing serverless functions.

However, applying function composition in a FaaS system is difficult to compare with a traditional monolithic application. The function composition is simple and efficient in common monolithic applications because they are mostly internal calls within the same host. In contrast, it is more difficult in the FaaS system, because each function is located in a distributed cloud running environment. Specifically, each serverless function is located in an unknown place in the cloud, and any two functions may be run at totally different hosts and contexts. Therefore, the serverless functions cannot call each other as quickly and directly as normal monolithic software.

The task of functional composition can be implemented by developers using manual coding, which is simple but inefficient. Manually coded functional composition is primitive and easy to understand, where developers write code that calls each serverless function in turn, following the logic of function composition. These manual codes run on a standalone server or in the client browser, and they integrate serverless functions

distributed in the cloud system. However, manual code strategies are tedious, unsafe and inefficient, burdening developers with all the work of composing serverless functions. Additionally, in fine-grained serverless applications where the number of functions increases, this inefficient manual effort further burdens developers. Therefore, it is possible to schedule the serverless function composition manually for coarse-grained microservices applications, but it is non-trivial to manually code of the function composition in a complex and fine-grained serverless application (Carver, Zhang, Wang & Cheng, 2019). For this reason, it is best to have a workflow engine to automatically schedule functions in serverless applications.

Using a workflow engine is efficient to implement function composition than developers manually writing functional composition logic in serverless applications. A workflow engine is a software application that automatically manages, schedules or monitors action tasks according to predefined rules, where each action task can even be located on a different system. By taking advantage of the workflow engine, distributed serverless functions can be scheduled and combined to implement complex tasks of the function composition in serverless applications, which is why the function compositions can also be called workflows. For example, a function composition task can be defined as a workflow composed of multiple serverless functions, and then run in the workflow engine to complete the complex functional composition task. This approach frees developers from the heavy and tedious coding of functional composition and implements functional composition in serverless applications through standard, maintainable workflow rules. Therefore, using a workflow engine is a more efficient way than manually writing function composition logic in the early days.

Nevertheless, the function composition is best done with a redesigned workflow engine dedicated to serverless systems rather than a traditional one. The ideas of workflow came before FaaS, and they did not take into account many of the special characteristics of serverless functions. Although a traditional workflow engine can schedule serverless

functions, it does not fit perfectly into serverless function composition. For example, traditional workflow engines orchestrate virtual machine-based instances, but serverless workflow orchestrates function-based instances. Most tasks of traditional workflow engines would handle load balancing and cluster requirements but the serverless workflow is not necessary to do it. The same opinion is also expressed by Akkus et al. (2018); Carver et al. (2019). They said that the traditional Serverful[1] workflow engines schedule task assignments and resource allocation under various objectives including load balancing, cluster utilization, fairness and so on. However, it is not the same case in FaaS which provide an unbounded amount of ephemeral resource and does not care about load balancing or cluster problems. The main tasks of the traditional server engines have shifted, so the serverless workflow mechanism and its optimization should be changed as well. Hence, there are emerging specially destined engines for serverless workflow such as AWS step functions, Google Workflows, IBM Composer and Azure Durable Functions.

Figure 1.1 summarizes the key terms related to the thesis topic and their relationships. These terms were introduced in the previous paragraphs and include Serverless Computing, FaaS, Function Composition, and Workflow. Essentially, these key terms are presented in a sequential order, with concepts ranging from broader to more specific. Serverless Computing is a vast topic, and within it lies the property of functional nature, which leads to the consideration of FaaS. FaaS enables the composition of serverless functions for the development of more complex applications. Function Composition becomes challenging in distributed cloud systems, as each function may be randomly located. To address this challenge, the concept of Workflows is introduced to manage Function Composition. However, traditional workflow engines need to be redesigned to cater to the specific needs of serverless workflows.

[1] Serverful: contrary to serverless, users need to manage facilities such as servers.

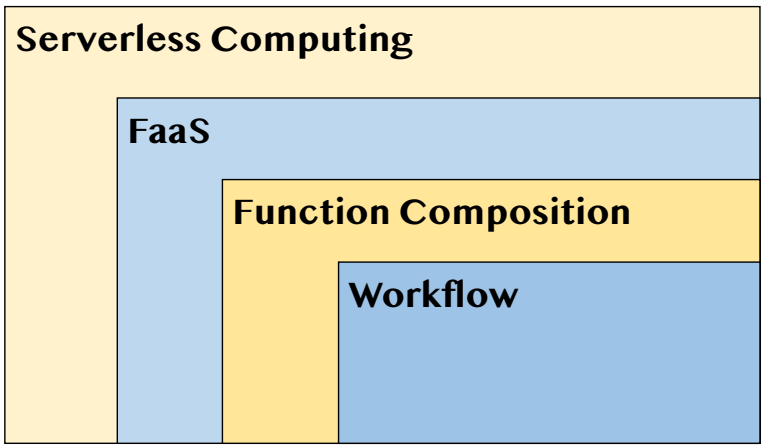

Figure 1.1: Key terms in the thesis and their relationships

1.1.3 Motivation for the Research

Although FaaS is promising, it is also facing severe challenges especially since its performance issues are more prominent. A Berkeley view on serverless computing by Jonas et al. (2019), which predicts a surge in FaaS usage but also faces the following challenging goals: (1) high-performance, affordable, transparently provisioned storage. (2) minimize startup time. (3) high-performance communication. (4) security. (5) new hardware architectures for the FaaS needs. Moreover, Hassan, Barakat and Sarhan (2021) conducted a survey listing the challenges researchers are most interested in, related to performance, security, debugging, cost and pricing. Based on the above academic resources, it can be seen that performance issues are getting more and more attention from most researchers.

Performance issues in serverless functions are also prevalent in serverless workflows, or worse. A serverless workflow is composed of different serverless functions, so the performance issue of each function is also an issue of its workflow. Additionally, performance issues with workflows can be exacerbated due to the cumulative effect of performance issues across all serverless functions in the workflow. For example, sandbox cold-start issues not only cause delays for individual serverless functions,

but cold-start issues for each function in a workflow add up to cause severe startup delays for the entire workflow. This cumulative workflow latency can sometimes be overwhelming for users, especially as the number of composite functions increases.

As serverless applications become increasingly complex, the need for research and resolution of performance issues in serverless workflows has become more urgent. Initially, a basic application may consist of only a few serverless functions, and the number of serverless workflows needed may be minimal. However, as the application grows in complexity, a larger number of functions are required. Consequently, complex workflows with multiple combined functions also increase within the system due to the expansion of business logic. Unfortunately, performance issues with serverless workflows can impede the development of FaaS (Function as a Service) serverless applications, transitioning them from simple to complex stages.

For example, Baldini et al. (2017) suggest that while serverless computing is economically attractive, it falls behind the state of the art in function composition. Therefore, optimizing function composition is a significant concern addressed in this thesis. Additionally, while a workflow may be composed of numerous serverless functions, real-world applications often require the workflow to be executed within a specific timeframe. A study analyzing Alibaba workload traces reveals that more than 50% of analyzed batch jobs, tasks, and instances are completed within 10 seconds (Cheng, Chai & Anwar, 2018; Guo et al., 2019). If the serverless workflow platform can efficiently handle these short, high-volume, and large-fan-out tasks, it will significantly enhance the applicability of cloud applications in the real world.

While there is currently some research on the optimization of serverless functions, there is a lack of optimization for functions in workflows. Current research focuses on the optimization of individual serverless functions in FaaS systems, such as function runtime optimization or function sandbox caching strategies. However, they do not pay sufficient attention to serverless functions within a workflow. There are close

relationships between the functions in the workflow, and these functions have logic, interaction, or communication with each other. The existing workflow engines treat each function of a workflow as an individual serverless function invocation without considering the performance of the entire workflow. In contrast, the concept of optimizing each workflow as a whole guides our optimization direction, with a focus on the workflow level rather than individual serverless functions.

Suppose there is a weather warning program consisting of three functions. Function A is used to receive the original information about weather changes. Function B is used to identify the person who needs to be notified of weather information. Function C is a specific notification method used to send the message to the intended recipient. In the traditional workflow, these three functions are simply connected sequentially: A is executed first, followed by B, and then C. A, B, and C do not collaborate or optimize with each other throughout the workflow. This highlights the need to fully consider the relationship between functions within the same workflow and optimize their performance at the workflow level. For instance, the warm-up preparations for functions B and C can be performed at the beginning of function A, rather than waiting for the completion of the previous function before starting the next one. Additionally, since each function in the distributed cloud system may be executed randomly on different distant hosts, causing remote communication delays, arranging the logically related function instances within an environment where the communication delay is minimal should be considered.

In summary, serverless workflows face thorny performance issues that need to be addressed, but the lack of such research motivates us to optimize the performance of function composition in serverless systems. Specifically, performance issues in serverless functions also exist in serverless workflows, or may even be worse. Additionally, as the number of functions in a serverless application increases, performance issues become more pronounced, hindering the growth of serverless applications that

use serverless functions. Therefore, the performance problem of serverless function composition needs to be solved urgently. On the other hand, most current research focuses on the performance optimization of individual serverless functions, while there is a lack of research on serverless workflows. We believe that optimization should also focus on the workflow level, not just the individual serverless function level. Therefore, the thesis conducts research on the performance optimization of function composition (workflow) in serverless applications.

1.2 Aims and Objectives

Our research aims to optimize the performance of function composition in serverless computing systems. This thesis will analyze the performance issues of function composition based on motivation. It will compare existing solutions and propose efficient strategies for functional composition, along with corresponding framework ideas for serverless computing. The objectives of this research are to propose effective optimization strategies or frameworks for serverless workflows, aiming to enhance the performance of serverless computing when composing serverless functions. These objectives have the following specific tasks: 1). Research workflow performance issues. 2). Investigate current performance optimization strategies for addressing these issues. 3). Analyse the performance optimization of serverless workflow and propose innovative strategies. 4). Design a new workflow engine (BriskChain) based on our proposition 5) Evaluate BriskChain to verify the effectiveness of our proposed method for performance optimization of serverless workflows.

1.3 Organization of Thesis

The thesis consists of five chapters including an introduction, literature review, research-related methodology, evaluation and analysis, and conclusion. The first chapter introduces the research motivation, topic, and objectives. This chapter leads out the research topic which is the performance optimization of serverless function composition in cloud computing. The second chapter is a literature review of serverless workflow performance issues and corresponding optimization methods. Specifically, the work of other researchers is highlighted and their results are thoroughly analyzed and compared. The third chapter is to review research related methodology as well as proposes a new methodology for the thesis, which includes performance optimization strategies for high-performance serverless workflows and related workflow engine ideas. The fourth chapter is to evaluate and analyses the proposed Briskchain methodology, which verifies and analyses the effectiveness of our proposed optimization methods. Specifically, the chapter conducts experiments using several synthetic experimental benchmarks, followed by experiments with two real-world applications. Based on the experimental data results, the effectiveness of the serverless workflow optimization methods is analysed and verified. Finally, the last chapter of this thesis provides a summary of this study and attaches future research directions inspired by this study.

Chapter 2

Review of Serverless Function and the Function Composition

This chapter is a literature review on FaaS, serverless workflows, the performance issues and optimization strategies. The underlying technologies of FaaS and serverless workflows are thoroughly investigated, and their critical performance issues are found as well. Then, performance optimization strategies related to the issues are analysed and compared in detail.

2.1 The Review of Serverless Function and FaaS Systems

The virtualized sandbox is an isolated environment for each serverless function. It acts as a container that encapsulates and isolates the code of a single serverless function, each function must be packaged into its own virtualized container. Currently, there are several commonly used virtualization mechanisms for serverless sandboxes, including Unikernel, FireCracker, gVisor and Docker. For instance, AWS Lambda utilizes

FireCracker as the sandbox to isolate its serverless functions. Additionally, many open-source FaaS systems, such as OpenWhisk and OpenFaaS, employ containerised sandboxes like Docker to encapsulate serverless functions.

The sandbox provides an independent runtime environment for each serverless function. Serverless functions are essentially business code that cannot run by themselves without a runtime environment. However, virtualized sandboxes - such as Docker containers - can provide a lightweight, standalone, and executable environment for serverless functions. These sandboxes contain the necessary system tools, libraries, or configuration files that are required for the serverless function to run.

The containerized serverless function is the basic unit of a serverless application. Each containerized serverless function consists of a fine-grained function code and a container, making it an autonomous and independent entity. Therefore, multiple containerized serverless functions are used to build complex serverless applications. Figure 2.1 shows how serverless code can be used in a FaaS system. When designing a serverless application, developers write logic code following the specification of serverless functions. In FaaS runtime, each function is loaded into a sandbox to run, such as in the popular Docker container, and there are many such containerized functions in a FaaS application.

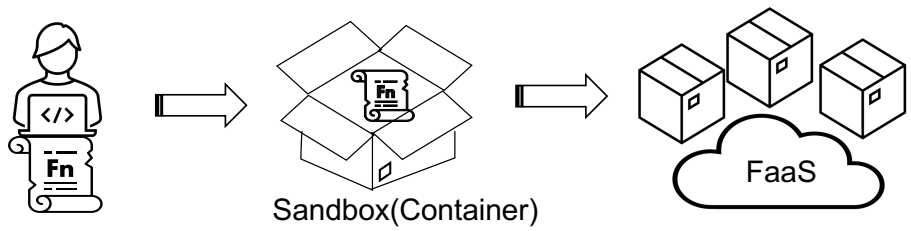

Figure 2.1: Containerized Serverless Functions

In addition to serverless functions, there are related components such as gateways, events, and BaaS, which together constitute the FaaS System. Figure 2.2 demonstrates a typical structure of the serverless system, and the serverless functions are located

in the middle of the figure. The function services can be exposed externally through API Gateway or Event. For example, users can call the serverless functions through API gateways. Serverless functions can also rely on Backend as a Service (BaaS) for additional cloud support. For instance, AWS S3 and Google Cloud Storage are BaaS storage services that can be used for persistent storage services required in the business logic of serverless functions.

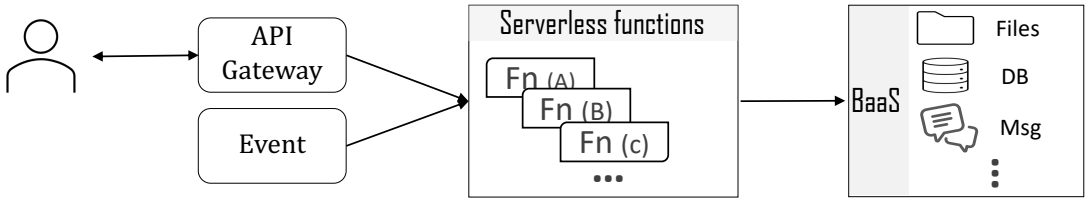

Figure 2.2: A Typical Structure of FaaS and Related Components

The FaaS system, with its highly service-oriented nature and a multitude of fine-grained functional services, stands out among other systems. Figure 2.3 illustrates the different levels of granularity, depicting the progression from monolithic applications to microservices, and finally to FaaS. Traditional monolithic applications are often difficult to decompose as they heavily rely on specific and unified running environments. Although these applications are constructed from various small modules, their strong interdependencies prevent them from running independently. In contrast, the prevalent microservices architecture follows a service-oriented approach by leveraging virtualization technology to create independent services that can operate autonomously. An application built using microservices can be composed of multiple independent and self-contained microservices, each representing a distinct runnable system. With the aid of virtualization technology, FaaS utilizes containers to provide standalone functional services. However, FaaS sets itself apart by adopting a fine-grained functional service architecture instead of the coarse-grained microservices typically found in traditional microservices architectures.

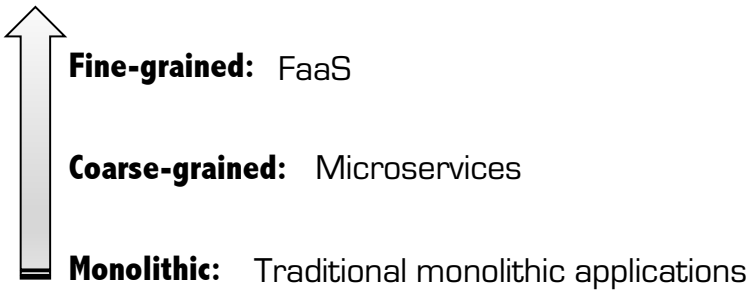

Figure 2.3: FaaS with Fine-grained Abstractions

The architecture of FaaS can be divided into four layers, from machinery computer resources to end-user applications (Li et al., 2021). Figure 2.4 shows the layers such as virtualization, encapsulation, system orchestration, and system coordination from the bottom to the top respectively. The virtualization layer abstracts physical machines into virtual resources. Currently, container-based isolation is commonly used in this layer, and a container is a sandbox or basic running unit that use to wrap up a small piece of code. The next encapsulation layer manages the virtual containers. Its tasks include container pool managing or container prewarming. The orchestration layer focuses on the application tasks that schedule resources and run sandboxes efficiently, and load balancing and monitoring tasks as well. The top layer faces end-users and other BaaS systems. For example, API Gateway is located in this layer to help FaaS expose its services to end clients.

2.2 The Review of Workflow Engines and DAGs

A Directed Acyclic Graph (DAG) is a directed graph with no directed cycles, and it is supported by many serverless workflow engines. A DAG consists of vertices and edges, with each edge pointing from one vertex to another. The vertices, also known as nodes, are serverless functions in the workflow of FaaS, and the edges of DAG are workflow logic. There are three typical DAG forms: sequence, branch and parallel. These forms

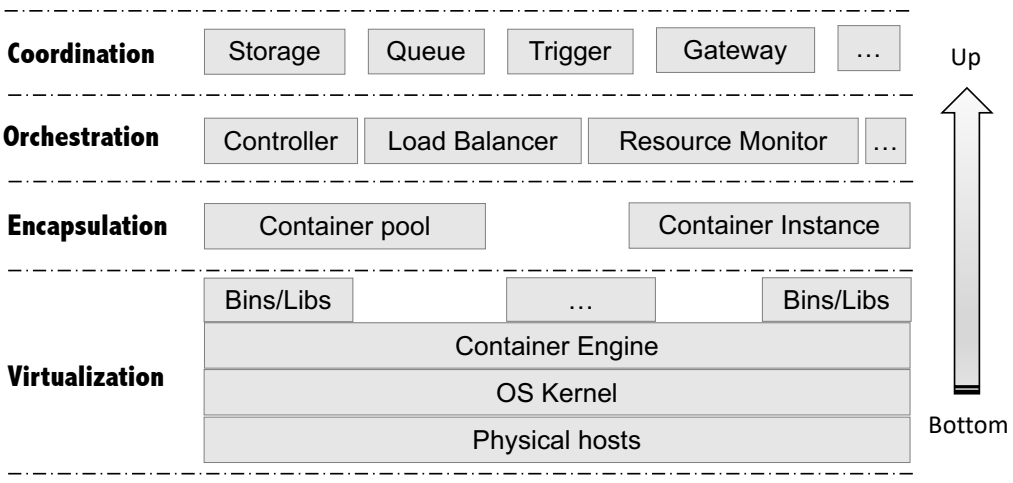

Figure 2.4: General Implementation of the Serverless Architecture

are supported by many renters such as AWS Step Functions, Azure Durable functions and IBM Cloud Functions. In DAG, the input of the current node depends on the output of its previous node (Adhikari, Amgoth & Srirama, 2019), and a function should wait until all previous functions have finished (Wu, Mi & Xia, 2020). For example, OpenWhisk uses its sequence feature to chain actions[1] together to finish sequential workflow. In addition, the branch flow decides one direction from several possible branches, and the parallel flow does several tasks simultaneously. Workflow systems use a centralized scheduler for task assignments and resource allocation (Carver et al., 2019). Figure 2.5 shows a typical workflow engine scheduling serverless functions. It can be seen from the figure that the workflow engine and FaaS are two completely independent systems although they can cooperate to complete the workflow tasks. FaaS is responsible for executing serverless functions that can actually be thought of as tasks in a workflow, and the workflow engine is in charge of scheduling these serverless functions according to predefined workflow logic. For example, a FaaS-based workflow can typically be accomplished as follows. 1) Engineers build the serverless functions needed for workflows and deploy them to FaaS systems. 2) Engineers design workflow logic based on existing serverless functions, and upload the completed workflow definition to the

[1]The term of actions in OpenWhisk are the same as the functions in FaaS

workflow engine. 3) At runtime, the workflow engine schedules serverless functions based on predefined workflow logic.

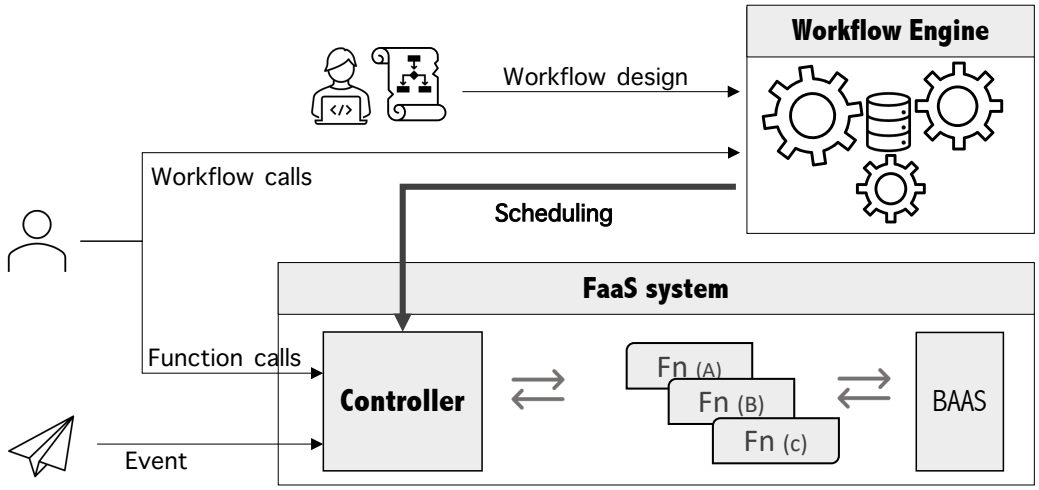

Figure 2.5: Traditional Workflow Engines with FaaS

Most serverless workflow engines work as a separate external system to FaaS like the process described in the previous paragraph. For example, Amazon Step Functions is a workflow engine that orchestrates Lambda functions into workflows. In this situation, Amazon Step Functions and AWS Lambda are two independent systems. There is a substitution principle of FaaS systems that a serverless workflow itself can be a serverless function (Baldini et al., 2017). Obviously, Amazon Step Functions does not obey the substitution principle because a Step Functions workflow and a Lambda function are different things in the two systems and one cannot be replaced by another.

In contrast, a workflow engine can be integrated into a FaaS system, working in close collaboration. Figure 2.6 illustrates a potential architecture where a workflow engine is embedded within FaaS. In this scenario, the FaaS controller uniformly handles both serverless functions and workflow jobs, treating them as equivalent entities. By adhering to the substitution principle outlined in (Baldini et al., 2017), this approach ensures seamless composition of workflows and serverless functions. It is worth noting that this design is relatively new, and only a few workflow engines, including our

innovative BriskChain, currently employ this methodology. More detailed information about BriskChain will be provided in subsequent chapters.

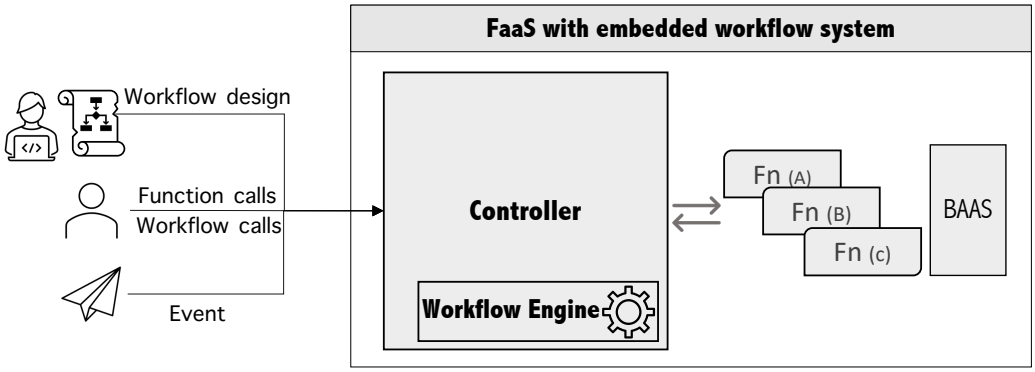

Figure 2.6: FaaS with Embedded Workflow Engine

2.3 The Performance Issues

This section discusses the performance issues associated with function composition in serverless applications. After conducting a thorough literature review and detailed analysis, we identified three critical performance issues. These issues not only affect each individual serverless function (FaaS), but also impact the performance of compound functions (workflows).

2.3.1 Startup Latency Issue

Serverless functions experience significant delays during startup delays, which is the duration between the user invoking the function service and the function actually executing(Daw, Bellur & Kulkarni, 2020). These delays are primarily caused by the zero-scaling capability of FaaS, where inactive serverless functions consume zero resources. Scaling to zero helps reduce resource costs for idle functions. However, it introduces a cold start issue when new service requests arrive after the sandbox is

deactivated. Specifically, cold start involves a lengthy startup latency as serverless functions need to establish their running environments, including containers, libraries, and other dependencies. According to Jonas et al. (2019), there are three main challenges contributing to the startup time of serverless functions: (1) scheduling and launching the sandboxes (containers) for the serverless functions, (2) downloading the application software environment required to execute the function code, and (3) performing application-specific startup tasks, such as loading and initializing data structures and libraries.

The startup latency is a bottleneck in FaaS and serverless workflows. Studies have estimated that a cold start takes up to 70% of a serverless function's lifetime (Du et al., 2020), and (Shahrad, Balkind & Wentzlaff, 2019) reports that initialization of a function related resources needs at least 500 milliseconds. Moreover, since each function of a workflow is suffered from a startup delay, the workflow response can last an unacceptably long time on the client side because the total processing time is accumulated by each function. In addition, the total startup time will be extended further when the number of functions is increased in a workflow.

2.3.2 Interaction Issues between Functions

Slow inter-function communication can cause performance issues in function-intensive serverless applications. The performance issue of interaction between functions is not prominent in traditional monolithic systems, because they can efficiently communicate and interact in the same host environment. However, it is not the same case in a distributed cloud environment like FaaS. Serverless applications are decomposed into multiple separate functions, distributed across different hosts. Also, internal communication between the functions is now turned into remote network invocations which introduce considerable communication delay. Therefore, it matters in serverless platforms that the

interactions between the serverless functions are remote and causes significant latency than local connections.

The low-performance issue of interactions between serverless functions is a critical obstacle in serverless workflow applications. Due to the nature of the distributed serverless functions, there are overhead problems related to remote communications. In the case of serverless workflow, there are increasing demands for internal interaction between functions in the same workflow because each node (servers function) of the workflow has to interact with the previous or next node of the workflow. Unfortunately, these internal requirements are remote calls in the FaaS system. For example, PyWren uses remote invocation with many fine-grained serverless computing tasks, and it involves significant overheads (Carreira, Fonseca, Tumanov, Zhang & Katz, 2019; Jonas, Pu, Venkataraman, Stoica & Recht, 2017). Moreover, since the distributed architecture in FaaS, many FaaS vendors have a quota for data transmission. For example, AWS Lambda limits the data transformation within 6MB synchronous or 256K asynchronous per request (Li et al., 2022). This will hinder the communication ability between the functions in a workflow. Each serverless function in a workflow needs to pass its output data to the next function in the workflow; if there are workload quotas for the communication transfer between the functions, many workflow applications which exceed the quota will not be able to run on the system.

2.3.3 Poor Communication Patterns

Jonas et al. (2019) indicate FaaS has poor performance for standard communication patterns. They compared three common communication patterns (broadcast, aggregation and shuffle) in both Virtual Machine (VM) based and FaaS solutions. In their research, VM instances process and combine the inner data locally before sending it to another host. In contrast, every serverless function with a payload must interact frequently on

distributed remote hosts. Hence, the current communication patterns of FaaS have low performance. For example, Figure 2.7 demonstrates the difference in broadcast patterns in VM and FaaS-based systems. It can be seen that the FaaS-based system has more frequent remote calls than the original VM-based systems. If *N* is the number of VM hosts, the communication complexity of operations is $O(N)$ to the VM-based system in the figure samples. However, the complexity is increased to $O(N \times K)$ in the FaaS system where *K* is the number of functions per host (Jonas et al., 2019).

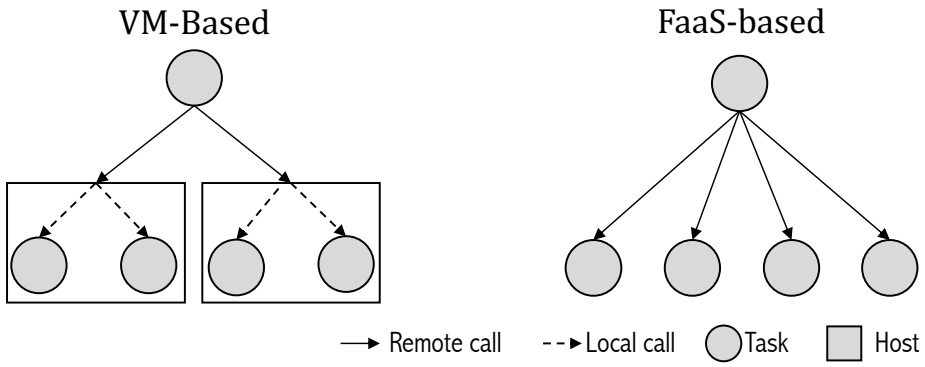

Figure 2.7: The Communication Path of Broadcast Pattern in VM and FaaS Based System

Poor communication patterns are critical issues existing in the workflow as well. A real-world application is often complex involving intensive interactions between smaller tasks (Daw et al., 2020), and the functions of workflows need to collaborate with each other. However, current interact patterns lead the fine-grained functions heavily and inefficiently for the inner interaction (Li et al., 2021), especially for the case of the workflow. For example, the composed functions must frequently interact with their central controller or the workflow engine to exchange internal states, so the centralized workflow pattern increases the connection burden between the inner functions.

Most workflow systems use the master-side pattern, but it leads to low performance in serverless workflows. The pattern relies on an important and powerful workflow

engine as the master, which schedules, allocates or balances tasks and resources. Correspondingly, there are many workers (tasks) around the master who accept management from the master. Currently, the majority of workflow systems adopt the master-side pattern approach (Adhikari et al., 2019; Ao, Izhikevich, Voelker & Porter, 2018; Fouladi et al., 2017; Malawski, Gajek, Zima, Balis & Figiela, 2020). The related studies include but are not limited to Ao et al. (2018); Balis (2016); Fouladi et al. (2017); Mahgoub et al. (2021); Malawski et al. (2020). Unfortunately, due to high scheduling overhead and frequent data movement, the master-side pattern does not perform well in the composed serverless functions(Li et al., 2022). The reason is also related to the poor interaction patterns between the serverless functions (Jonas et al., 2019). Specifically, each task (worker) of a workflow is a serverless function that needs to interact with its remote master frequently; if one serverless function needs to transport data to another function, it must move the data to the master side first. Therefore, the master-side pattern is a low-performance pattern in workflow systems although it is widely used in traditional workflow systems.

2.4 The Performance Optimization

This section examines performance optimizations for serverless workflows and a comparison and critical analysis of various optimization strategies. The current work on performance optimization is thoroughly studied, especially the solutions to the problems mentioned in the previous section. It turns out that each optimization strategy solves different problems from different perspectives, and they have different advantages and disadvantages.

2.4.1 Sandbox Startup Optimization

Cold start latency is a bottleneck in serveries workflow performance issues (see 2.3.1) that has attracted many researchers. Techniques to minimize the cold start problem while still scaling to zero are critical (Castro, Ishakian, Muthusamy & Slominski, 2019). Paradoxically, if the sandbox instance is not stopped, there will be no cold start problem; however, the serverless system thus also loses the advantage of scaling to zero on idle resources. Therefore, many researchers try to find fast-start sandboxes or use warm-start mechanisms to avoid long-time cold-start problems. These optimizations are performed at the virtualization layer at the bottom of the serverless architecture in Figure 2.4.

1) Fast-startup Sandbox

Some researchers make great efforts to investigate a fast-start mechanism for sandboxes, but they face with the challenge of a tripartite trade-off. Currently, there are sandbox isolation methods such as using Docker, gVisor, Kata, Firecracker, or Unikernel. Whatever which one is used, it is difficult to possess the following three critical advantages simultaneously: strong isolation, strong flexibility and low startup latency. The isolation determines the running environment which is important to the sandbox security. For example, each sandbox should not interfere with the other. The flexibility indicates testing/debugging abilities and additional support for extending over the system. For example, the Unikernel model is not flexible for applications since it is not adaptable for developers once built. Table 2.1 shows the trade-offs among the three concerns within different virtualization mechanisms (Li et al., 2021). It can be seen obviously from the table: strong flexibility and isolation level will lead to long startup latency. In contrast, if an isolated sandbox has lower startup latency, then its flexibility or isolation advantage could be lost. For example, a traditional VM has a strong isolation mechanism, but has more than 1000ms startup latency which is the biggest latency value; Unikernel isolated

virtualization has great isolation and short startup latency, but it loses flexibility; Google and Amazon developed their own sandboxes with strong isolation and flexibility but code-startup time still not expected. In summary, researchers are still struggling to reduce the start-time issue meanwhile guarantee strong isolation and flexibility to run a sandbox.

Virtualization	**Kernel**	**By**	**Startup**	**Flexibility**	**Isolation (Security)**
Unikernel	built-in	Docker	10ms-50ms	Weak	Strong
FireCracker	unsharing	Amazon	50ms-500ms	Strong	Strong
gVisor	unsharing	Google	50ms-500ms	Strong	Strong
Docker	host-sharing	Docker	50ms-500ms	Strong	Weak
VM	unsharing	-	>1000ms	Strong	Strong

Table 2.1: The Trade-offs between Startup-latency, Flexibility and Isolation in Different Virtualization Mechanisms

2) Sandbox Warm-up

In addition, using the warm-up strategy can avoid the cold start of the sandbox, thereby reducing the sandbox startup time. The code start issue, described in the section 2.3.1, causes severe delays for FaaS applications. The latency is related to several sideworks such as the cold container start, system library loading or application library loading. Many researchers had experimented and shown that the cold start consumes much more time than a real customer's logic. For example, launching a containerized sandbox takes about 100+ milliseconds, but the function code may only execute in a few milliseconds(Du et al., 2020). Current researchers address this issue by prewarming containers, so the serverless functions can always have ready-to-used sandboxes to run the functions.

FaaS systems typically have warm-up strategies for individual serverless functions

but are rarely warm-up optimized for workflows. The system can decide which sandboxes need to be preloaded or cached based on how often the function is used. This avoids cold start problems and improves the performance of function calls. For example, frequently used but idle sandboxes may not be destroyed immediately but cached in a pool for later use. However, this strategy does not take into account the specificity of workflows, which have predefined running routes for each workflow task (serverless function). Due to the predefined workflow logic, the sandbox can be warmed up along the run route specified by the workflow definition. For example, when a running process comes through in a specific node (sandbox) of a workflow, the node has been warmed in advance according to the flow route. For the above reasons, it is expected to prewarm the sandboxes according to the predefined route of the serverless workflows.

There are static and dynamic forms of workflow optimization with the warm-up strategy(Bhasi et al., 2021), but the dynamic form is difficult for the sandbox prewarming. A static workflow means the connections among the function tasks are determined in a workflow design-time, and will not be changed during the workflow runtime. In contrast, a dynamic form dynamically changes the running route of the chained serverless functions in a workflow (Wu et al., 2020). Currently, sandbox warm-up with static workflows is simple because the DAG engine can warm up the sandboxes with known run paths. For example, the sequence and parallel DAG forms are easily prewarmed according to a predictable route. However, the real route of a dynamic workflow is unknown before its runs, so the workflow engine is difficult to pre-warm the sandboxes of dynamic workflows. For instance, branch DAG forms are difficult to warm up because only one of the branches will be used. Prewarming a wrong branch will waste warmed sandboxes, and the system also has to correct them in a hasty time.

Fewer researchers try to predict running routes in dynamic DAGs. Xanadu (Daw et al., 2020) estimates the running path of dynamic DAGs using a strategy called most-likely-path (MLP). Xanadu pre-warms instances by a speculative-based approach and

makes a just-in-time resource provisioning. Kraken (Bhasi et al., 2021) is another research using different predictive strategies in dynamic DAG. The authors of Kraken argue that Xanadu will provision 32% containers (sandboxes) when subject to moderate and heavy loads. Instead, Kraken uses Variable Order Markov Model which has a probability-based policy with weights parameters for workflow-aware prewarming. However, Li et al. (2021) believes that no matter what kind of prediction strategy is used, it will always cause some wrong warm-up, and then introduce additional waste of resources in the multi-branch scenario of DAG.

3) Fast-startup Sandbox Summary

Both fast-start and warm-up sandboxes can reduce serverless function startup latency, but they each have their pros and cons. First, some sandboxing techniques with fast startup properties can alleviate the problem of startup delays (such as Unikernels), but such sandboxes may lose the flexibility or security of serverless functions. Second, the prewarming strategy is needed not only for individual functions but also for a serverless workflow. Since the workflow has a predefined running route, each function of a workflow can be pre-warmed according to the route precisely. Third, there are static and dynamic DAG flows, and the prewarming for static DAG flows is easy, safe and efficient. However, the prewarming for dynamic DAG flows suffers resource waste because of the error prediction.

2.4.2 The Close Locality Strategy

Close locality strategy improves performance and avoids the inefficient communication problem between the serverless functions mentioned in the section 2.3.2. The thesis uses the term Close Locality, or Locality for short, to denote a strategy that uses fast local communication to improve the performance of interactions between functions. The

reason for this is that metadata exchange may continually happen between serverless functions in an application, and the long-distance data exchange between distributed serverless functions can cause severe performance degradation. In contrast, if any two functions with data dependency are scheduled on the same physical host, the data transmission can be significantly reduced by middle-wares (Li et al., 2021). Moreover, there is a strong logical relationship between the functions and functions in the same workflow, so there will be strong interaction requirements between these functions.

However, Li et al. (2021) indicates that the current serverless system is a data-shipping architecture, which sends data to the code host to parse instead of sending code to the data host. Thus, on the one hand, a serverless system cannot guarantee that the data stored and the workers scheduled are just in the same physical node. On the other hand, frequent code transferring should also be avoided due to security and privacy concerns. Li et al. (2021) mentioned that improving data locality can effectively reform the application design from a data-shipping architecture into a code-shipping one that improves performance and avoid inefficient communication, but they only mentioned this idea in their paper without further details.

Akkus et al. (2018) use a similar strategy that orchestrates function executions of the same application as locally as possible. They proposed SAND which allows serverless functions from the same application to be co-located in the same container. Instead of the usual containerized isolation of each serverless function, they utilize different processes to isolate each serverless function code and then group those functions into the same container. Thereby, SAND can accelerate the communication performance because the functions in the same container acquire locally high-speed communication advantage. For example, Carreira et al. (2019) indicates that AWS Lambda can only sustain dramatically low bandwidth with 60MB/s, but if the functions can communicate locally then the bandwidth will be extremely large to 1GB/s. However, SAND has weaker isolation than common containerized isolation. It is a trade-off concern between

startup time, flexibility and isolation level mentioned in the section 2.4.1. In the situation of SAND, the trade-off means a high-performance sandbox may either be weak isolated or lose flexibility.

The locality idea is also used in cloud regions. GlobalFlow (Zheng & Peng, 2019) considers a geographically distributed scenario, where functions reside in one region and data in another region. It groups the functions in the same region into sub-graphs and connects them with lightweight functions, so it improves data locality and reduces transmission latency. However, their strategy is not common because it is too limited in specific cloud providers or environments.

We believe that the concept of the locality strategy should also be applied to serverless workflows. In the workflow environment, functions belonging to the same workflow should be scheduled on the same cluster node, which is similar to using the same host. By orchestrating the functions of the same workflow on a single host, related functions can efficiently communicate with each other locally. Figure 2.8 illustrates the difference between a serverless workflow with and without the application of the locality strategy. On the left side of the figure, the functions of the same workflow are located on the same cluster node, avoiding cross-host interaction between these functions. In contrast, on the right side of the figure, the serverless functions are randomly located on different cluster nodes, resulting in heavy internal interaction overhead between these functions.

In summary, current locality strategies differ in how related functions are grouped and what locality should be used. Both GlobalFlow and SAND group together serverless functions of the same application. The locality place of GlobalFlow is the cloud region, and SAND uses the same container. Therefore, GlobalFlow schedules application functions to reside in one cloud region to avoid long-distance cross-region communication overhead. SAND schedules application-level serverless functions in the same container to improve the interaction performance between functions.

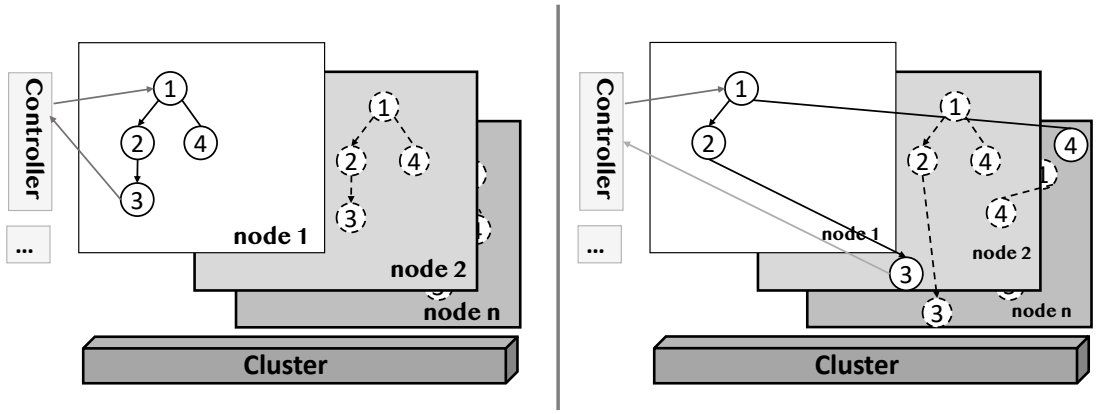

Figure 2.8: Comparison of serverless workflows with (left) and without (right) locality strategies

2.4.3 The Worker-side Pattern

The poor communication patterns and the weakness of the master-side pattern had described in section 2.3.3. In contrast, a decentralized worker-side pattern can be a way to improve performance. The main idea of this pattern is to offload the overhead from a central master side to multiple worker side (Li et al., 2022). A few studies have emerged using similar ideas in recent years.

Jia and Witchel (2021) present Nightcore which is an efficient serverless computing framework. Firstly, Nightcore orchestrates chained serverless functions of a workflow into the same host. Nightcore then relies on a gateway in each worker host to schedule tasks within that host, where the gateway acts like a smaller workflow engine proxy that takes over some tasks from the original master controller. As a result, Nightcore offloads many of its scheduling tasks from the single master-side controller to gateway proxies residing in each worker host. In this way, the inner calls between chained functions can be processed locally and efficiently because they are located in the same host. Also, their internal data can be transmitted, interacted and preprocessed by the local gateway first, without relying on the remote workflow engine. Consequently, Nightcore can significantly reduce the overhead of serverless workflows.

It looks like Nightcore is similar to a worker-side pattern. However, Li et al. (2022) argues that the Nightcore still falls back to the master-side pattern because its gateway serves as the centralized manager to assign tasks and allocates execution states.

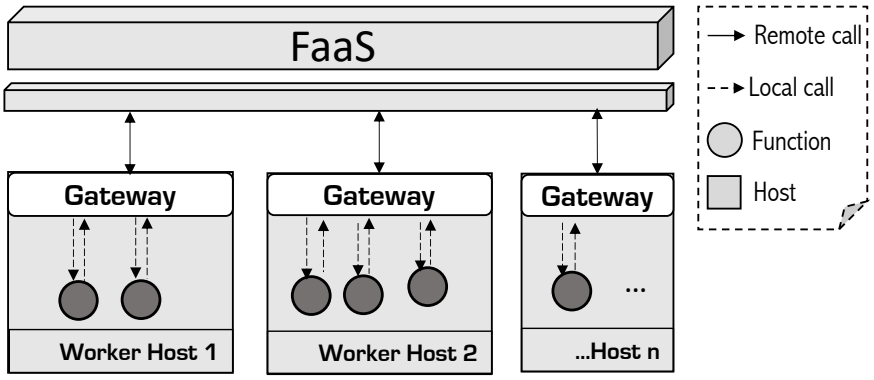

Figure 2.9: Worker-side Pattern Structure in FaaSFlow

Li et al. (2022) propose FaaSFlow which is closer to a worker-side design than Nightcore did. Figure 2.9 shows the basic structure of FaaSFlow. In the figure, there is a decentralized workflow engine (Gateway) that worked as a scheduler located in each worker host as Nightcore did. 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 totally in FaaSFlow. Thereby, as Li et al. (2022) publicized that their FaaSFlow is a worker-side decentralized serverless workflow pattern.

However, Nightcore and FaaSFlow are not thorough worker-side patterns. Although they offload the overhead from the centralized master to the worker sides, a proxy engine located in each worker host is still considered as a small regional master (engine). Their worker-side model relies on a host-level engine (gateways) to handle internal interactions between serverless functions within the same host, and between the gateways to handle remote cross-host interactions. We believe that the worker-side pattern should be further decentralized, and the worker-sides should have the ability to schedule themselves according to predefined workflow logic without any level of master (engines). For

example, when the start function (workflow node) is initiated in a workflow, the end-to-end serverless functions should have the ability to schedule themselves one alter another without relaying any central controller again.

2.5 Summary

There are several essential backgrounds before optimizing serverless workflow performance. FaaS system mainly consists of fine-grained functions, and each function is then wrapped into a virtualized sandbox. Virtualization is the core of the FaaS system, which is used to isolate each function providing a running environment for them. Therefore, each serverless function can run safely, independently, and scalably on the cloud. Moreover, there are independent workflow engines and embedded workflow engines of FaaS systems. Currently, most serverless workflow engines are independent and outside of the FaaS system. This is similar to a traditional workflow engine, where the workers and the workflow scheduling engine run in two different systems. In contrast, based on the particularity of the serverless workflow, the serverless workflow engine may also be embedded in the FaaS system. Thus, both serverless functions and workflow tasks can be scheduled in one unified system.

Table 2.2 summarizes the three performance issues studied in the literature review. Solutions to these problems are further summarized in the following paragraphs. Based on a study of the literature, three key performance issues of serverless workflow were identified that hinder the adoption of FaaS. First, due to the containerized isolation of each function, serverless functions suffer from a long time for cold start of the sandboxes when the functions first run. This situation can be worse in the case of function composition, as the workflows suffer from accumulated cold-start delays. Second, although the workflow functions are logically related to each other, in fact, each distributed function runs on different physical hosts. This incurs remote cross-host

communication overhead between individual workflow functions or between a function and the workflow engine. Third, current master-side communication patterns involve heavy overhead in the workflows of FaaS. The master side relies on a control controller (master) to schedule multiple function tasks (workers). Each task worker in the pattern must communicate with its master in order to step end-to-end tasks for the workflow job. Moreover, to the natural principle of FaaS, serverless functions are more numerous and fine-grained than traditional workflow tasks. As a result, numerous fine-grained serverless functions lead to frequent interactions between the master and workers for workflow scheduling and ultimately degrade performance.

Issues	**Solutions**	**Shortcomings of current research**
Startup Latency [2.3.1]	Quick-start sandbox [2.4.1] Warm-up strategy [2.4.1]	This strategy faces a three-way tripartie challenge. The strategy will inevitably lead to false warm-ups in dynamic workflows.
Interaction Issue [2.3.2]	Locality Strategy [2.4.2]	There is locality strategy in SAND and GlobalFlow, but this strategy has not been used in serverless workflows.
Poor communication patterns [2.3.3]	Worker-side pattern [2.4.3]	Nightcore and FaaSFlow initially adopted the worker-side pattern, but it is not a complete decentralization solution.

Table 2.2: Performance issues and solutions for current workflows

Aiming at the above three performance issues, there have been related research. The first issue can be addressed by designing a quick-start sandbox or sandbox warm-up strategy. Creating a fast-start container is the most straightforward way to speed up boot time for sandboxes. However, it faces a tripartite trade-off challenge and cannot guarantee the following three advantages at the same time: strong isolation, security, and low startup latency. The prewarming strategy simply warm-up sandboxes previously. The static workflows have deterministic running paths, so their serverless functions can

be warmed up effectively. However, the real running path of dynamic workflows can only be determined at runtime, and their warm-up will inevitably lead to resource waste and efficiency regression.

The second issue can be addressed by a short-location strategy which avoids low performance of long-distance cross-host communication. Since the cloud nature of FaaS, each serverless function may be located in a different host even if the functions of the same workflow are close logical related. Some current research re-orchestrates logically related serverless functions into physically close locations to reduce long-distance communication overhead between functions. These studies differ in which logically related functions are selected and in which close physical locations they are rearranged. For example, SAND groups together all the functionality of the same application, which it considers to be logically related. Then reschedule these logically related functions into the same container to achieve the advantages of local communication. In contrast, GlobalFlow reschedules logically related functions into the same cloud zone to avoid low performance by cross-cloud region communication.

Third, replacing the traditional master-side pattern with the worker-side pattern can greatly reduce the overhead of internal interaction in serverless workflows. Currently, there are two emerging worker-side workflow frameworks, Nightcore and FaaSFlow, which offload the overhead of serverless workflow from the master side to the worker side. This new pattern eliminates frequent internal interactions between functions of workflow, so reduced the overhead of serverless workflow scheduling. However, Nightcore and FaaSFlow rely on a gateway embedded in each worker host to schedule each workflow function. We argue these gateways as master-side controllers as well, so they are not like a fully decentralized worker-side pattern.

Chapter 3

Research Related Methodology

This chapter proposes methods to improve the performance of function composition. Since there are gaps of the current research to serverless workflow performance issues, the thesis puts forward its own performance optimization ideas and explains the related reasons in detail. In addition, a new concrete workflow engine BriskChain is designed based on the proposed optimization strategies in order to make the proposed optimization ideas verifiable. The newly designed workflow system is extended from OpenWhisk, so a comparative approach is used to interpret the detail design and implementation of the new system.

3.1 Proposed Optimization Strategies

This section presents ideas and reasons for optimizing serverless workflows. Serverless computing is promising, but faces performance issues, especially in the use of function composition. Based on the literature review (see chapter 2), there are three ways related to the performance optimization of serverless workflow such as the sandbox prewarming, the close-locality strategy and the worker-side pattern. Despite some research progress, gaps still exist. Therefore, this section presents our optimization ideas and reasons to

address the performance issues in serverless computing.

3.1.1 Sandbox Warm-up for Static DAGs

The sandbox warm-up strategy is better suited for static DAG workflows to reduce startup delays for serverless functions. Since the startup latency is a bottleneck in the sandboxes of serverless computing which takes up to 70% of a function's lifetime (Du et al., 2020). As the strategy to mitigate the cold-start issue, the sandbox prewarming is important to be adopted for the optimization purpose. At present, the warm-up research of a single serverless function in most FaaS systems has been done well, but the research on function warm-up in a workflow following predefined workflow path is still lacking. The static DAG workflows can be prewarmed correctly because they are static running routes and are previously known. However, the dynamic DAGs have uncertain route at the run time. Although there are existing strategies to predict final route of the dynamic DAGs according to the frequency of history running path, the missed prediction leads to resource waste and extra overhead. As above reasons, this thesis suggests the prewarming strategy should be used in the static DAGs of serverless workflows, but it is not worth used in dynamic workflows such as branch DAG because of the error prediction.

3.1.2 The Close Locality Strategy for the Workflows

The close locality strategy can be used to improve communication performance of serverless workflows. The reason is that the local communication is simply much more efficient than remote communication. Plus, there are quota restrictions on data movement between functions in a cloud environment (see the section 2.3.2), and local communication can eliminate this limitation. Consequently, it makes sense to use short-range communication between logically related serverless functions.

The close locality strategy schedules functions of a same workflow into the same host. The locality strategy is inspired by previous researches such as SAND and GlobalFlow (see 2.4.2). They group functions of the same workflow into areas that can communicate in close proximity to enable efficient communication between functions. GlobalFlow orchestrates those close functions in the same cloud region to avoid cross-region communication, and SAND arranges those functions in the same container for inner-container communication. However, SAND loses security due to weak isolation for each serverless functions (see 2.4.2). In contrast, this thesis proposes that functions in the same workflow can be grouped together as much as possible into the same host for fast unrestricted local communication. The background reason is that the functions of a same workflow are logically close each other. For example, the input of a function depends on the output of the previous function in a same workflow. In addition, each serverless function should be isolated independently in a container (sandbox), because this containerized isolation is more secure than putting multiple serverless functions in the same container (such as SAND).

3.1.3 The Decentralized Worker-side Pattern

The decentralized worker-side pattern is proposed to reduce communication overhead between the functions of a workflow. Most workflow systems have heavy pressure on the master-side workflow engines. Since the master-side architecture relies on an outside master to server act as a workflow engine, the pattern leads to frequent distrusted communications between the master and workers (see 2.3.3). Additionally, this pattern may rely on a remote distributed database to store the state of each workflow task, which creates more overhead. In contrast, the worker-side pattern offloads the works from one master to many end-to-end workers. There are no internal interaction requirements between the master and workers. Instead, a function of the same workflow

can transport its output directly to its next function in the new worker-side pattern. Moreover, serverless functions are fine-grained functions, and there is a high demand for combining many fine-grained functions in serverless systems. The interaction performance between workflow functions is crucial when dealing with many fine-grained functions. Compared with the master-side mode, the worker-side pattern can greatly omit the intricacies of internal interactions related to the master-side. Based on the above performance benefits, the worker-side pattern in serverless workflow is proposed in this thesis.

Although two proposals about the worker-side pattern emerged, they are not totally decentralized because they still rely on sub-masters in each host (see the section 2.4.3). This thesis presents a novel worker-side pattern which orchestrate end-to-end functions in a workflow without any central controllers (masters). If each function of a workflow has the ability that self-driven its next function as predefined workflow route, then no controller is needed after the first function of the workflow starts. In addition, each function of the workflow only care of its predecessor and successor functions because the input of the current function comes from the output of its predecessor. This inspired us that if the function (sandbox) can be self-driven or self-orchestrated, no controller is needed. Therefore, this thesis creatively proposes a decentralized worker-side pattern that the chained functions of a workflow can run one after another by themselves without relaying any controller. In this way, a workflow can be invoked from the first function of the chained functions, and the workflow result can be got from the last function of the workflow.

Figure 3.1 compares the existing and our newly proposed workflow patterns. Suppose there is a simple sequential workflow (shown in the lower right corner of Figure 3.1) that links tasks in the workflow from task A to task B, and then to task C. The figures demonstrate the processing of the sample sequence workflow using different patterns such as: A) Master-side pattern, B) Worker-side pattern using FaaSFlow (see 2.4.3) and

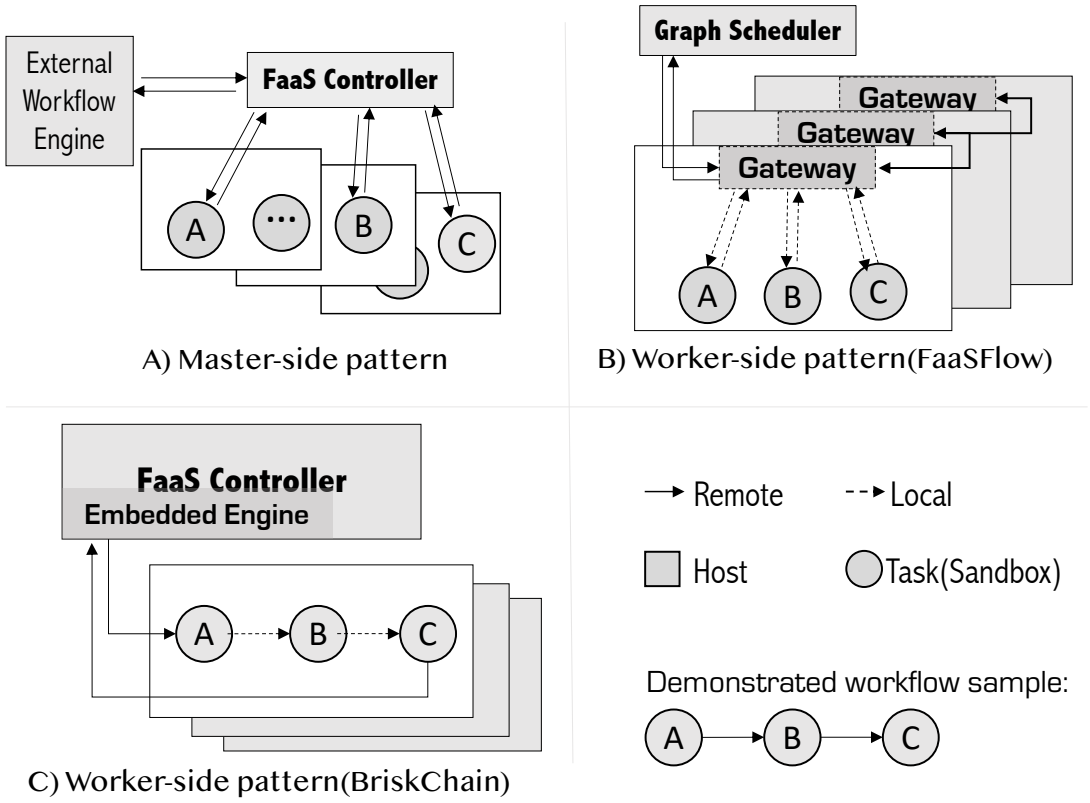

Figure 3.1: The Comparison between Centralized and Decentralized Workflow Patterns

C) The newly proposed worker-side pattern in BriskChain. Lines with arrows represent interactions from one function end to the other function of the sample workflow; solid lines represent cross-host interactions, and dashed lines are local communications.

Figure 3.1 shows that the master-side pattern (upper left corner of the figure) has the most interactions for the workflow tasks from the A to C, and all interactions are TCP/IP connections across hosts. The second pattern in the figure is the newly proposed FaaSFlow worker-side pattern. First, FaaSFlow groups workflow functions into the same host, and then schedules local workflow tasks through its local Gateway which is a decentralized workflow engine embedded in each host. In this way, FaaSFlow will first use local interactions to replace part of remote communication across hosts as much as possible when scheduling workflow tasks. Therefore, FaaSFlow saves a lot of overhead on cross-host interaction compared to the traditional master-side mode. The

third pattern in Figure 3.1 is the worker-side pattern used in BriskChain. The different between BriskChain and FaaSFlow is that BriskChain runtime (sandbox) has the ability to schedule itself to the next sandbox of the same workflow, so BriskChain does not need a controller (master) or even a local Gateway like in FaaSFlow for scheduling. BriskChain uses a clear, simple and minimal interactions to complete the workflow because it only takes two remote and two local interactions to the workflow example. Therefore, BriskChain is a fully decentralized worker-side pattern.

3.2 Proposed Workflow Engine

This thesis proposes a concrete serverless engine (framework) to enable high-performance serverless workflows. The proposed workflows engine is named BriskChain which adopts the optimization ideas mentioned in Section 3.1 and intends to solve the issues found in the first chapter. BriskChain adopts the locality strategy to orchestrate chained serverless functions into a same host. More importantly, it uses our proposed worker-side pattern to orchestrate workflow tasks without any centralized master node.

Figure 3.2 demonstrates the basic structure of BriskChain and related components in the whole FaaS system. The left side of the figure includes key components of FaaS that are used to support serverless functions (sandboxes). The first part of BriskChain is a workflow controller which embedded in FaaS typically in FaaS controller. The main jobs of the embedded workflow controller is to initiate workflow or schedule the sandboxes into the same host. The middle side of the figure is the sandboxes and its running environments. Each serverless function is wrapped in a sandbox, and every sandbox runs on a cluster of multiple physical hosts. A sandbox can schedule itself to the next sandbox in the workflow it is in. On the right side of the figure is BaaS, which includes several backend services that may be used by serverless functions.

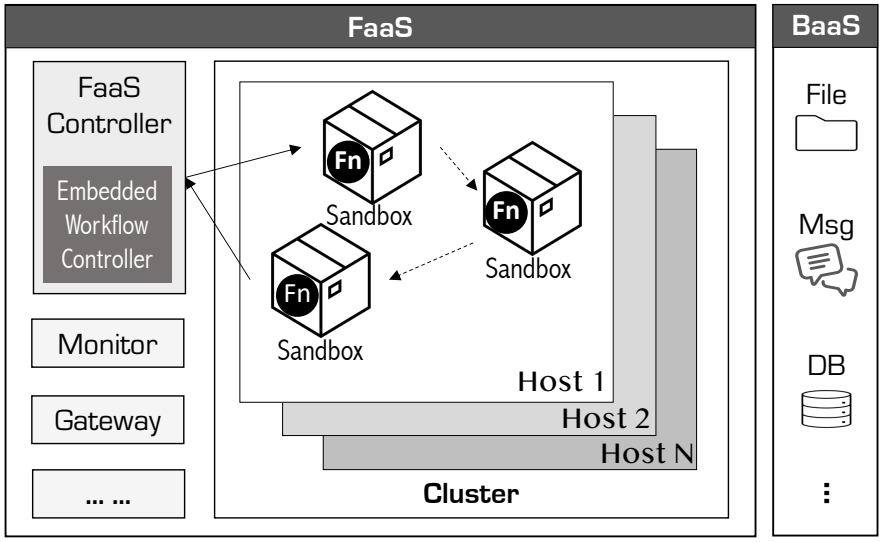

Figure 3.2: Infrastructure of BriskChain and Related Components

BriskChain has two important components, and they are embedded workflow controller and BriskChain runtime. The embedded workflow controller is an internal workflow controller embedded in FaaS controller. BriskChain runtime is a serverless function runtime environment specially for workflow functions. These two components form the core of BriskChain, and they will be introduced in detail below.

3.2.1 Embedded Workflow Controller

BriskChain uses an embedded workflow controller. The different types of serverless workflow engines have been introduced in Section (see 2.2): the engines can be embedded in a FaaS system or work with FaaS as an external standalone system. Most workflow systems use an external engine to schedule their workflow tasks, and the engine server resides outside the main system as an add-on. In contrast, this thesis proposes an embedded workflow controller integrated in a FaaS system for the following reasons.

The traditional external workflow engine approach used in FaaS has several disadvantages. Firstly, both FaaS and workflow engines have their own controller, leading to

redundancy in the role of serverless function orchestration. The workflow engine acts as a central controller for orchestrating workflow tasks (functions), and FaaS system typically also rely on a controller to manage serverless functions or handle oncoming requests. Both FaaS controllers and workflow engines (controllers) act on serverless functions, adding to the role of redundant scheduling. Secondly, the mechanism of external workflow engine leads the restriction for unified user cases. There is a substitution principle proposed by Baldini et al. (2017), and the the main meaning of the principle is that a workflow (composed-function) can itself be used as a serverless function. The substitution feature is useful for integration between workflow and functions in complex applications, so tasks of independent functions and workflows can interact with each other as if they were in the same system. However, external workflow engines cannot follow the substitution principle because workflow and serverless functions are controlled by two different controllers. For example, a workflow task made by Amazon Step Functions cannot be invoked like an AWS Lambda function because the two tasks are different things managed by two different systems (López et al., 2018).

In contrast, if the workflow engine (controller) is merged into FaaS controller becomes one unified controller, then each workflow task and single serverless function task can be treated equally in the system (see 3.4.8). The substitution principle is also easy to implement due to the unified controller, and the principle means that a workflow is treated as an ordinary serverless function. For example, serverless workflows have all the attributes and capabilities of normal serverless functions, such as default parameters, limits, blocking invocation, web export. Therefore, the workflow engine is integrated into the FaaS system in BriskChain, which unifies the workflow controller and the FaaS controller together.

3.2.2 BriskChain Runtime (Sandbox)

A sandbox is a runtime environment for serverless functions, also known as the BriskChain runtime.[1] BriskChain uses containerized isolation for each serverless function, which means that each function is wrapped in a containerized sandbox. The BriskChain runtime expands the original sandbox in FaaS system and enhances the ability to schedule workflow functions. Therefore, BriskChain not only provides a runtime environment for serverless functions like a common FaaS sandbox, but also has the ability to schedule workflow tasks.

The BriskChain runtime is the key to implemente our newly proposed worker-side pattern. Inside each BriskChain sandbox (container) has an embedded micro-engine that schedules workflow from current node to the next. Thus, each sandbox can schedule itself to the next task of the workflow without the support of a central controller. Consequently, BriskChain is a complete worker-side pattern, where each worker (sandbox) can schedule itself without relying on a master or sub-master like other workflow systems.

3.3 OpenWhisk

The chapter uses a comparative approach to present the design and implementation of BriskChain. Before involving detail BriskChain engineering, an original design of common open-source FaaS system is first analyzed, and then BriskChain can be comparatively introduced. OpenWhisk was chosen as an example to demonstrate the main principles of the FaaS system, but these optimization ideas are common in most open-source FaaS system. In addition, due to the many limitations of closed-source FaaS systems, some open-source FaaS systems such as OpenWhisk[2] help us understand

[1]The BriskChain runtime and the sandbox are interchangeable terms in the thesis.
[2]OpenWhisk is open-source, but it also has a commercial edition called IBM Cloud Functions.

the secrets inside the FaaS system to facilitate our research.

3.3.1 The Common Sandbox Structure

The sandbox encapsulates serverless functions, and isolates the functions in serverless applications. It is commonly a containerized isolation in many FaaS systems.[3] Virtualized containers can provide an autonomous runtime for serverless functions, such as some system libraries or program dependencies. One serverless function is usually encapsulated into one sandbox; there can be multiple functions in a sandbox, but this design is rare in existing FaaS systems because of the low security of functional isolation. A serverless application is made by many functions, yet each function must be wrapped into a sandbox because a bare serverless function code cannot executable without context or running environment. In summary, each serverless function needs be encapsulated into a sandbox as a basic unit to form a complex serverless application.

Figure 3.3 shows a basic structure of a sandbox (container). The bottom layer of the figure indicate an operating system (OS) level environments, which prepare a suitable system environment for its up layer. Since containerized sandboxes such as Docker share the kernel with their host system, there are only underlying library dependencies in the structure figure and no kernel. The second layer of the sandbox is the FaaS runtime, which is the core runtime environment for serverless function and contains programming language environment for running code in different programming languages. For example, the FaaS runtime relies on distinct programming language libraries to run the function source code with different programming languages, so serverless application supports multiple programming languages such as Java, JavaScript, Python and Go. The top layer of the sandbox structure is the function source code, which is written by FaaS application developers to complete their application logic.

[3] See the table 2.1 for different virtualized isolation such as Unikernel or FireCracker and so on.

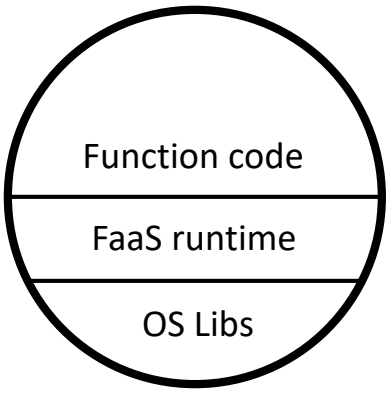

Figure 3.3: Sandbox Structure

3.3.2 The Sandbox Life Cycle

A sandbox life cycle begins when an image is initialized as a sandbox (container) instance until the container instance is destroyed. Initially, the sandbox is an image file (such as Docker image file) in FaaS libraries. A runnable sandbox can be processed by several steps showing at Figure 3.4 when after a sandbox instance loaded (started) from a docker image. At the first step, a loaded sandbox is a runnable environment of a function including system libraries and FaaS runtime. In the second step, source code of the serverless function will be injected into the loaded sandbox. Specifically, the function code is designed at the application design time by the developers and is injected into the sandbox at the runtime by the FaaS controller. Up to this point, at step 3, the sandbox has been capable of processing the function request and providing the result of the function service in accordance with the requested parameters. A ready sandbox can continue responding the service as long as it is alive. The life of the sandbox is ended when a ready sandbox is destroyed. For example, it can be destroyed by the controller for an idle situation to save the resource consumption.

However, a sandbox can also be stored in a pool for later use, thus avoiding cold starts. Studies have estimated that a cold start can take up to 70% of a function's lifetime (Du et al., 2020), and Shahrad et al. (2019) reports that initializing a function with its

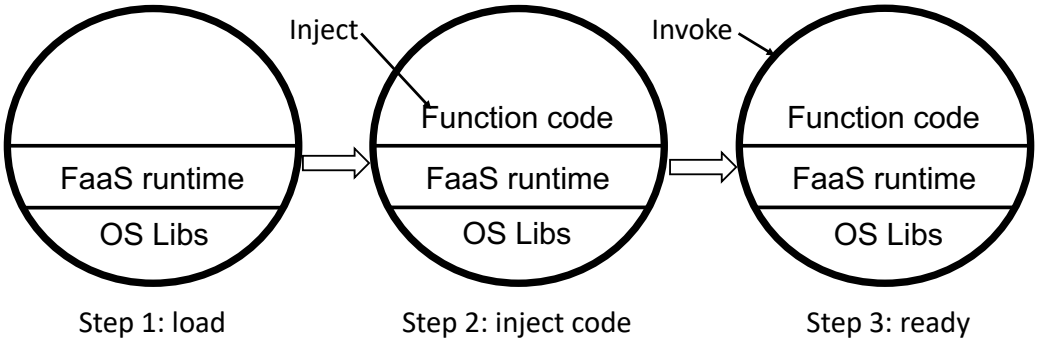

Figure 3.4: The Process Steps of a Sandbox

related resources requires at least 500ms. Since most open-source FaaS systems employ containerized isolation, the time it takes to start the sandbox from a Docker image significantly contributes to the cold start latency issue. In OpenWhisk, the additional tasks required by the system before executing the serverless function fall under the category of cold-start work. These tasks include loading runtime libraries and injecting function code when launching a new sandbox instance. Due to concerns about cold starts, it is beneficial to keep ready sandboxes alive. Therefore, most FaaS systems employ strategies to cache live sandboxes in memory, saving them in a pool for later reuse. Additionally, sandbox caches may have different levels. For example, instances from step 1 or step 2 in Figure 3.4 can be cached in the pool for future use.

3.3.3 OpenWhisk Architecture and Components

Figure 3.5 shows the internal structure of OpenWhisk and its processing flow. NGINX is a gateway service using at OpenWhisk. It handles outside request from the users, and expose inner function-services to the outside of FaaS system. CouchDB provides persistent storage for running FaaS system. The persistent data include such the functions' source code and the function running states. The controller is like a central manager of the whole FaaS system that manages and schedules the serverless functions. When the controller is received a function task from NGINX, it will fetch the function

code from CouchDB and inject the code into a sandbox. Then, the controller invoke the sandbox and get the function result. Kafka is like a message bus responsible for sending messages or commands to the sandboxes (functions). The bottom side of the figure demonstrated several sandboxes represented by circles. The sandbox can run on any host in the cloud, because OpenWhisk just requests resources from the cloud to run sandboxes, and does not care where to run them.

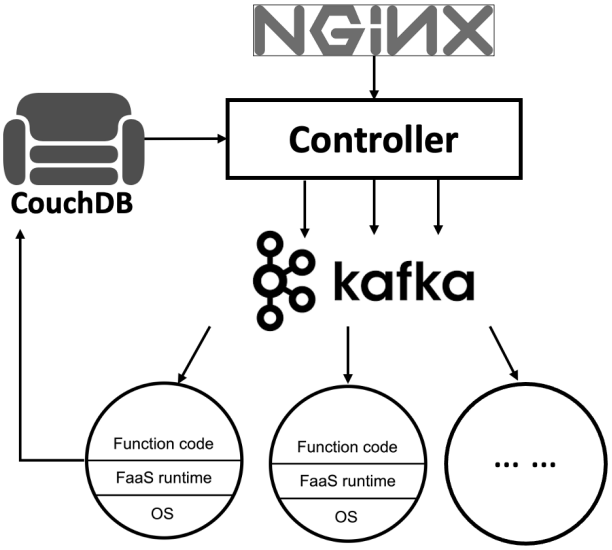

Figure 3.5: The Internal Structure of OpenWhisk

3.3.4 Processing of Serverless Functions in OpenWhisk

There are design time and run time of a FaaS application. The time when developers build their serverless applications is called the design time. The developers write the function code and deploy them into FaaS platform at the design time. Specifically, the function source code will be saved into CouchDB when after deployed the functions in OpenWhisk system. The runtime handles the users' requests, processes them in the FaaS system, and returns the results to the users. For example, the tasks of the runtime include such as executing a serverless function, orchestrate serverless functions, or prepare the runnable sandboxes.

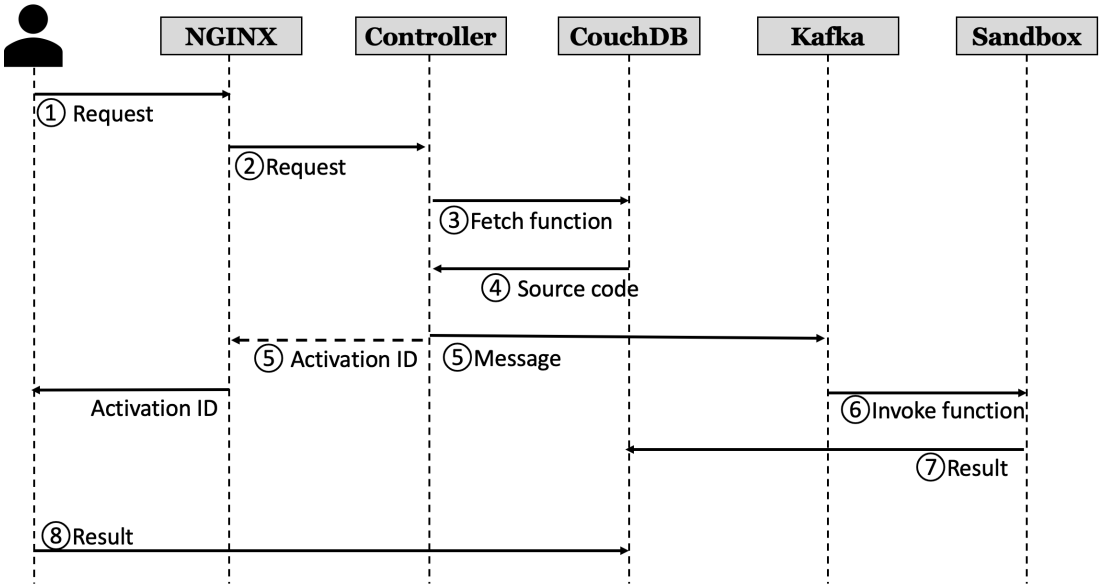

Figure 3.6: OpenWhisk Sequence Diagram

Figure 3.6 shows a sequence diagram of OpenWhisk at runtime. It demonstrates a request of the serverless function from a user, then the serverless function responses results to the user. ① The whole process start from the user initiating a serverless function request to the gateway (NGINX), which includes function parameters and identities. In addition, the FaaS system will verify the legitimacy and security of the user and the requested resource after receiving the service request, but these parts are ignored in the figure for the convenience of demonstration. ② NGINX will forward the request to the controller because the controller is a real manager to the function services. ③ ④ The controller obtains the program source code of the function from CouchDB through the function identity, which the code is stored in CouchDB at the design time. The function code will be injected into a loaded sandbox, which will further become a runnable sandbox. ⑤ Then, the controller sends an invocation message to Kafka. Meanwhile, the controller will asynchronously feed back a token (an activation identity) to the user because the function may run for a while and the user can request the result later with the token. ⑥ Kafka has received the innovation message (command), but it is only a message bus which do not process the request indeed. Kafka will transfer the

request to the sandbox to invoke the function. ⑦The function request will be processed in the sandbox and the result will be stored to CouchDB. ⑧ Finally, the user can fetch the function result from CouchDB.

3.4 BriskChain

BriskChain is a framework that supports high-performance function composition (workflow). It follows the optimization strategy and program framework ideas proposed in the previous sections. The BriskChain system is extended based on open-source FaaS systems with two extension points: the embedded workflow controller and the BriskChain runtime sandbox.

3.4.1 The Workflow Logic Definition

Workflow logic is defined as a script file that instructs the run path and logic of each function in the workflow. The workflow logic script is written by the developers at the design time, and the workflow program will be injected into the BriskChain sandbox at runtime. The workflow program definition follows the form of a directed acyclic graph (DAG) consisting of vertices and edges. The vertices here can be understood as serverless functions, and the edges are the running paths between the functions in the workflow. Also, a workflow has only one entry vertex and one output vertex, without any loops in the workflow graph. In general, a workflow's definition file contains references to serverless functions and run paths between the functions.

3.4.2 BriskChain Sandbox Structure

The BriskChain sandbox is extended from the common FaaS sandbox with additional capabilities for handling workflow functions. Compared to the normal sandbox, the

BriskChain runtime is set on top of the FaaS runtime, as shown in the shaded grey area in Figure 3.7. The BriskChain runtime is a new runtime environment of the serverless functions that handles both a single standalone serverless function and a serverless function in a workflow. For example, if the function in the sandbox is a normal serverless function (not a function of the workflow), the BriskChain runtime will forward the task to its next layer of FaaS runtime for processing. This ensures that the processing logic of the original serverless function remains the same. In contrast, if the function in sandbox is a serverless function of a workflow, the BriskChain runtime will process the function according to workflow definition. The BriskChain runtime understands how to schedule the result of the current serverless function according to the workflow definition, which will be explained in detail at the next section.

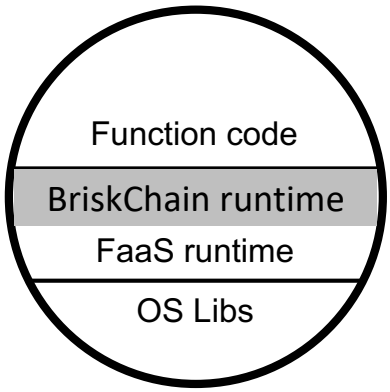

Figure 3.7: BriskChain Runtime (Sandbox)

3.4.3 BriskChain Architecture

Figure 3.8 illustrates the fundamental components of BriskChain integrated into the OpenWhisk system. The BriskChain framework extends two components of the common open-source FaaS system. One is the BriskChain runtime located within the sandbox, as introduced in the section 3.4.2. The other component of BriskChain is embedded in the FaaS controller. These two extended parts are represented in grey

within the new system, based on the original OpenWhisk structure shown in Figure 3.5.

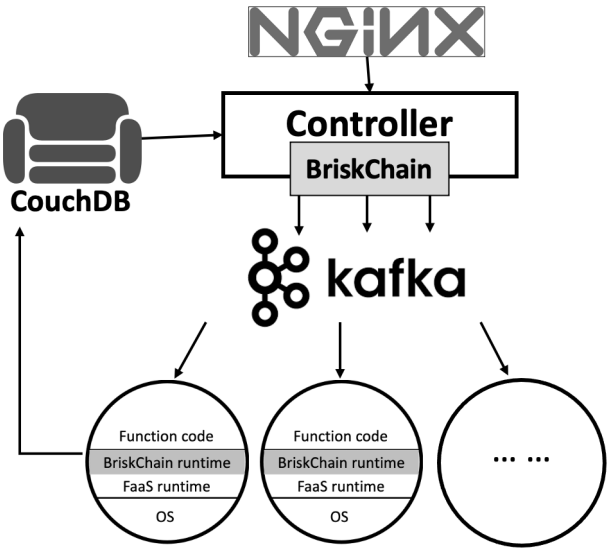

Figure 3.8: Components of BriskChain Integrated in OpenWhisk System

Figure 3.9 gives another view of BriskChain in the architecture of FaaS system. The left side of the figure shows the main components of FaaS that support the serverless functions. For example, the controller can invoke, manage and schedule each serverless function; the gateway expose service of the functions to the outside of FaaS system; the database stores the source code and request response status of serverless functions. In the middle of the figure indicates serverless functions and its running environment. The most open-source FaaS systems rely on Kubernetes to orchestrate containers (sandboxes). A 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, and each sandbox runs in a cluster. The right side of Figure 3.9 shows different BaaS systems. For example, if a serverless function has persistent storage requirements, it can use a BaaS service such as AWS S3 to hold its business data.

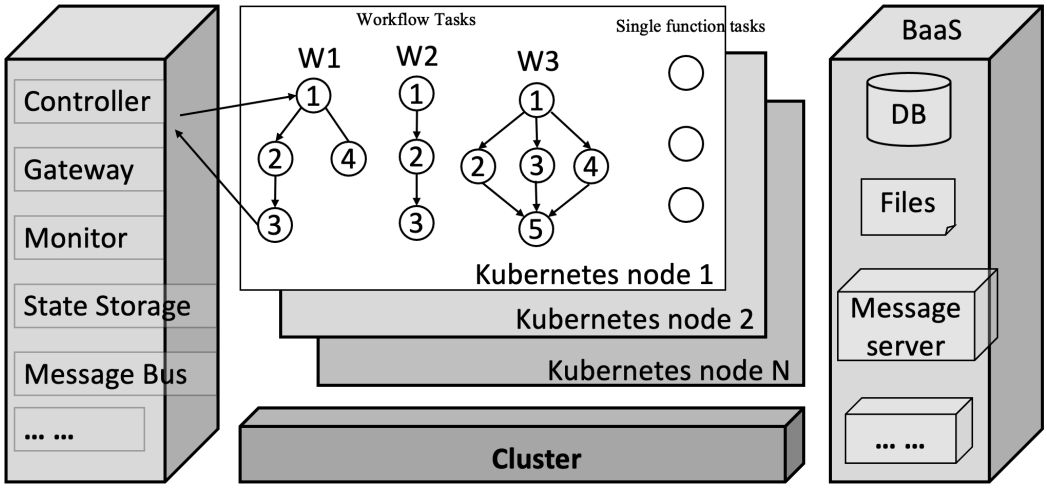

Figure 3.9: The Architecture of BriskChain and FaaS

3.4.4 The Substitution Principle

BriskChain abides the substitution principle that a composed-function workflow itself can be a serverless function. BriskChain extends the FaaS controller letting it have the ability to invoke workflow tasks. For example, in the case of the workflow W1 at Figure 3.9, the controller invoke workflow W1 based on the calling of vertex 1 in W1. If the branch vertex 1 comes to the vertex 2 in the runtime, then the result of vertex 3 is the workflow result which is responded to the controller. The enhanced controller treats a workflow task same as a normal serverless function task, which also means a workflow is just one task for the controller. Whatever a single serverless function or a group functions in a workflow, there are no difference managed by the controller. In this way, a complex FaaS application can smoothly be integrated many workflows and serverless functions together.

3.4.5 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 offloads work from the master side to the worker side, so the workers (sandboxes) are the core of this pattern. 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 current function to the next function according to the predefined workflow schema; It also forwards the result of the current function to its next function in a workflow directly without feed 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 BriskChain runtime using simple

pseudocode pieces.[4] 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. In line 5 of the algorithm, if the schema parameter (workflow definition) is not passed into the current sandbox, it means that the current function is a regular single serverless function or the last function of the workflow. Therefore, the BriskChain runtime forwards the function to the FaaS runtime (the layer below the BriskChain runtime) for processing, and then responds with the function result. Otherwise, the BriskChain runtime will request new sandbox resources for the next step of the workflow, and then asynchronously transfer the results of the current sandbox to the new sandbox. There are three cases to the next step of a workflow: sequence, parallel and branch. In the case of sequences, on line 7 of the algorithm, the current function result is sent to the sandbox of the next function in the workflow as the input parameter of the new function; if the current workflow node fan-out many parallel sub-functions, then BriskChain runtime will asynchronously dispatch the current function results to multiple sub-functions; if the current workflow node is a branch, BriskChain runtime determines its subsequent route according to predefined conditional rules.

3.4.6 The Locality Strategy in BriskChain

The extended controller has the ability to support function locality strategy. The controller orchestrate all the sandboxes of the same workflow into the same host, which the host is specially a Kubernetes node in our demonstration. Therefore, BriskChain supports that each function in the same workflow can interact with each other like local communication. For example, W1 is the first workflow in Figure 3.9; the functions 1, 2, 3 and 4 are located in the same Kubernetes node 1, so these four functions can efficiently communicate locally.

[4]The pseudocode is not a real code implementation, just a convenient way to describe BriskChain's design logic.

3.4.7 The Prewarming Strategy in BriskChain

The BriskChain controller supports the prewarming strategy for serverless workflows. Open-source FaaS systems such as OpenWhisk have their own prewarming strategies, but the prewarming is only for a single serverless function. For example, the controller can pre-warm sandboxes and cache them in a pool based on how often each sandbox is used. BriskChain will not change the original prewarming capabilities of the FaaS system. Nevertheless, Brisk Chain adds prewarming ability for the workflows based on its predefined running route. The sandboxes of a same workflow will be warmed previously by the controller before the workflow run except encounter a branch node in a workflow. For example, all the sandboxes can be warmed up in advanced for the workflows W2 and W3 in Figure 3.9 because these two workflows do not contain any branch vertices. In a case of workflow including branch, the FaaS controller will continue to warm-up the subsequent sandboxes when after a branch vertex in a workflow. Specificity, when a branch vertex (sandbox) requests the running resource for the next sandbox from the FaaS controller, the controller will warm up the subsequent sandboxes from the current vertex of the workflow.

3.4.8 Processing of Serverless Function in BriskChain

Figure 3.10 shows the workflow processing sequence diagram using the BriskChain framework integrated in OpenWhisk. The figure explains relevant runtime process steps after workflow performance optimization, and it will be better understood if compared with Figure 3.6 before performance optimization. Sine the current sequence diagram 3.10 is more complex than the original sequence diagram of OpenWhisk (Figure 3.6), the processing explanation is divided into three parts in the following three paragraphs.

BriskChain treats a workflow as a unit task in the FaaS runtime, so the way a user

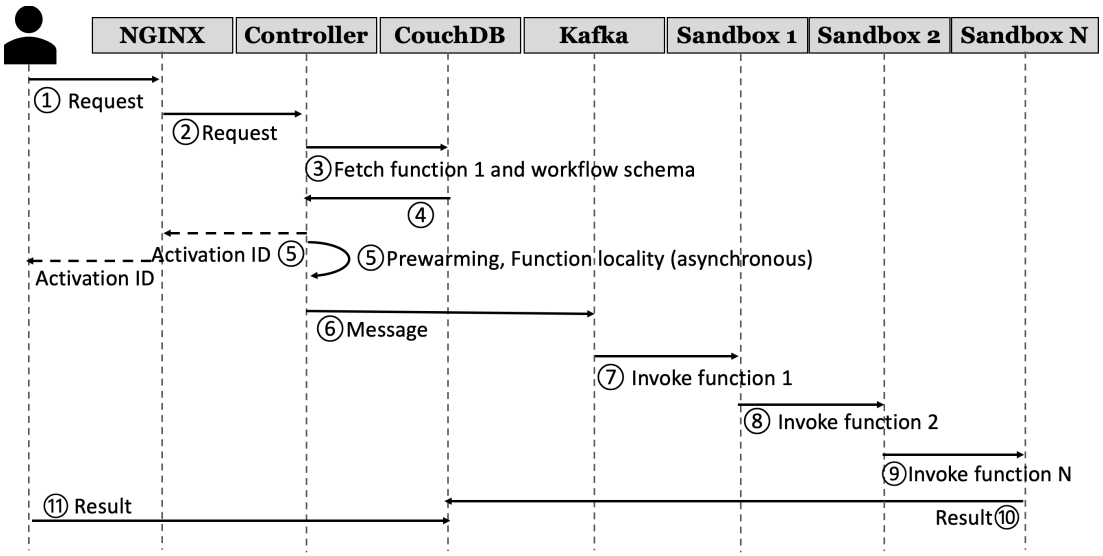

Figure 3.10: BriskChain Sequence Diagram in OpenWhisk

invokes a workflow is the same as a single function in FaaS. The original OpenWhisk sequence had been introduced at the diagram 3.6, which a user sends a serverless function task to OpenWhisk and obtain the result after the system is finished the task. The sequence is similarly to BriskChain that a user requests a workflow task and obtain the result even though the workflow task is composed by multiple function tasks. Figure 3.10 shows a new sequence diagram with BriskChain framework. ① At the runtime, an end user requests a workflow task to the gateway (NGINX). Specifically, the user invokes the first function of a workflow with a predefined workflow schema[5] as a calling parameter. Hence, BriskChain knows how to schedule the following functions of the workflow according the workflow logic definition. ② NGINX forwards the requirement to the controller with the function parameters, function ID and the workflow schema ID. ③ ④ The controller uses the function and workflow ID to get the function source code and detailed workflow definition from CouchDB. Then, the user will receive a token (activation ID) at the step ⑤ when after the controller correctly accepts the workflow task. At the end of processing, the result of the workflow comes

[5]The workflow logic definition

from the result of the last function of the workflow. For example, the step ⑩ returns the workflow result from the last function N and stores the result into CouchDB. ⑪Once the workflow result is stored in CouchDB, the user can use the token (activation ID) to get the workflow result. From the above sequence diagram, it can be observed that BriskChain handles workflow tasks as a unit as a whole, and this workflow unit is no different from a single functional task for FaaS controller scheduling. Also, there is no operational difference between a user invoking a workflow task and a single functional task.

The controller asynchronously warms up the sandboxes and schedules the locations of the functions at the beginning of a workflow run. At the step ④ of the figure 3.10, the controller obtained the first function of the workflow and the workflow schema. At the step 5, the controller warm up the sandboxes according to the order of the chained functions in the workflow schema.[6] Meanwhile, the controller schedule all the ready sandboxes of the workflow into a same Kubernetes node (same physical host). The prewarming and the locality works to the sandboxes of a workflow are processed asynchronously.

The functions of a workflow run one after another without repeated contact with the central controller. It can be seen that the controller sends a workflow task to Kafa at the step ⑥ of Figure 3.10 which is same to the step ⑤ of the original OpenWhisk sequence diagram in Figure 3.6. Then, there are many sandboxes from the sandbox 1 to the sandbox N in Figure 3.10 to run N functions of a workflow. These sandboxes are located in a same physical host (in a same Kubernetes node), so they are communicated locally. Furthermore, the chained functions are invoked one after another in the steps ⑦, ⑧, and ⑨ of Figure 3.10. For example, in the figure, the result of the function 1 (the step ⑦) is sent to the function 2 (the step ⑧) as the function input. Finally, the result of

[6]Some sandboxes may already be warmed by the original strategy of FaaS previously (not by BriskChain), then BriskChain can reuse the sandboxes without warm up again.

the end function N is the workflow result, which will be saved into CouchDB (at the step ⑩) for the end user to query.

3.5 Summary

This chapter looked at our method to improve the performance of serverless workflows. Three optimization strategies are proposed to address performance issues of serverless workflow such as cold start, cross-host communication and low performance interaction pattern. First, a warm-up strategy can solve the cold-start performance problem. While this strategy is already common in FaaS, it is just to warm up running environment based on how often a single function is used. The thesis proposes that the running environment of the functions in the workflow should also be warmed according to the running path of serverless workflows. Specifically, the sandboxes of the workflow should be warmed up or cached in a pool according to predefined workflow logic before the serverless run. Second, close locality strategy can solve the performance issue caused by long cross-host communication. This strategy lets the workflow engine schedule the function tasks of a same workflow located as in a same host. Thereby, the state data between the end-to-end functions of a workflow can be fast transferred. Third, the traditional master-side workflow engine pattern is abandoned to avoid the frequent communication requirements. A new decentralized worker-side workflow pattern is proposed in the thesis for the workflow performance optimization.

The thesis proposes a decentralized work-side pattern which can significantly reduce communication overhead in serverless workflow. The pattern is decentralized without a central master in the workflow, for the central master offloads its scheduling workload to each worker side. In the worker-side pattern, each worker node has the ability to route state from itself to the next worker node according to a predefined workflow schema. Here is an example of workflow processing in work-side pattern: the workflow

is initialized by a serverless controller,[7] and calls the first function of the workflow; after that, the workflow can route itself until the end function task of the workflow is completed; finally, the output of the last function in the workflow will be the result of the entire workflow. Our research proposes the work-side pattern because it can improve workflow performance based on the following advantages. Each worker task can transmit its state data directly to the next worker in the workflow, rather than feeding it back to the master side again. This approach eliminates the TCP/IP connections between the master and worker, thus significantly reducing the overhead of the workflows. At the same time, the pressure on the master side is released, and the scheduling overhead is distributed to each worker.

Based on the optimization strategies proposed in this thesis, a decentralized workflow framework BriskChain for latency-sensitive and interactive serverless computing is further presented. BriksChain extends an open-source FaaS system called OpenWhisk with two extensions, the BriskChain runtime and the embedded controller, to optimize serverless workflow performance. The BriskChain runtime is a containerized sandbox which extended common serverless function runtime. The difference between common serverless function runtime and the BriskChain runtime is that the BriskChain runtime can handle not only normal serverless function requests, but also the functions in the workflow. Moreover, BriskChain runtime has the ability to schedule itself from the current node of a workflow to the next node of the workflow without relying on a central controller (master). Therefore, BriskChain is a decentralized worker-side pattern that offloads scheduling tasks from the master to the worker. The second major component of BriskChain is the embedded controller. It is not responsible for task scheduling in the workflow like the controller (master) of other workflow engines because the scheduling tasks have been distributed to each functional sandbox. Instead, it is responsible for

[7] It is the controller of FaaS but not the workflow controller or engine

allocating workflow resources and some auxiliary tasks, such as implementing close-locality and warm-up performance optimization strategies. The embedded controller, as part of the FaaS controller, has the ability to schedule sandbox locations and warm up the sandbox. Specifically, it schedules sandboxes (functions) of the same workflow into the same host to avoid remote interactions between workflow functions; it also warms up sandboxes used in workflows based on predefined static workflow logic.

In addition, BriskChain schedule serverless workflow as a whole unit same as to schedule a single serverless function. This means that serverless workflows are also be used same as serverless functions, so abides the substitution principle (Baldini et al., 2017). Back to the general workflow system, they use an external workflow engine as a master to control each workflow task, which makes the substitution principle invalid. The reason is that workflow tasks and serverless function tasks are different types of tasks managed in two systems (FaaS system and workflow system), so the two kinds of tasks cannot be interchanged or work together. In contrast, BriskChain eliminates the external workflow master and offloads scheduling tasks from the master to each worker (workflow function); the BriskChain embedded controller also is a part of FaaS system. For the newly designed workflow framework BriskChain, there is no difference between serverless function tasks and workflow tasks for the scheduling work of the controller. Therefore, BriskChian regards the functions in a workflow as a whole, and schedules the entire workflow just like a normal serverless function.

Chapter 4

Evaluation and Analysis

This chapter presents the evaluation and analysis of the proposed worker-side pattern with other co-designed optimization strategies in the thesis. Several experiments will be conducted to assess the performance of function composition (serverless workflow), such as scheduling overhead of the workflows. A workflow framework called BriskChain was introduced in the previous chapters, which implements the optimization strategies proposed in the thesis. This chapter assesses BriskChain's performance and compares it with other workflow systems such as Apache OpenWhisk (Apache Software Foundation, 2022a) and Apache OpenWhisk Composer (Apache Software Foundation, 2022b). Various cases were applied to this evaluation, including synthetic workflows and real-world applications. Finally, the experimental results are collected, summarized and analysed in detail.

The evaluation seeks to answer the following questions:

(i) What is the scheduling overhead of BriskChain compared with others in different DAG forms? (§4.2)

(ii) What is the performance of BriskChain scheduling serverless functions based on the different payloads of the workflows? (§4.3)

(iii) How does BriskChain perform in real-world applications? (§4.4)

4.1 Methodologies of Evaluation

4.1.1 Compared and Evaluated FaaS Systems

The performance comparison is based on BriskChain, OpenWhisk, and Apache OpenWhisk Composer (also called Composer or Apache Composer). BriskChain is a workflow framework that implements the performance optimization strategies proposed in the thesis. It currently supports DAG forms such as sequence, branch and parallel. OpenWhisk is an open-source FaaS platform which provides running environments of serverless functions. It also supports sequential function composition, but not more forms such as branching and parallelism. Therefore, the evaluation involves Apache Composer when the benchmarks include branch or parallel cases. Apache Composer is a new programming model for composing cloud functions built on OpenWhisk, and it supports branch, parallel and other forms of function compositions.

Apache OpenWhisk Composer, BriskChain, and OpenWhisk have similar runtime environments of serverless functions for a fair evaluation. Composer is based on OpenWhisk, which schedules the serverless functions relying on the original OpenWhisk's runtime (container). Likewise, BriskChain extends OpenWhisk's function runtime, and schedules tasks based on the extended runtime. The compared FaaS systems have fairly similar underlying runtime environments, yet they also include different aspects of interest for evaluation, such as workflow scheduling or internal interaction. Therefore, there will be a less unfair evaluation of the comparison methods.

4.1.2 Benchmarks and Experimental Methods

The experiments are split into two categories: (1) Synthetic Micro Workflows and (2) Real World Application Benchmarks. The synthetic benchmarks are further subdivided into three individual sections including sequence, branch or parallel DAG forms. For

example, the sequence benchmark consists only of sequence DAG forms that chain together identical functions of different lengths. Likewise, the branch benchmarks synthesize branching nodes in the workflows. The parallel benchmarks fan out a task to a varying number of parallel subtasks. When the parallel tasks are complete, they are fanned into one output result. Lastly, two representative real-world applications are built. They are meaningful use cases with payloads and external dependencies and involve all three DAG forms.

The synthetic micro workflows and the real-world applications cover different aspects of our evaluation. First, the synthetic workflows are evaluated for each DAG form individually. At the experiments, different size of payloads are set, and the impact caused by any third-party APIs are eliminated. The specific interest can be evaluated without being affected by any unrelated parameters. Second, in contrast, real-world applications are difficult to evaluate to a point of specific details. However, real-world applications can be convenient to examine the overall effects of workflow optimization.

4.1.3 Parameters of Interest

The following list shows different parameters that are affected by the experimental workflows.

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

(b) **Variability and 95th latency:** The 95% 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.

4.1.4 Metrics of Goodness

Overhead is a key metric to measure the quality of BriskChain workflow scheduling. The runtime overhead of the workflows will be measured based on the different parameters of interest (4.1.3). The overhead is related to the runtime of the workflow. In our benchmarks, unless specifically stated beforehand, the overhead mostly measures all time spent outside the functions in a serverless workflow. It typically includes scheduling time, state transition time, and delay time between the workflow functions. The overhead result is calculated by subtracting the time spent by each function in the workflow from the total runtime of the workflow. Suppose $t(s)$ is the total execution time of a workflow, and $t(f_i)$ is the execution time to a specific function of the workflow. Therefore, the overhead of the workflow is calculated as the following formula.

$$Overhead = t(w) - \sum_{i=1}^{n} t(f_i)$$

4.1.5 The Experimental Environment

Table 4.1 shows detailed hardware and software setup for the environments. A Kubernetes cluster was built on Google Kubernetes Engine (GKE) with three host nodes installed in the same cloud zone. Then, OpenWhisk, Apache Composer and BriskChain are installed in the Kubernetes environments.

	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 4.1: Hardware and Software Setups in the Experiment

4.2 Scheduling Overhead Evaluation and Analysis

The overhead will be assessed in detail separately through different DAG forms in this section. Three independent synthetic micro-benchmarks are conducted, and there are sequence, branch and parallel shown in Figure 4.1. They will be evaluated for the scheduling overhead in different numbers of nodes in the workflow benchmarks.

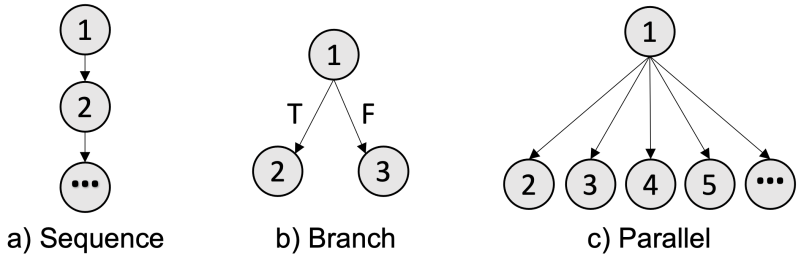

Figure 4.1: Synthetic Benchmarks

4.2.1 The Sequence DAGs

This subsection evaluates the runtime overhead of sequential DAG form using Brisk-Chain. The same functions are used in the workflows and chain these functions together in sequential form. These functions are empty and respond to calls immediately. Also, there is no payload[1] transferred between the internal functions of the workflow. The functions will encounter a long cold start delay when they are run for the first time. The results of this experiment do not collect the time-consuming results of the first cold start for the time being. Additionally, this test does not run workflows concurrently to avoid performance degradation due to resource contention, and each workflow case is executed 100 times to avoid randomness. Lastly, the experiments are conducted at the Kubernetes cluster with three machine nodes.

Figure 4.2 shows the overhead of the synthetic sequential workflows. In the figure,

[1]To facilitate data analysis, only a short string is transmitted to record the execution status of the workflow.

BC-average, OW-average, BC 95%-ile latency and OW 95%-ile latency represent the experimental results of the average overhead in BriskChain (BC), OpenWhisk (OW), respectively, and their corresponding 95% percentile tail latency (95%-ile). The experiment compares the overhead of workflows running in BriskChain and OpenWhisk, where OpenWhisk specifically refers to using OpenWhisk sequence action[2].

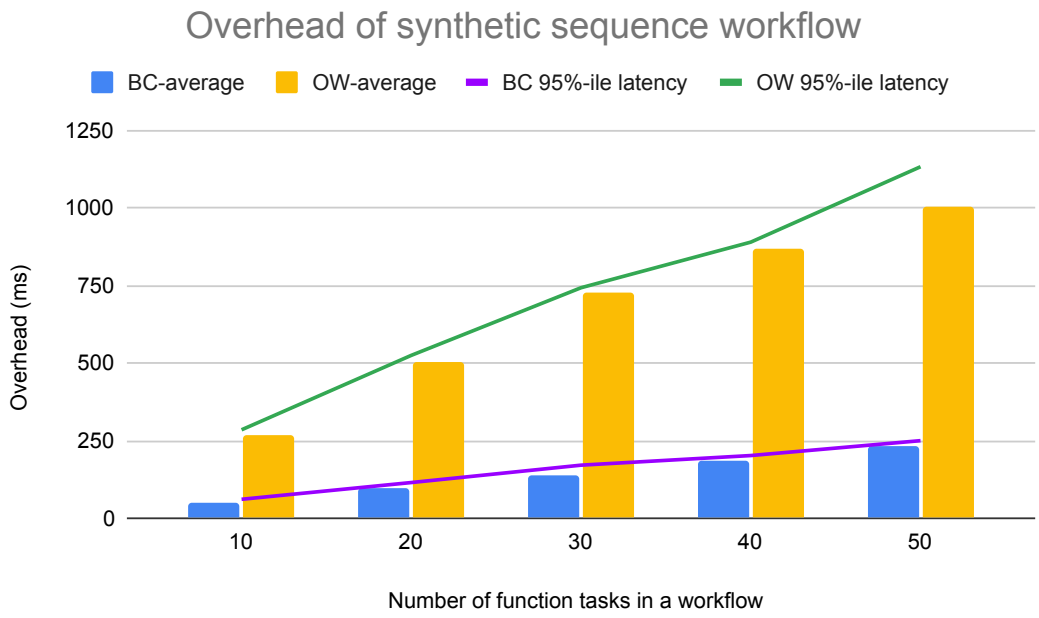

Figure 4.2: Overhead of Synthesize Sequential Workflows

Overall, BriskChain has higher performance than OpenWhisk in running sequential workflows, and their overhead grows linearly with the number of functions in Figure 4.2. Firstly, it can be found that BriskChain has at least 70% less overhead of the average value than OpenWhisk. For example, in a workflow consisting of 50 sequential functions, BriskChain takes about 239 milliseconds of time overhead, which is about 75% less than running the same workflow with OpenWhisk. Secondly, the corresponding 95% percentile latency of the overhead are little higher than their average data in both of BriskChain and OpenWhisk. Thirdly, the overhead of the sequence workflows grows

[2]OpenWhisk sequence action is used to compose sequential serverless functions, which is an inherent feature in OpenWhisk.

linearly with the number of functions running in both BriskChain and OpenWhisk.

Analysis ①: The overhead of BriskChain is lower than OpenWhisk (as shown in as in Figure 4.2) because of a worker-side pattern in BriskChain runtime[3]. BriskChain initiates the first function of the workflow through the FaaS controller. When the function completes, its result is dispatched directly by the BriskChain as an input parameter to the next function of the workflow. From then on, subsequent functions of workflow scheduling no longer need to feed back function status to the controller. This is the worker-side pattern, where BriskChain does not interact with the controller frequently to reduce the overhead of internal interactions in workflows. In contrast, OpenWhisk uses a master-side pattern, which involves a lot of scheduling overhead between the controller (master) and the function tasks (workers) of the workflow. The workflow process in OpenWhisk starts with the controller accessing a function task, and then the function response with the result to the controller. After that, the controller schedules the next task of the workflow again in the same process until all the functions of the workflow are completed. As a result, OpenWhisk (OpenWhisk sequence action) involves multiple internal interaction loops between the master and the workers when processing the workflow, resulting in an expensive overhead.

Analysis ②: Another reason why BriskChain's overhead is lower than OpenWhisk sequences is that BriskChain omits many unnecessary remote database accesses that can cause workflow delays. 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. Specifically, the number of remote database accesses is massively accumulated based on the length of the workflow. As a result, these large cumulative database accesses can cause time delays in the sequential workflows that OpenWhisk schedules. However, we believe that the internal data of the workflow does not have

[3]The BriskChain runtime, also known as the BriskChain sandbox, wraps a serverless function into a container and provides a running environment for the function.

to use time-consuming persistent storage, except for special purposes such as logging, debugging, and exceptions. BriskChain sends the result of the current function directly to the next function of the workflow, without accessing the database unless any errors occur. Therefore, BriskChain uses less overhead than OpenWhisk by omitting unnecessary database accesses.

4.2.2 The Branch DAGs

The branch DAG form (Figure 4.1.b) selects one branch from several possible paths in the workflow through logical judgment. Specifically, supposing a branch node is followed by two branch sub-processes, and the node is a judicial function. If the result of the judicial function is true, the result is dispatched to the next first branch, otherwise, go to the second branch.

This subsection compares the performance overhead of Apache Composer and BriskChain in branching workflows. OpenWhisk itself does not support branch scheduling, so this round of comparison experiments uses Apache Composer and BriskChain. Nevertheless, OpenWhisk is still needed for the workflow programs because both Composer and BriskChain rely on OpenWhisk to run their serverless functions. Moreover, Composer and BriskChain do not support multiple branches (more than two branch structures) but can do so by nesting multiple current branches. This experiment does not evaluate the DAG use case under different numbers of multi-branches. Instead, only bidirectional branches are evaluated in the current experiment to both Composer and BriskChain, and the evaluated performance data includes the average overhead and 95% percentile latency.

Figure 4.3 shows the experimental results of the synthetic branch benchmark overhead, in which the experimental workflow consists of a judgment node and two subsequent branch nodes. Overall, it is found that the branch overhead results are similar

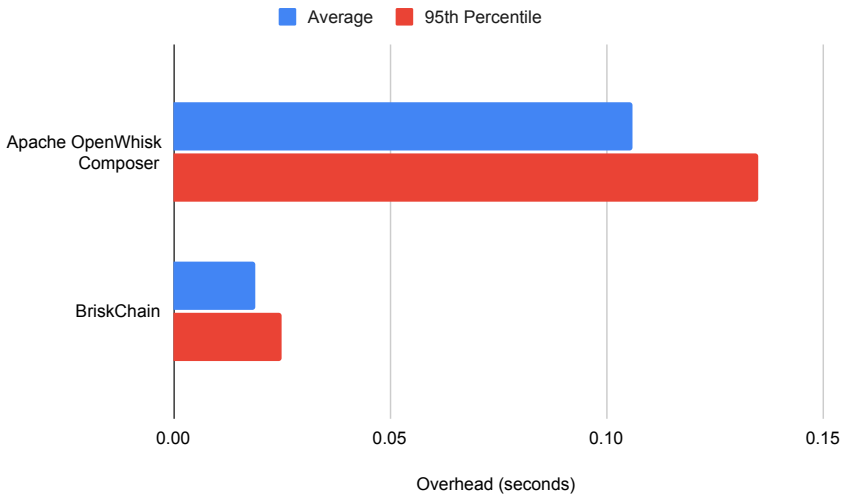

Figure 4.3: Overhead of Synthetic Branch

to the sequence experiments in the subsection 4.2.1: BriskChain spends less scheduling overhead than Apache OpenWhisk Composer in the branching DAG form. In the figure, BriskChain spends only 0.023 seconds on average which is the one-third overhead of OpenWhisk Composer. Additionally, BriskChain spends no more than 0.025 seconds of 95% of percentile latency which is 70% lower than Composer's 95% percentile latency value.

Analysis: Apache OpenWhisk Composer spends much more overhead than BriskChain because BriskChain dispatches the branch node directly from the condition node without waiting for the scheduling order by 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 used by BriskChain in the branching benchmark is also low, similar to the sequence experimental results. Specifically, in BriskChain, there is a customer-defined conditional function, the branch judgment code, which will be set in the BriskChain runtime (BriskChain sandbox 3.4.2). Its branch scheduling only depends on the judgment code and BriskChain runtime, without consulting the controller. In contrast, Apache OpenWhisk Composer relies

heavily on a centralized controller to schedule tasks, whose scheduling involves many remote TCP/IP connections between serverless functions and the controller. Therefore, BriskChain can handle branching workflows with less overhead.

4.2.3 The Parallel DAGs

This subsection evaluates the overhead of parallel DAG forms using BriskChain. The parallel, also known as fan-out and fan-in workflows, splits (fan-out) the task of a serverless function into multiple concurrent sub-functions, and collects (fan-in) the results of the sub-functions as the final result of the parallel. The experiment simulates synthetic parallel workflows composed of a different number of sub-functions and calculates their overhead 1000 times based on OpenWhisk Composer[4] (co) and BriskChain (bc).

Figure 4.4 shows a whisker plot of the overhead for OpenWhisk Composer and BriskChain in a different number of concurrent functions. Overall, BriskChain uses 50%-75% less overhead than OpenWhisk Composer across different numbers of parallel functions. Also, the overhead of BriskChain grows steadily with the number of parallel functions, but Composer shows exponential growth and great variability. For example, in a parallel test of 30 concurrent functions, the overhead of BriskChain is only 1/5 of the overhead of OpenWhisk Composer. In addition, BriskChain has stable overhead in each number of concurrent groups, and the minimum and maximum overhead of BriskChain is not far from its average value. However, OpenWhisk Composer exhibits a large variability. For example, in 30 parallel tasks, some executions use nearly 3000 milliseconds which is 50% higher than its lowest value.

Analysis ①: As observed, BriskChain has a stable and lower overhead than Composer. The reason is similar to the sequence and the branch evaluations described before:

[4]OpenWhisk has many limitations of concurrency. For example, the default concurrency number is 1, no action level reuse and no more than 60 innovations in a minute. These limitations are adjusted in our experiments to accommodate concurrency needs.

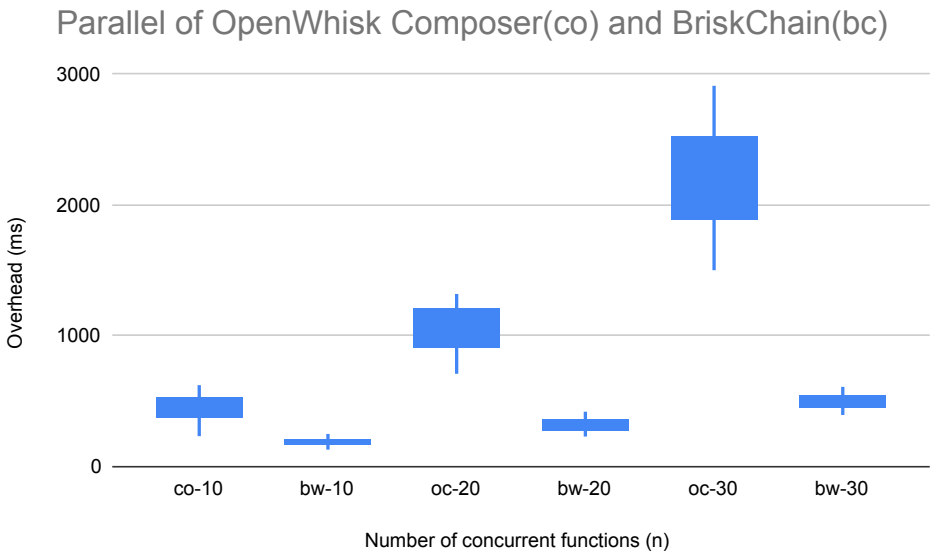

Figure 4.4: Whisker Plot of Parallel

the worker-side pattern eliminates internally distributed interactions between the function and the controller and then reduces the communication overhead of the workflow. Furthermore, the parallel scheduling of Composer requires complex multi-system dependencies, and these complexities result in Composer having higher and more variable overhead values than BriskChain. For example, executing parallel operations in Apache OpenWhisk Composer involves many distributed participants, including controllers, fan-out functions, CouchDB, and Redis. Delays in any one of these participants can cause delays in the entire workflow. As the number of parallel functions increases, the probability of latency becomes greater, and thus the variability. In contrast, BriskChain only involves local connections between the limited essential functions because of its worker-side pattern and the locality strategies. Therefore, BriskChain is characterized by stability and low overhead due to its simple worker-side pattern.

Analysis ②: Another reason that BriskChain shows higher performance than Composer in Figure 4.4 is about resource competition. The parallel DAG form consumes a lot of resources for a short period of time while executing its parallel subtasks and

can cause resource competition. This situation may not be prominent in sequential and branching DAG forms, but many subtasks in a parallel workflow are likely to cause large resource demand spikes. For example, since there are many cross-host interactions and data transfers between the functions of the workflow when Composer schedules parallel tasks in the workflow, it may cause network stress between the hosts for a short period of time. Consequently, the network stress will lead to instability and time delays in workflow processing. Fortunately, BriskChain's locality strategy (3.4.6) enables serverless functions of the same workflow to the same host. Therefore, the data transmission and scheduling of the workflow will be performed by BriskChain locally without the problem of network resource contention. This is why BriskChain shows higher performance and stable scheduling ability than Composer.

4.3 Payload Evaluation and Analysis

This section evaluates the performance of BriskChain under different payloads. The experimental workflow is built by composing six sequential functions. There are different sizes of payloads fed into the workflow, and each function of the workflow hands off the payload to its next function until the workflow ends.

Figure 4.5 shows the overhead of the workflow with different payloads. Overall, BriskChain uses less overhead with different payloads compared to OpenWhisk Sequence, and when the payload increases, BriskChain's overhead rise moderately. It can be seen that the OpenWhisk sequence uses roughly three times the overhead of BriskChain on a 1K payload. However, when the payload increased to 1M, the overhead of the OpenWhisk sequence quickly increased to 5 times that of BriskChain.

Analysis: Due to the worker-side pattern, BriskChain achieves small and stable overhead values on different payloads, as shown in Figure 4.5. Specifically, it greatly reduces the complexity of workflow scheduling, thereby reducing the overhead of the

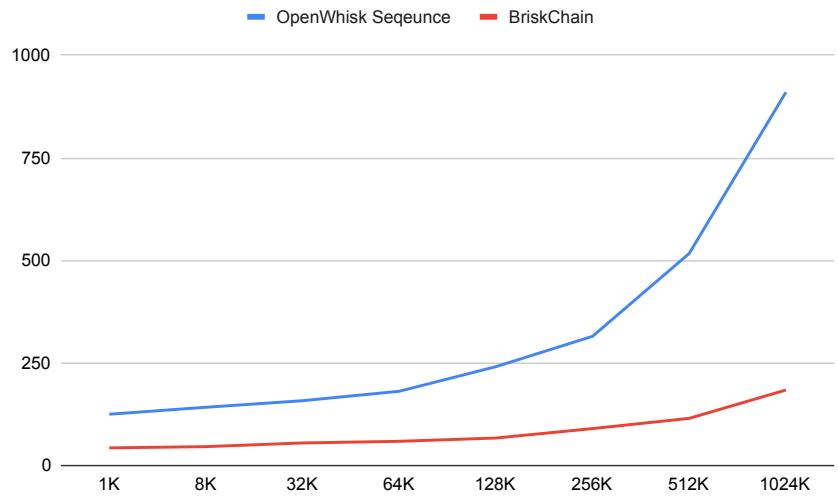

Figure 4.5: Overhead with Different Payloads

payload. Furthermore, since BriskChain's locality mechanism effectively avoids cross-host internal communications, even large-sized payloads can be efficiently transmitted and scheduled in the workflow. By contrast, OpenWhisk's performance drops dramatically as the size of the payload increases. In fact, many FaaS platforms even have limited payload size requirements. For example, OpenWhisk limits the default payload size for a serverless function to 1M. The reason is that transferring large amounts of state data between serverless functions across distributed hosts can lead to inefficiencies.

4.4 Real World Case Studies

After showing evaluations on synthetic micro-benchmarks, this section assesses workflow performance optimizations using two real-world applications. The two real-world case studies include a continuous integration pipeline and a video-converting application, and they have been built by the DAG forms of sequence, branch and parallel.

4.4.1 Travis2slack

Our first real-world serverless application was inspired by Travis2slack (*Travis CI to Slack*, 2022) which is like a robot constantly aware of continuous integration and continuous deployment (CI/CD) software projects. Travis2slack is a serverless application that responds to notifications from continuous integration applications. When there is any change in the status of the CI/CD project, it automatically analyses the project log and publishes the analysed report and the original error address to the subscribers.

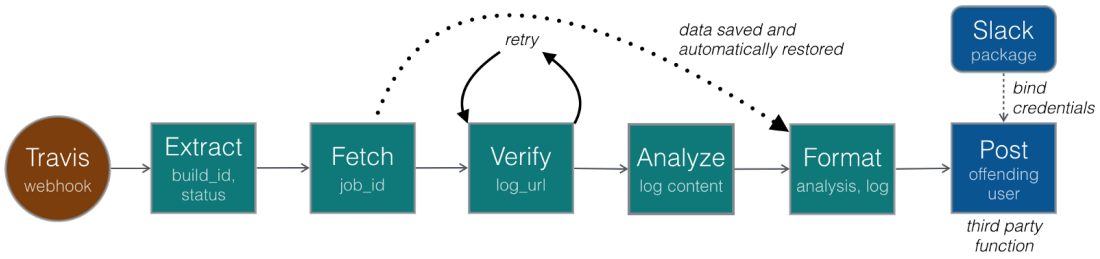

Figure 4.6: Pipeline of Travis CI to Slack

The process steps of Travis2slack are shown in Figure 4.6. Initially, Travis2slack uses a webhook to receive the build notifications from Travis-CI[5], and then retrieves the pull request and build details. The second main task is to fetch and analyse build and test logs. Finally, Travis2slack generates Slack[6] message for subscribers.

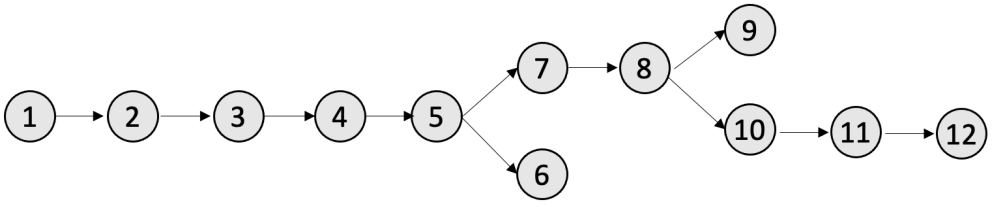

Figure 4.7: Workflow of Travis CI to Slack

Figure 4.7 shows the workflow of Travis2slack which consists of twelve workflow nodes. Table 4.2 shows the serverless functions of the twelve workflow nodes with their

[5]Travis-CI (Idera, Inc., 2022) is a hosted continuous integration service used to build and test software projects hosted on GitHub and Bitbucket.

[6]Slack (Slack Technologies, LLC, 2022) is a messaging program designed specifically for the office

descriptions.

	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 4.2: Workflow Nodes for the Travis2slack Benchmark

4.4.2 Video Transcoding

Video Transcoding (Vid) is a serverless application that converts video to other encoding formats, and it is inspired by a use case written by Alibaba Cloud (2022). Video transcoding is a resource-intensive task, and traditional transcoding application software consumes a lot of computing resources and takes a long time. Cloud computing can use its endless resources to quickly transcode videos. However, common serverless functions are limited by their own characteristics and cannot run the task for a long time. Fortunately, serverless workflows can be used to solve this problem.

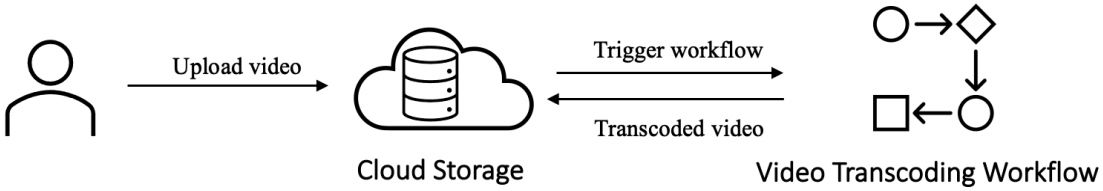

Figure 4.8: Video Transcoding Use Cases

Vid uses the serverless workflow to convert a video with major steps from splitting

to transcoding to merging. The splitting is to divide the video stream into a series of slice files according to the specified time interval. In order to break through the limitations[7] of the serverless function execution environment and speeds up the transcoding of large videos, the video is split into many small video pieces. The second major step is transcoding, which is to transcode the sliced video file into the required formats. In the workflow, many sliced videos are transcoded in parallel, so this is much more efficient than using traditional video transcoding. In addition, some optional jobs can be performed on sliced videos in parallel, such as porn detection or terror detection content security review. Finally, after doing all the work on the sliced videos, merge the sliced video results into one final video file.

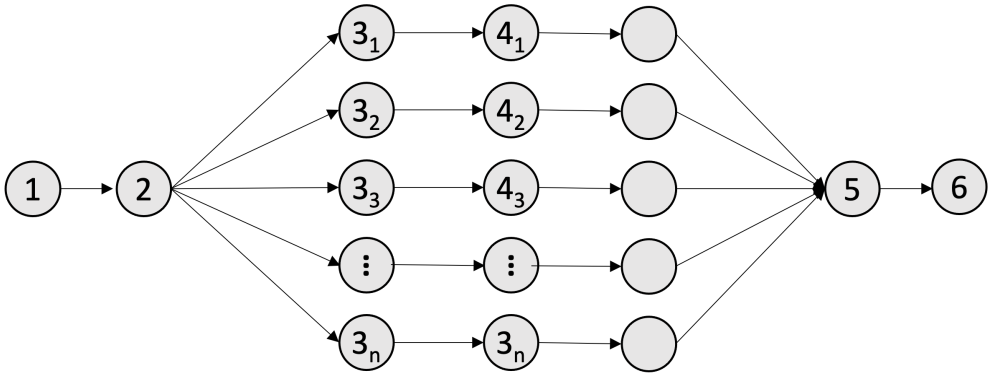

Figure 4.9: Workflow of Video Transcoding

Figure 4.8 describes how to use the video transcoding application. A user uploads the original video to the cloud video repository. This newly uploaded video will then automatically trigger the video transcoding workflow. After the workflow finishes transcoding, the transcoded video will be put back into cloud storage.

Figure 4.9 details Vid workflow. It consists of sequential and parallel workflow nodes. The number of parallels is determined by the video split interval. The shorter the interval time of video segmentation, the more nodes are transcoded in parallel in

[7]Serverless functions cannot execute for too long. For example, OpenWhisk has a default limit of 1 minute; Function Compute (Alibaba Cloud) has a limit of 10 minutes.

	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 4.3: Workflow Nodes for the Video Transcoding Benchmarks

this workflow. Table 4.3 presents the serverless functions of the workflow according to the index numbers.

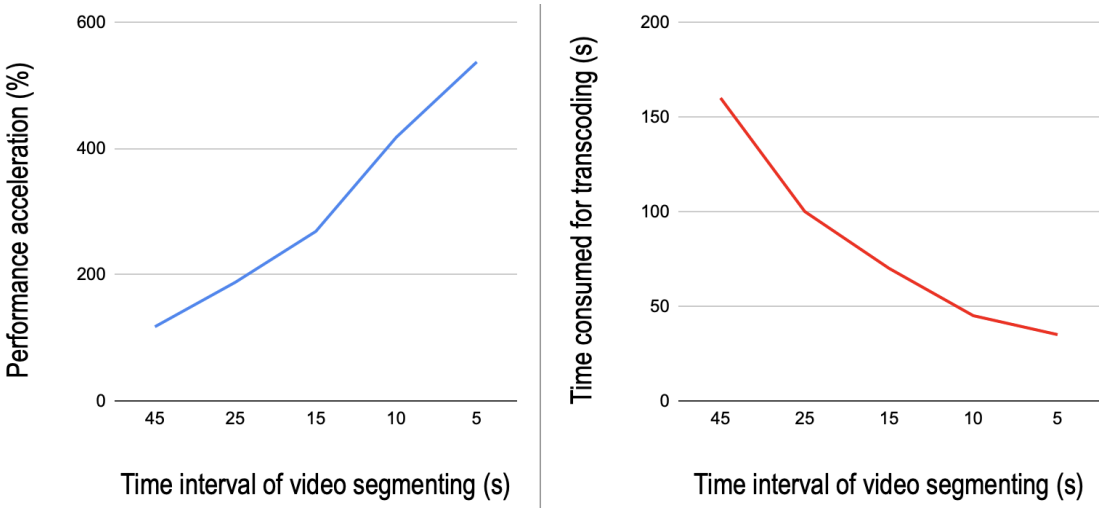

Figure 4.10: The Effect of Video Transcoding

The serverless workflow of video transcoding is effective. An experiment is conducted with a nearly 100-second video file, and it took about 190 seconds to transcode from MOV to MP4 format using the traditional converting method. Figure 4.10 shows the excellent transcoding performance of the video using serverless functions and Vid workflow. In general, as the time to split the video gets smaller, the time consumed by transcoding will decrease a lot, so the transcoding performance will be significantly improved. In the figure, if the video is split into two small video slices with a time

interval of 50 seconds, the workflow takes about 170 seconds for the transcoding, and the performance is improved by above 110%; if the video is split at an interval of 5 seconds, the workflow only needs about 38 seconds, and the transcoding performance is improved by more than 5 times.

4.4.3 Evaluate the Real-world Applications

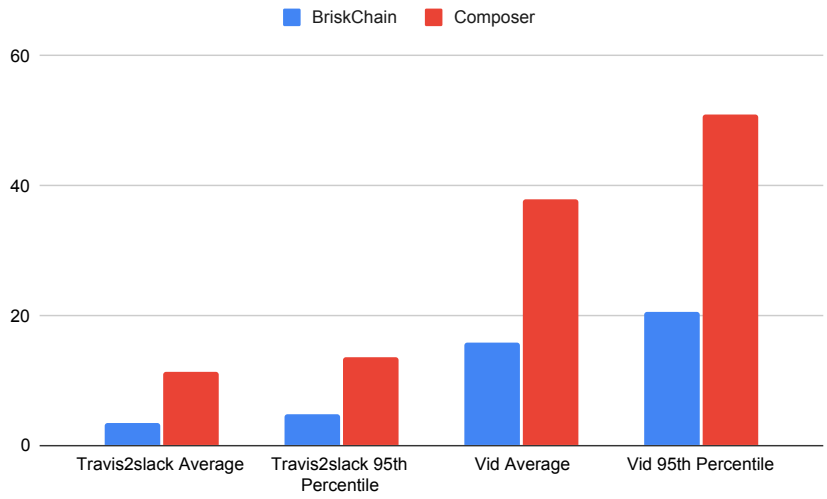

Figure 4.11: Overhead of the Real-world Applications

BriskChain exhibits a high scheduling performance on the two real-world serverless applications. Figure 4.11 shows BriskChain takes a shorter workflow scheduling time than Apache OpenWhisk Composer (Composer) in both study cases. Not surprisingly, the overhead results for the study cases are also similar to the values in the synthetic micro-benchmarks. In the figure, BriskChain spends an average of 3.5 seconds overhead on Travis2slack's workflow scheduling, which is only 30% of Composer's overhead of 11.3 seconds. In the case of Vid, BriskChain takes 60% less overhead on average than Composer (38 seconds) on the video converting with an interval time of 5 seconds. Additionally, the 95% percentile latency values for BriskChain and Composer increased by 35% and 38% respectively. Overall, the above overhead values demonstrate BriskChain

has high workflow scheduling performance, and its worker-side pattern and locality strategies are extremely effective for workflow scheduling.

4.5 Summary

BriskChain has been comprehensively evaluated, and the experimental results show that BriskChain is a high-performance workflow engine. The experiments had been conducted with diverse cases running with BriskChain, OpenWhisk and Apache OpenWhisk Composer (Composer). The experimental benchmarks include synthetic micro cases and real-world applications. In the synthetic micros benchmarks, BriskChain uses 70% less scheduling overhead in DAG forms of sequence and branch than using OpenWhisk sequence action and Composer. Also, it uses 50%-70% less scheduling overhead in the parallel synthetic benchmarks than using Composer. Moreover, the overhead value of BriskChain increases moderately than others when increasing the payload or number of nodes of the workflow benchmarks. Unsurprisingly, BriskChain still exhibits high scheduling performance on two real-world serverless applications, which only took 30% to 40% scheduling time than using Composer. Overall, the high-performance BriskChain demonstrates the effectiveness of our performance optimization strategies.

Chapter 5

Conclusion

5.1 Summary of Contribution

The thesis investigated performance optimization for serverless function composition. The composition of serverless functions (serverless workflows) was examined in detail and its key performance issues were identified. Then, potential solutions for performance optimization were studied and analysed. Based on the findings, a concrete workflow framework named BriskChain was further presented, which is equipped with the proposed optimization strategies. Finally, BriskChain was used to evaluate the effectiveness of the proposed performance optimization strategies. These detailed studies are further summarized below.

Based on our research, three performance issues related to serverless function composition were identified. First of all, due to the containerized isolation and zero-scaling capabilities of each serverless function, serverless functions suffer from the latency of their sandbox cold start. Furthermore, in a serverless workflow, all functions of the workflow accumulate cold-start delays, resulting in significant delays when executing the workflow. Secondly, due to the distributed nature of serverless functions in a cloud environment, the internal interactions between functions are cross-host,

leading to complex low-performance remote communication problems. Thirdly, most workflow engines are master-side pattern, which relies on a central controller to schedule tasks in the workflow. This pattern requires frequent interaction from the master side to the worker side to complete workflow tasks. Furthermore, serverless functions are fine-grained and numerous, which makes the overhead problem of function interactions in workflows more apparent.

After analysing existing optimization methods, the thesis proposed three performance optimization strategies for serverless workflows in detail. First, sandbox warm-ups can be used not only for ordinary single serverless functions but also for the functions of the workflow. Specifically, the study proposed a warm-up strategy to the serverless function sandbox of the same workflow based on predefined workflow logic. Second, orchestrating functions of the same workflow to the same host avoids remote interactions between the functions of the same serverless workflow. Third, using a worker-side pattern avoids the heavy overhead of frequent interactions from the worker side to the master side. Since there are many inner interaction requirements between the functions of the same workflow, the decentralized worker-side pattern can significantly reduce the overhead of internal interactions.

A high-performance workflow framework BriskChain was further designed and implemented utilizing the above strategies. BriskChain traits composed functions as a whole unit, and orchestrate the functions with predefined workflow logic. The novelty of BriskChain came from a totally decentralized worker-side pattern that any chained function in a serverless workflow can exchange the state to its next function directly without unnecessary centralized communications. Meanwhile, BriskChain used a locality strategy to avoid long-distance communication, so it extremely improved the performance of serverless workflow. Consequently, the pattern will offload the frequent communication burden in the serverless workflow.

Our evaluation shows BriskChain has high performance for the function composition

of FaaS. The experiments are conducted by synthetic micro-benchmarks and real-world applications. The experiments compare the scheduling overhead of BriskChain with OpenWhisk and Apache OpenWhisk Composer (Apache Composer). In the case of sequence and branch synthetic micro-benchmarks, BriskChain has a 70% reduction in scheduling overhead compared to OpenWhisk and Apache Composer. In parallel synthetic micro-benchmarks, BriskChain consumes only 30% to 50% of the overhead compared to Apache Composer. Meanwhile, as the workflow payload increases, the overhead of BriskChain increases moderately, which is lower than that of OpenWhisk and Apache Composer. In addition, benchmarks of the two real-world serverless applications demonstrate BriskChain's ability to integrate workflows using different DAG forms, and BriskChain also performs more efficiently than other competitors. Specifically, BriskChain's scheduling overhead uses only 30% to 40% of the overhead used by Apache Composer in the two real-world applications. Overall, the above evaluation results show that BriskChain is a high-performance workflow framework. Therefore, our proposed workflow performance optimization strategy is effective.

5.2 Future Direction of Research

The decentralized architecture can be widely used in the cloud and become a future research direction. Most FaaS systems are centralized consisting of master and worker side. The master (controller) side is responsible for many critical tasks such as auto-scaling, monitoring, resource management, load balancing, health checking or fault tolerance. The decentralized architecture can be widely used if offload these master-side tasks to the worker side. It is not limited to the worker-side pattern before but others as well. For example, auto-scaling is an important feature of every virtualized container, and is supported by the master in most FaaS systems. However, the auto-scaling ability can also be managed by the decentralized containers themselves rather than the master

(controller). Specifically, each container has its own life when it is created. If a running container is no longer active, its life is automatically terminated by itself after the predetermined length of life. That is how to fulfil the requirement of scaling down or scaling zero of containers. On the other hand, when the service demand increases, each container can also spawn more replicas by themselves to realize the scaling up.

References

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

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., ... Hilt, V. (2018). Sand: Towards high-performance serverless computing. In *2018 usenix annual technical conference* (pp. 923–935).

Alibaba Cloud. (2022). *Build an elastic and highly available audio and video processing system in a serverless architecture.* Retrieved 2022-09-30, from `https://www.alibabacloud.com/help/en/function-compute/latest/build-an-elastic-and-highly-available-audio-and-video-processing-system-in-a-serverless-architecture`

Ao, L., Izhikevich, L., Voelker, G. M. & Porter, G. (2018). Sprocket: A serverless video processing framework. In *Proceedings of the acm symposium on cloud computing* (pp. 263–274).

Apache Software Foundation. (2022a). *Apache openwhisk.* Retrieved 2022-09-30, from `http://openwhisk.apache.org`

Apache Software Foundation. (2022b). *Apache openwhisk composer.* Retrieved 2022-09-30, from `https://github.com/apache/openwhisk-composer`

Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., ... Tardieu, O. (2017). The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 acm sigplan international symposium on new ideas, new paradigms, and reflections on programming and software* (pp. 89–103).

Balis, B. (2016). Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows. *Future Generation Computer Systems*, *55*, 147–162.

Bhasi, V. M., Gunasekaran, J. R., Thinakaran, P., Mishra, C. S., Kandemir, M. T. & Das, C. (2021). Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the acm symposium on cloud computing* (pp. 153–167).

Carreira, J., Fonseca, P., Tumanov, A., Zhang, A. & Katz, R. (2019). Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the acm symposium on cloud computing* (pp. 13–24).

Carver, B., Zhang, J., Wang, A. & Cheng, Y. (2019). In search of a fast and efficient

serverless dag engine. In *2019 ieee/acm fourth international parallel data systems workshop (pdsw)* (pp. 1–10).

Castro, P., Ishakian, V., Muthusamy, V. & Slominski, A. (2019). The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry. *arXiv preprint arXiv:1906.02888*.

Cheng, Y., Chai, Z. & Anwar, A. (2018). Characterizing co-located datacenter workloads: An alibaba case study. In *Proceedings of the 9th asia-pacific workshop on systems* (pp. 1–3).

Daw, N., Bellur, U. & Kulkarni, P. (2020). Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st international middleware conference* (pp. 356–370).

Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., ... Chen, H. (2020). Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems* (pp. 467–481).

Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., ... Winstein, K. (2017). Encoding, fast and slow:{Low-Latency} video processing using thousands of tiny threads. In *14th usenix symposium on networked systems design and implementation (nsdi 17)* (pp. 363–376).

Guo, J., Chang, Z., Wang, S., Ding, H., Feng, Y., Mao, L. & Bao, Y. (2019). Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the international symposium on quality of service* (pp. 1–10).

Hassan, H. B., Barakat, S. A. & Sarhan, Q. I. (2021). Survey on serverless computing. *Journal of Cloud Computing*, *10*(1), 1–29.

Idera, Inc. (2022). *Travis ci.* Retrieved 2022-09-30, from `http://travis-ci.com`

Jia, Z. & Witchel, E. (2021). 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* (pp. 152–166).

Jonas, E., Pu, Q., Venkataraman, S., Stoica, I. & Recht, B. (2017). Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 symposium on cloud computing* (pp. 445–451).

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., ... others (2019). Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*.

Li, Z., Guo, L., Cheng, J., Chen, Q., He, B. & Guo, M. (2021). The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)*.

Li, Z., Liu, Y., Guo, L., Chen, Q., Cheng, J., Zheng, W. & Guo, M. (2022). 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* (pp. 782–796).

Lin, B., Zagalsky, A., Storey, M.-A. & Serebrenik, A. (2016). Why developers are

slacking off: Understanding how software teams use slack. In *Proceedings of the 19th acm conference on computer supported cooperative work and social computing companion* (pp. 333–336).

López, P. G., Sánchez-Artigas, M., París, G., Pons, D. B., Ollobarren, Á. R. & Pinto, D. A. (2018). Comparison of faas orchestration systems. In *2018 ieee/acm international conference on utility and cloud computing companion (ucc companion)* (pp. 148–153).

Mahgoub, A., Wang, L., Shankar, K., Zhang, Y., Tian, H., Mitra, S., ... others (2021). Sonic: Application-aware data passing for chained serverless applications. In *2021 usenix annual technical conference (usenix atc 21)* (pp. 285–301).

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

Meyer, M. (2014). Continuous integration and its tools. *IEEE software*, *31*(3), 14–16.

Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., ... Patterson, D. A. (2021). What serverless computing is and should become: The next phase of cloud computing. *Communications of the ACM*, *64*(5), 76–84.

Shahrad, M., Balkind, J. & Wentzlaff, D. (2019). Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture* (pp. 1063–1075).

Slack Technologies, LLC. (2022). *Slack.* Retrieved 2022-09-30, from `https://slack.com`

Travis ci to slack. (2022). Retrieved 2022-09-30, from `https://github.com/rabbah/travis-to-slack`

Wu, M., Mi, Z. & Xia, Y. (2020). A survey on serverless computing and its implications for jointcloud computing. In *2020 ieee international conference on joint cloud computing* (pp. 94–101).

Zheng, G. & Peng, Y. (2019). Globalflow: a cross-region orchestration service for serverless computing services. In *2019 ieee 12th international conference on cloud computing (cloud)* (pp. 508–510).

Appendix A

Glossary

Sandbox: It is a running environment of serverless function, which isolates each serverless function. It is also a basic running unit in the serverless applications. Most FaaS systems use Docker containers as sandboxes, so this thesis sometimes refers to sandboxes as containers.

Workflow: A complex serverless task composed by more than one serverless functions. The functions are chained one after another that one function's input comes from the output of its previous function. The terms of chained functions, composed function or DAG are interchangeable in this thesis.

DAG: It is a directed acyclic graph without cycles.

Composed functions: Same as the term of Workflow in the thesis.

Serverless computing: Serverless computing is a novel functional and cloud based programming paradigm.

FaaS: Same as serverless computing in this thesis.

State data: State data is the input and out data to a serverless function. The data can be transported to the the controller and to the other functions. It is considered internal data when dealing with serverless workflows.

Vertex: A workflow is composed by edges and vertexes, and the vertexes connected

by directional edges. A vertex also called workflow node in the thesis.

Node: There are two different nodes in the thesis. One is the workflow node that we also call it Vertex, and it is a function node of a workflow. Another one indicates a Kubernetes node, which is a host resource managed by Kubernetes engine.

Serverful: Contrary to serverless, users need to manage servers and other facilities by themselves.