DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration

Zijun Li*
Shanghai Jiao Tong University
Shanghai, China
lzjzx1122@sjtu.edu.cn

Chuhao Xu*
Shanghai Jiao Tong University
Shanghai, China
barrin@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Jieru Zhao
Shanghai Jiao Tong University
Shanghai, China
zhao-jieru@sjtu.edu.cn

Chen Chen
Shanghai Jiao Tong University
Shanghai, China
chen-chen@sjtu.edu.cn

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

ABSTRACT

Serverless computing that runs functions with auto-scaling is a popular task execution pattern in the cloud-native era. By connecting serverless functions into workflows, tenants can achieve complex functionality. Prior research adopts the control-flow paradigm to orchestrate a serverless workflow. However, the control-flow paradigm inherently results in long response latency, due to the heavy data persistence overhead, sequential resource usage, and late function triggering.

Our investigation shows that the data-flow paradigm has the potential to resolve the above problems, with careful design and optimization. We propose DataFlower, a scheme that achieves the data-flow paradigm for serverless workflows. In DataFlower, a container is abstracted to be a function logic unit and a data logic unit. The function logic unit runs the functions, and the data logic unit handles the data transmission asynchronously. Moreover, a host-container collaborative communication mechanism is used to support efficient data transfer. Our experimental results show that compared to state-of-the-art serverless designs, DataFlower reduces the 99%-ile latency of the benchmarks by up to 35.4%, and improves the peak throughput by up to 3.8X.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Cloud computing**.

KEYWORDS

FaaS, Function-as-a-Service, serverless workflow, workflow orchestration, control-flow paradigm, data-flow paradigm

*Both authors contributed equally to this research.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0394-2/23/03…$15.00
https://doi.org/10.1145/3623278.3624755

ACM Reference Format:
Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. 2023. DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3623278.3624755

1 INTRODUCTION

Serverless computing is popular in the cloud-native era [41, 47, 53, 66, 73], and cloud computing vendors all provide serverless computing services (e.g., AWS Lambda [5], Microsoft Azure Functions [7], and Alibaba Function Compute [13]). When a user submits a request with serverless computing, the corresponding function is triggered to run in a container. As the logic of a function is simple, a complex application is usually implemented as a function workflow. Traditionally, programmers build such a workflow by writing the code to trigger each function manually, and they also need to handle function orchestration, intermediate data transfer, and error retry [14, 30, 43]. Cloud computing vendors therefore offer an easier way to build and run a function workflow, called serverless workflow. With serverless workflow, programmers only need to declare the call dependencies between functions. The serverless platform automatically manages function orchestration, intermediate data, and fault tolerance [45, 48, 50, 54, 57, 58].

For a serverless workflow, based on its call dependency graph, serverless platforms (e.g., AWS Step Functions [6], Alibaba Serverless Workflow [1], OpenWhisk [17] and Fission Workflows [12]) use a workflow orchestrator (or controller) to trigger a function based on the completion status of its predecessor functions. As shown in Figure 1, the orchestration is typically done based on the control-flow paradigm. In general, the orchestrator maintains the states of all functions and triggers functions in a sequential order based on user-defined control dependencies. When all predecessors of a function are complete, the orchestrator triggers it for execution. Once triggered, the function's input data is loaded from the backend storage. If a function needs to transfer output data to a destination function, the sender stores the intermediate data in backend storage (e.g., remote storage) and the receiver reloads it [29, 37, 40, 61]. The persistence of intermediate data is necessary for the mainstream stateless serverless computing. Some platforms [6, 11] also provide workflow interfaces for stateful functions. They support the persistence of small data without backend storage (i.e. <256KB in

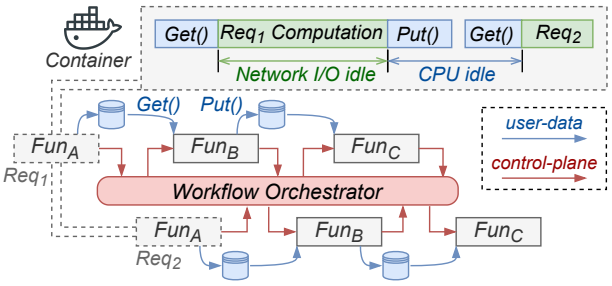


Figure 1: Run a workflow using the control-flow paradigm. An orchestrator decides when/where to trigger a function.

AWS Step Functions [16]). Larger data still relies substantially on backend storage even in stateful serverless platforms.

Analyzing the above steps, the orchestration based on the control-flow paradigm suffers from three main limitations. Firstly, the double transfer of the intermediate data from the source to the backend storage, and from the storage to the destination results in heavy communication overhead. Especially in the case that a serverless container has limited network bandwidth for fairness in production [27, 73]. Secondly, by default, the orchestrator cannot schedule a container to run multiple function invocations concurrently [4, 23, 36, 45]. The sequential steps of executing a function (i.e. loading data with Get(), executing the function, storing data to backend storage with Put()) will idle either the CPU or the network (the upper part of Figure 1). Resources are not fully utilized. Thirdly, the state management overhead and the state-driven triggering pattern of the orchestrator make the actual trigger time of a function later than its ready time.

The data-flow paradigm [34, 53, 63] has the potential to properly resolve the above issues. Following the data-flow paradigm, a data-flow graph describes the data dependencies between functions, and a function starts to run immediately once all its input data is ready. Based on the data-flow graph, the intermediate data flows asynchronously to the destination as soon as the data is generated, without relying on backend storage. This feature allows computation and communication to overlap. Moreover, each function determines whether it can start to run independently without a centralized orchestrator in an out-of-order triggering fashion.

It is nontrivial to use the data-flow paradigm in serverless workflows. Most importantly, the computation and data transmission logic are all tightly coupled in the user function code. The current serverless frameworks do not abstract data-flow graphs for workflows. It is necessary to redesign the serverless programming and execution model to separate a function's computation logic and the data transmission logic. In this way, mechanisms can be designed to overlap the computation and the communication.

Secondly, the workflow orchestration has to be re-architected to support the data-flow based function triggering. As serverless computing is typically applied to stateless applications, some function containers are generated/recycled, and the data-flow graph changes dynamically. The utilization of decentralized workflow orchestration engines, rather than a centralized orchestrator like control-flow, is paramount in ensuring precise communication and execution results, as well as achieving efficient synchronization of the up-to-date data-flow graph across all workflow functions.

Thirdly, the direct communication mechanism between functions without persistent storage should be carefully designed. The destination container may not even be started when the data flows to it from the source function, due to the cold startup and time-consuming user-related environment setup. A corresponding cache management mechanism are necessary on each host node to support the low-overhead function-to-function data transfer.

We therefore propose **DataFlower**, a data-flow paradigm-based serverless workflow scheme. In terms of the execution model, we abstract the container for a function as *Function Logic Unit (FLU)* and *Data Logic Unit (DLU)*. The FLU is responsible for executing the function code, and the DLU transforms the output data from the FLUs and sends it to the destination. On each node, a *workflow scheduling engine* parses the data-flow graph, and synchronizes the data-plane with DLUs. The engine replaces the workflow orchestrator to scale containers for functions, based on the transmission pressure on each DLU. In addition, DataFlower introduces a *host-container collaborative communication mechanism* to support the implicit data transfer between functions. A streaming-based pipe connector is used to minimize the data transfer latency. On each host node, a data sink is maintained for each function to temporarily hold its data. The data in the sink automatically expires once the data is loaded by the container, to reduce the memory overhead.

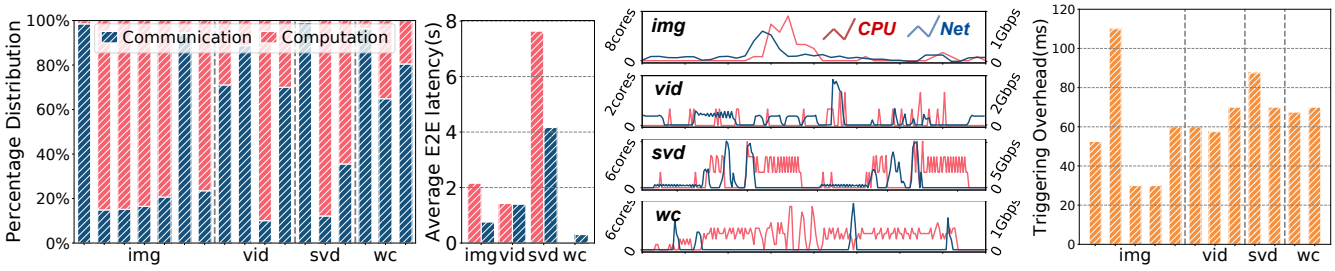
We have implemented the DataFlower mechanism and built it into FaaSFlow, an open source serverless workflow framework. To the best of our knowledge, DataFlower is the first work that uses the data-flow paradigm to orchestrate serverless workflows. The main contributions are as follows.

(1) **The semantic comparison and analysis of control-flow and data-flow paradigms for serverless workflows.** The investigation shows the drawbacks of using the control-flow paradigm for scheduling serverless workflows, and highlights the advantages of the data-flow paradigm.
(2) **The scheme of a data-flow driven function orchestration for serverless workflow.** It enables autonomous communication between functions, early function triggering, and computation-communication overlap in a container.
(3) **A set of mechanisms to improve the data-flow driven orchestration for serverless workflows.** We reveal that the communication backpressure and inefficient data lifetime management are performance bottlenecks. Pressure-aware function scaling and host-container collaborative mechanisms are proposed to tackle the bottlenecks.

We evaluate DataFlower using real-world applications. Extensive experimental results show that compared to control-flow based serverless designs, DataFlower reduces the 99%-ile latency of applications by up to 35.4% and improves the peak throughput by up to 3.8X, while reducing resource usage by up to 69.3%.

2 SERVERLESS WORKFLOW INVESTIGATION

In this section, we first explain the background of serverless workflow. We then discuss the control flow paradigm for serverless computing. Finally, we will analyze the potential of using the data-flow paradigm to alleviate the drawbacks.



(a) End-to-end communication and computation time breakdown (b) Resource usage timeline (c) Triggering overhead

Figure 2: The e2e latency breakdown, resource usage timeline, and triggering overhead with the control-flow paradigm.

2.1 Backgrounds of Serverless Workflows

In the cloud-native era, a complex application is often decoupled into fine-grained functions. Each function scales independently with serverless computing. Many popular applications [25, 31, 38] have already been decoupled and orchestrated to be a serverless workflow, to achieve the functionality of the original application [22, 45, 50, 54, 57, 58].

The existing workflow definition relies substantially on the Workflow Definition Language to describe the control and dependency logic between functions within a workflow, while building a state machine to templatize the trigger logic on the control-plane [8, 18]. In general, a serverless workflow can be represented by a DAG (Directed Acyclic Graph) [28, 32, 55, 67, 68, 70, 78], where the nodes represent the functions and the edges represent the control dependencies.

When a serverless workflow invocation arrives, functions are triggered according to dependencies. If all predecessors of a function are done, the orchestrator triggers it to run. If there is already a warm container for the function, it reuses the warm container directly. Otherwise, a new container is started to host the function, incurring the overhead of a cold startup [35, 52, 60, 62, 66, 71–73].

2.2 Serverless with Control-flow Paradigm

Current serverless platforms [6, 12, 17, 45, 50, 54, 56] adopt the control-flow paradigm to model and execute a serverless workflow. With control-flow, the data plane ensures the correctness of communication, and the control plane controls the invocation time and order of the functions.

With the control-flow paradigm, the generated data by a completed function should be persisted in a backend storage or passed to the destination function through the message queue. It is necessary to persist the intermediate data, as users are not aware of the data locality and containers may be recycled once timeout [29, 37, 56, 61, 62]. On the control-plane, a workflow orchestrator monitors the states of the functions, and decides when and where to trigger the next function based on the control dependencies and the data-availability.

We first run serverless workflows on 3 popular production serverless platforms (i.e., AWS Step Functions, Azure Durable Functions, Alibaba Serverless Workflows) to study their performance. These platforms use the control-flow paradigm to orchestrate a serverless workflow. Four production-level best practice serverless workflows, *Video-FFmpeg* (*vid*) [21], *ML-based Image Processing* (*img*) [20], *Singular Value Decomposition* (*svd*) [58] and *WordCount* (*wc*) [77], are used as the benchmarks to perform the investigation. The benchmarks cover the same serverless scenarios as in FaaSFlow [54] and WISEFUSE [58]. As an example, Figure 2 shows the characterization result on one of the serverless platforms. Other production platforms show similar characteristics.

2.2.1 Heavy Data Persistence Overhead. Figure 2(a) shows the computation and communication time breakdown of the benchmarks in the production serverless platform. In the figure, each bar represents a function in the workflow. The communication time consists of the time to retrieve the data from the backend storage and the time to store the data in the backend storage. We can observe that the communication time is dominant for many functions in all the four serverless workflow benchmarks [75]. For the four benchmarks, communication accounts for 26.0%, 49.5%, 35.3% and 89.2% of the end-to-end latency, respectively. One recent research also reveals the communication bottleneck in production environments [48]. The communication overhead would be even larger, if more concurrent or bursty queries from the same function use the backend storage concurrently.

The communication time is long due to two main reasons. First, the data is transferred twice (from the source to the backend storage, and from the storage to the destination). Moreover, backend storage still has high access latency and offers either limited I/O performance, especially for highly scalable serverless containers from a single function. The double data transfer and the limited network bandwidth result in the heavy data persistence overhead.

These platforms that allow stateful functions to communicate without remote storage can eliminate the data persistence overhead for the small intermediate data. For instance, the state machine in AWS Step Functions supports the data cache smaller than 256KB [16]. Large intermediate data is still persisted through the remote backend storage. They also suffer from the data persistence overhead when executing our benchmarks and other large-scale serverless workflows.

2.2.2 Sequential Resource Usage. Figure 2(b) shows the CPU and network usage timeline when running these benchmarks. In the experiment, we collect the resource usage on each container. The figure reveals the CPU and network usage of all running functions

in a benchmark. The functions run sequentially in this experiment. As observed, either the CPU core or the network of a container is staggered to peak usage during its lifetime. The network I/O resource is mainly consumed during the data access phase (Get() and Put()), while the CPU is waiting for the I/O completion, and the thread is suspended. Similarly, when running the computation logic, the network is idle. Sequential resource usage is inherent, as the computation and data transmission logic are tightly-coupled in the user code with the control-flow paradigm. If multiple invocations of a function happen, the invocations queue up.

2.2.3 Late Function Triggering. The functions are triggered sequentially because the workflow orchestrator automatically enables the sequential execution of diverse logics (i.e, parallel, switch, and foreach). To maintain the correct order when processing these functions, the workflow orchestrator schedules and triggers the functions in a workflow sequentially in the topological order. This in-order triggering pattern prevents a function from being triggered early, even if all its input data is ready. Experiences in the computer architecture field have shown that out-of-order execution improves performance.

Figure 2(c) shows the time needed to manage the function states for ordering the function triggering on the control-plane with the production orchestrator. The triggering overhead is calculated to be the duration between the end timestamp of a function and the start timestamp of its successive function in the log file. As observed, it takes 63.3 milliseconds on average to manage the states between two adjacent functions. The overhead is relatively long, when a function's execution time is often short for serverless computing.

The three drawbacks of the control-flow paradigm originate from its inherent logic, and thus cannot be alleviated through minor modification. For instance, FaaSFlow [54] uses local memory to cache data for local functions, and SONIC [56] caches the data in the source function and the destination function fetches the data when it is triggered. However, they all suffer from sequential resource usage and late function triggering with the control-flow paradigm. The data is not transferred to the destination beforehand. We also show the performance of FaaSFlow and SONIC in Section 8.

2.3 Merits of Data-flow Paradigm

The data-flow paradigm that defines the data dependencies explicitly is capable of enabling out-of-order execution and maximizing the parallelism. The data dependencies are expressed to be a data-flow graph in the data-flow paradigm. For each function, the data-flow graph records the source functions of its inputs, and the destination functions of its outputs.

Figure 3 illustrates the better way we proposed to run a serverless workflow using the data-flow paradigm [34, 53, 63]. The basic idea here is decoupling and processing the logic of computation and communication independently. In this way, the data-passing operations may be done asynchronously when the container executes the function, and a container can run the next function invocation before the data passing completes. The data-flow paradigm resolves the drawbacks as follows.

2.3.1 Alleviating Data Persistence Overhead. With the up-to-date data-flow graph, a function knows destinations of its outputs. It is possible to directly transfer the data to destinations without back-end storage. Moreover, with direct transfer, the streaming technique can be introduced. When a data chunk is ready, it can be transferred to the destination before the entire data is generated. The streaming technique can be implemented with a pipeline connector that connects the source and the destination.

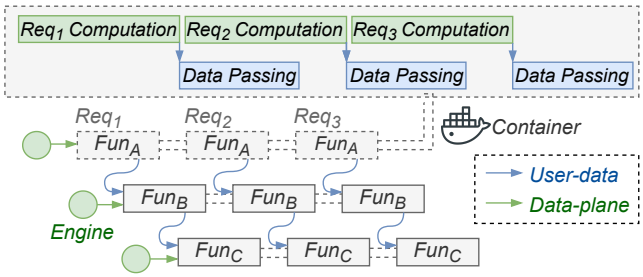


Figure 3: Run a workflow using the data-flow paradigm. Data flows to destinations according to the data-flow graph.

2.3.2 Improving Resource Utilization. As shown in the upper part of Figure 3, a container can execute a new request before the data transfer of the previous one completes. In this way, the computation and the communication overlap, and resource utilization is improved. The overlap is beneficial for reducing the response latency of a workflow at a high load, as the queries may start earlier.

2.3.3 Early Function Triggering. The data-flow paradigm enables the early function triggering from two aspects, as shown in the lower part of Figure 3. As for the first aspect, it enables out-of-order function triggering. As for the second aspect, the data of a function can start to be transferred to the destination functions earlier with the data-flow paradigm, and the destination's input data is ready earlier. It is possible that the data is generated in the middle of the function, but is transferred to the destination after the function completes, using the control-flow paradigm.

Based on the above analysis, we have two main insights. Insight 1: The widely-used control-flow paradigm inherently fails to minimize the end-to-end latencies and maximize the throughput of serverless workflows. Insight 2: Data-flow paradigm enables the design of reducing the data transfer overhead, overlapping compute and data transfer, and early function triggering and execution.

2.4 Challenges in Applying the Data-flow

Designing a serverless workflow scheduling scheme based on the data-flow paradigm faces three main challenges.

Challenge-1: The computation and data passing should be decoupled, however current serverless frameworks do not support such operation. The programming interface and the execution model should be re-designed to support the explicit data-flow graph declaration, and enable the ability to perform the computation and data transmission asynchronously.

Challenge-2: The workflow engine should fundamentally support the decentralized function triggering based on data-availability. However, current frameworks often use a centralized orchestratior. In particular, the workflow engine must be capable of parsing the data-flow graph, monitoring the availability of input

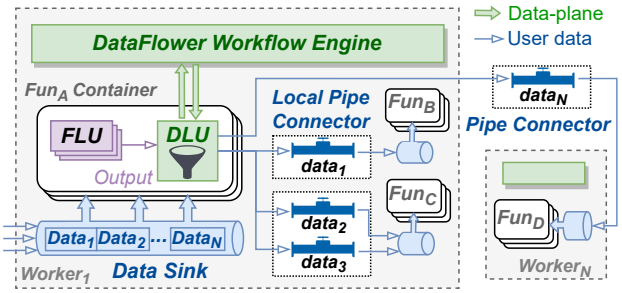
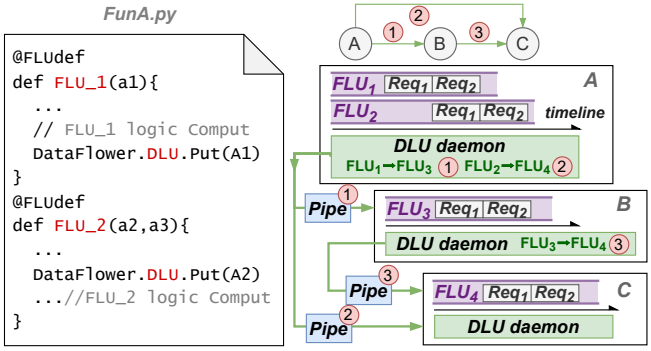


Figure 4: The scheme overview of DataFlower.



(a) Pseudocode of separating FLU and DLU. **(b) Example of the FLU triggering and DLU interaction.**

Figure 5: The programming and execution model that supports the FLU and DLU abstraction.

data for functions, and dynamically scheduling available containers for function invocations in a decentralized manner.

Challenge-3: The direct data communication mechanism should be carefully designed to support implicit data-plane. When a function transfers its data to the destination, the destination function has not yet been triggered and no container can receive the data. Therefore, the host node of the destination function should properly manage the data before the function is triggered, and recycle the data to reduce the memory overhead.

3 DESIGN METHODOLOGY OF DATAFLOWER

Figure 4 presents **DataFlower**, a scheme that achieves the data-flow paradigm for the serverless workflows. As shown, in each physical node, DataFlower deploys a *data-flow based scheduling engine* to schedule all the functions on the node. A container for a function is abstracted into a *function logic unit* (FLU), and a *data logic unit* (DLU). The FLU runs the function, and the DLU receives the input data for the function, and pushes the output data to the destination functions. Moreover, between each source and destination function pair, a *pipe connector* is used to transfer the intermediate data.

DataFlower runs a serverless workflow in following steps. First, once the workflow is invoked, DataFlower relies on the default function mapping method to assign functions to different nodes. DataFlower also provides an open interface for the function mapping method, so that other function mapping solutions can be implemented. Second, the data-flow graph is informed to the workflow engine on each node. Third, according to the data-flow graph, the workflow engine monitors whether each function's input data is ready. If all the input data of a function is ready, the engine triggers the function, and finds it an appropriate container to run it. Fourth, the container uses the FLU to run the function, and the DLU of the container writes the output of the function to the destination nodes, according to the data-flow graph, using the pipe connector.

Note that, when the data is ready to be transferred, it is possible that the container of the destination function has not been set up yet. In this case, the transmission is blocked. To this end, as shown in Figure 4, DataFlower designs a *host-container collaborative communication mechanism* that caches the data in the *data sink* on every host node. The data sink of a node accepts and caches the data from all the other nodes, so that a function is able to quickly obtain its input data, once it is triggered to run.

Due to the time-consuming data transfer per container, it is also possible that the DLU pushes the data through the pipe connector at a much slower rate than the data generated into the DLU. In this case, it is better to scale out more containers for the function to avoid the backpressure problem. We further propose a *pressure-aware function scaling mechanism* which monitors the data stream pressure on the DLU and determines whether to scale out the function containers.

4 THE FLU-DLU ABSTRACTION

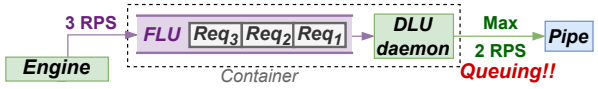
In this section, we first introduce the programming model that supports the FLU and DLU abstraction. Then, we present a pressure-aware function scaling mechanism that enables the container auto-scaling for each function in a workflow.

4.1 Programming and Execution Model

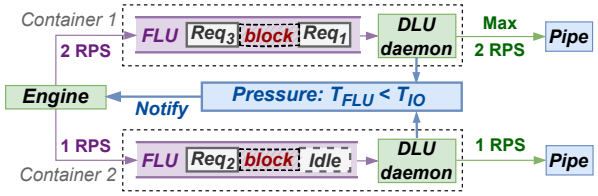
With the data-flow paradigm, the function computation logic and the data transmission logic should be separated to enable the potential computation and transmission overlap. However, the current workflow programming languages for serverless computing do not support the separation.

Figure 5(a) shows the programming model in DataFlower that supports the computation and data transmission separation. As shown, DataFlower provides an interface for the function to declare the data to be handled by the DLU. The FLU then executes the function and the DLU receives data from the function FLU and transfer data to the destinations. A FLU starts when a function is invoked in the container, and the DLU runs as a daemon process in the background.

It is possible for a function to generate data for multiple child functions (e.g. parallel, foreach). For simplicity, one possible design is to use the DLU to handle all the data at the end of the function. This design results in the late transfer, as some data may already be ready in the middle of the function. To this end, in the DataFlower programming model, whenever the data for a function is ready, the DLU should be called to initiate the transmission immediately. If a FLU generates data that has multiple optional destinations (i.e. switch), its DLU is able to select the correct destination based on the user-defined data-flow logic. Therefore, DataFlower naturally supports dynamic DAG declarations [28].



(a) Backpressure occurs if FLU produces more than DLU consumes.



(b) Pressure-aware notification for FLU-DLU scheduling.

Figure 6: The pressure-aware function scaling mechanism.

The DLU should be called at least once in each FLU definition to ensure that the intermediate data can flow out and trigger the next function FLU. For the terminal FLU function, an *end* signal is required to exit even if a user does not need a returned result. In addition, the computation of a function can be divided into multiple FLUs to increase the intra-function execution parallelism. Such abstraction further empowers the pipeline execution of the FLUs within a container.

Figure 5(b) shows the execution of a data-flow graph. As shown, the FLUs and the DLU of a container run asynchronously, and multiple FLUs in a function can run in a pipeline to maximize the parallelism. The DLU of a container on a node collaborates with the node's engine to transfer data. For each workflow request, a data-plane is provided by the scheduler and loaded into the DLU daemon to locate the destination functions (e.g. IP address, socket ID). FLU_3 can be triggered early when data arrives from FLU_1, without the necessity of waiting for Fun_A to complete.

4.2 Pressure-aware Function Scaling

With the FLU-DLU abstraction, the data transfer of a function becomes an asynchronous operation. The asynchronous operation may result in a long response latency. Figure 6(a) shows such an example. In the figure, the FLU receives 3 function requests per second, but the DLU daemon is only able to transfer the data to the destination for 2 requests per second. In this case, the queuing effect can result in severe long response latency throughout the serverless workflow.

It is not a problem if the data transfer is a synchronous operation, as in the control flow paradigm, because the serverless engine will create new containers for subsequent invocations when the container is processing a function. The situation is even worse for small function containers (e.g. 128MB), which are allocated with a lower upper limit of network bandwidth. A technique is required to tackle the problem due to asynchronous data transfer with the DLU daemon.

We therefore propose a pressure-aware function scaling mechanism as shown in Figure 6(b). Specifically, before the DLU receives data from the FLU, it will simultaneously acquire the average execution time T_{FLU} and the size of the data to be transferred *Size* of this FLU computation. On the basis of a prior knowledge of the container's network bandwidth B_w, Equation (1) calculates the expected data transfer pressure.

$$Pressure(FLU_f) = \alpha \frac{Size}{B_w} - T_{FLU} \quad (1)$$

In the equation, $Size/B_w$ represents the ideal transfer time, and α indicates the loss factor during actual data transfer. This loss factor is determined by the characteristics of the connector implementation. Suppose $Pressure(FLU_f) \leq 0$, the DLU can entirely consume the data generated from the FLU computations and does not need to scale up the containers. In this case, queries should remain dispatched simply based on idle FLUs within containers. Otherwise, if $Pressure(FLU_f) > 0$, backpressure happens, and the DLU sends a Callstack blocking signal to the workflow engine to block its FLU for $Pressure(FLU_f)$ time.

In this way, the FLU of the container cannot serve subsequent invocations, and the FLU-producing rate is limited to the maximum DLU-consuming efficiency. At the same time, the engine follows the serverless manner to scale out the containers to handle the remaining workflow invocations.

```
Workflow_name : wordcount
dataflows:
   wordcount_start: FLU_0: // FLU define
      input_datas:
         source:$USER.input
      output_datas:  // Output of FLU_0 define
         filelist:{type:FOREACH}
         destination:
            wordcount_count:{FLU_1:{file}}
   wordcount_count: FLU_1:
      input_datas:   // FLU_1 input define
         source:file:{type: NORMAL}
      output_datas:
         count_result:
            type:MERGE  // DLU Data destination
            destination:
               wordcount_merge:{FLU_2: {count}}
   wordcount_merge: FLU_2:
      input_datas:
         source:count:{type: LIST}//multi-input
      output_datas:
         output:{type: NORMAL}
         destination: $USER
```

Figure 7: Pseudocode of defining the data transfer relationship of functions in the benchmark *WordCount*.

5 SERVERLESS WORKFLOW EXPRESSION

With the FLU and DLU abstraction, the computation and the communication is decoupled. However, the data-flow graph for a workflow is still unavailable only with the abstraction, as the destination of a function's data is still not defined. With control-flow, the programmers need to define the function triggering relationships, the input data, etc. Similarly, the programmers need to define the output data and the destination of a function in DataFlower. As an example, Figure 7 shows the pseudocode of defining the data transfer relationship for the benchmark *WordCount*. In general, for each FLU, we need to define the source of the inputs and the destination of the outputs. However, a serverless workflow is still not able to run only with the data transfer relationship. This is because a function does not know on which node and containers the destination function of its outputs runs.

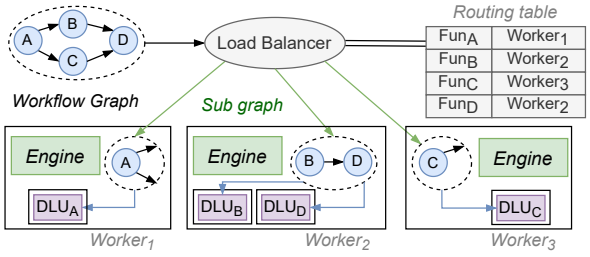


Figure 8: The upper load balancer maps functions to each worker node for data-flow graph parsing.

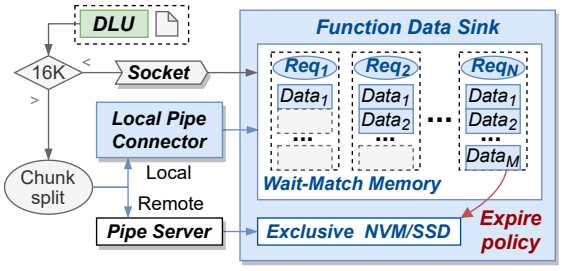


Figure 9: The function locality-aware pipe connector establishing, and the data structure organization in the data sink.

5.1 Workflow Scheduling with DataFlower

To resolve the above problem, DataFlower obtains the function deployment information from the load balancer. Figure 8 shows the way to parse the data-flow graph. In order to minimize the overhead of accessing the data-flow graph, the workflow scheduling engine in DataFlower is designed to be a decentralized component. As shown in the figure, DataFlower deploys a workflow scheduling engine on each node that hosts at least one function of the serverless workflow. Each engine parses the data-flow graph based on the code in Figure 7 and the function deployment information. For an engine, only the part of the data-flow graph related to the functions on the local node needs to be parsed.

The engine maintains a pool of containers for local functions. When a function is triggered, the local workflow engine allocates an idle container FLU or cold starts a new container to run it. The auto-scaling policy in serverless computing determines whether to scale out a new container for a function. In addition, the pressure-aware function scaling mechanism in Section 4.2 can also suggest starting a new container for a function due to queuing on the DLU.

Note that DataFlower does not rely on a specific load balancer. DataFlower exposes an interface to the upper load balancer for customized function deployment policies. DataFlower provides explicit control of the data-plane, allowing it to work with various load balancing strategies for high extensibility and flexibility.

5.2 Fault-tolerance and Data Consistency

DataFlower's fault tolerance model supports at-least-once or exactly-once execution semantics based on data flow persistence and integrity. The scheduling engine ensures that a function is not triggered if its predecessor fails or only sends partial data due to the data plane interrupt. In addition, the pipe connector itself periodically creates checkpoints asynchronously and incrementally to handle error retries. Based on the checkpoints, the workflow scheduling engine locates the workflow invocation to be retried, identifies the last pipe connector with correct data checkpoints, and *ReDo* the associated failed function execution through backtracking.

In addition, DataFlower's workflow scheduling engine uses a customized container keep-alive strategy to ensure data consistency. With the control flow paradigm, a container can be safely recycled if it is not processing a function. With the data flow paradigm, a container FLU may not be processing a function, but some destination data sinks may still be pumping new data from the container DLU through a pipe connector. In this case, if the container is identified as idle and recycled, data consistency will not be guaranteed. To this end, when using a keep-alive strategy, DataFlower ensures that a container is not recycled unless the container FLU is idle and there is no data remaining in the container DLU to be pumped.

6 HOST-CONTAINER COLLABORATIVE COMMUNICATION MECHANISM

With the data-flow paradigm, a function is triggered to run based on the data availability. A container will not be assigned until all input data of a function is ready, which indicates that the input data cannot be loaded immediately by the corresponding FLU. To cache and manage the input data for a function before triggered, we propose a host-container collaborative communication mechanism, as shown in Figure 9.

Specifically, DataFlower uses a *data sink* on each host node to temporarily cache the input data of all the functions on the node. When the DLU of a function determines to transfer the data to its destination function, a pipe connector is built between the source node and the destination node. If a single predecessor has to communicate with many child functions, The DLU of the predecessor will send the data to child functions through different pipe connectors in a FIFO fashion. The data is transferred to the data sink of the host node that deploys the destination function. Once the function is triggered, the corresponding container fetches the data from the data sink.

There are two ways to implement the pipe connector. A local pipe connector is established when the data-plane in the DLU indicates that two functions are on the same node. In this case, the data stream is pumped directly to the data sink. If the two functions are on different nodes, a pipe connector that supports data streaming is used to transfer the data across the nodes. For small data under 16K, DLU does not use a pipe connector and splits it, passing it directly to the destination via socket. Such a pipe connector mechanism can fully exploit function locality to improve data-flow efficiency.

It is also possible for a function to require input from multiple functions (e.g. merge), and the input from these functions may arrive in different order in different invocations of the serverless workflow. To boost the data loading of the DLU, we allocate a dedicated Wait-Match Memory [42] to maintain the cached inputs of a function. The Wait-Match Memory acts as a high-performance key-value store, and we design a multi-level index structure (*RequestID*, *FunctionName*, *DataName*) to further reduce the indexing overhead of a huge data sink.

Another challenging problem is that a function data sink maintains different data from different *RequestID* and cannot consume the data and trigger the FLUs in time. As shown in Figure 9, when the source function first pumps $Data_M$ of Req_N into the data sink, it will continue to occupy memory until the remaining data copies of Req_N arrive. To minimize the memory overhead for data caching, DataFlower proposes a hybrid mechanism to perform fine-grained data management in the Wait-Match memory and alleviate the data backlog.

- **Proactive release.** As soon as all destination FLUs have received the data, this intermediate data can be proactively released from the Wait-Match Memory. In the control-flow paradigm without prior knowledge of data dependencies, the caching design such as FaaSFlow can only remove the cache after each request completion.
- **Passive expire.** To further reduce the data footprint in Wait-Match memory, we use the passive expire mechanism based on a cache timeout. In this method, each piece of data is given a TTL (Time To Live), and it will expire and persist to the function-exclusive disk once timeout occurs.

The proactive release approach removes useless data, and the passive expire approach guarantees the data freshness in the memory cache. Therefore, both can increase the cache hit rate with limited Wait-Match memory.

7 IMPLEMENTATION OF DATAFLOWER

We have implemented and open-sourced DataFlower in FaaSFlow (https://github.com/lzjzx1122/FaaSFlow), which involves three modules: decentralized workflow engine (1466 LOC), function manager/invoker (546 LOC), and container encapsulation (902 LOC), excluding third-party libraries. In our implementation, each FLU is a thread created by the executor in the container, and the *WORKDIR* of each thread is exported into the container as an environment variable via the hash value of the user *RequestID*. We use a fixed 10-min keep-alive strategy for each FLU in the container. The computing resources of each function container are limited by Linux Cgroup, and the network is also limited by Linux TC.

Due to the performance degradation of the document-oriented CouchDB [9] via REST APIs, we implement the pipe connector based on kafka [3] that supports event streaming for efficient data pipelines. In the Kafka-based pipe connector, each data-flow corresponds to a topic, and each container is assigned a partition by default. The data sink implementation is based on a local SSD bound to the volume (for inter-node data storage) and Redis [65] (for intra-node data cache). The fault tolerance of maintaining states in Wait-Match Memory and exclusive NVM/SSD depends on the basic fault tolerance mechanism that provided by Kafka and Redis. The communication within the node consists of two aspects, the triggering of function requests and the data-plane synchronization during the execution of the DataFlower. The function invocations are triggered by REST APIs with the executor inside the container, while the interaction between the executor, the DLU daemon, and the host workflow engine establishes socket-based connections. DataFlower also uses a CouchDB server to collect the logs during execution.

8 EXPERIMENTAL EVALUATION

In this section, we first evaluate DataFlower in reducing the response latency and increasing the peak throughput. Then, we break down the effectiveness of the mechanisms in DataFlower and show the adaptiveness of DataFlower for various workflows. Lastly, we also discuss the architectural implications.

8.1 Experimental Setup

We evaluate DataFlower on a 5-node cluster. We use one node to generate workflow invocations, one node to be the backend storage node, and three nodes as the worker nodes. CouchDB [9], a widely-used document-oriented database, is used to be the backend storage for persisting the intermediate data (replaced with one Kafka node to support pipe connector for DataFlower). Each node is equipped with Intel Xeon (Ice Lake) Platinum @3.5GHz CPU (the load generator node and the backend storage node have 8 cores and 16GB memory, and each worker node has 16 cores and 64G memory) with 200GB SSD (3000 IOPS). We still use the four best practice serverless workflows, *Video-FFmpeg* (*vid*) [21], *ML-based Image Processing* (*img*) [20], *Singular Value Decomposition* (*svd*) [58] and *WordCount* (*wc*) [77], as benchmarks.

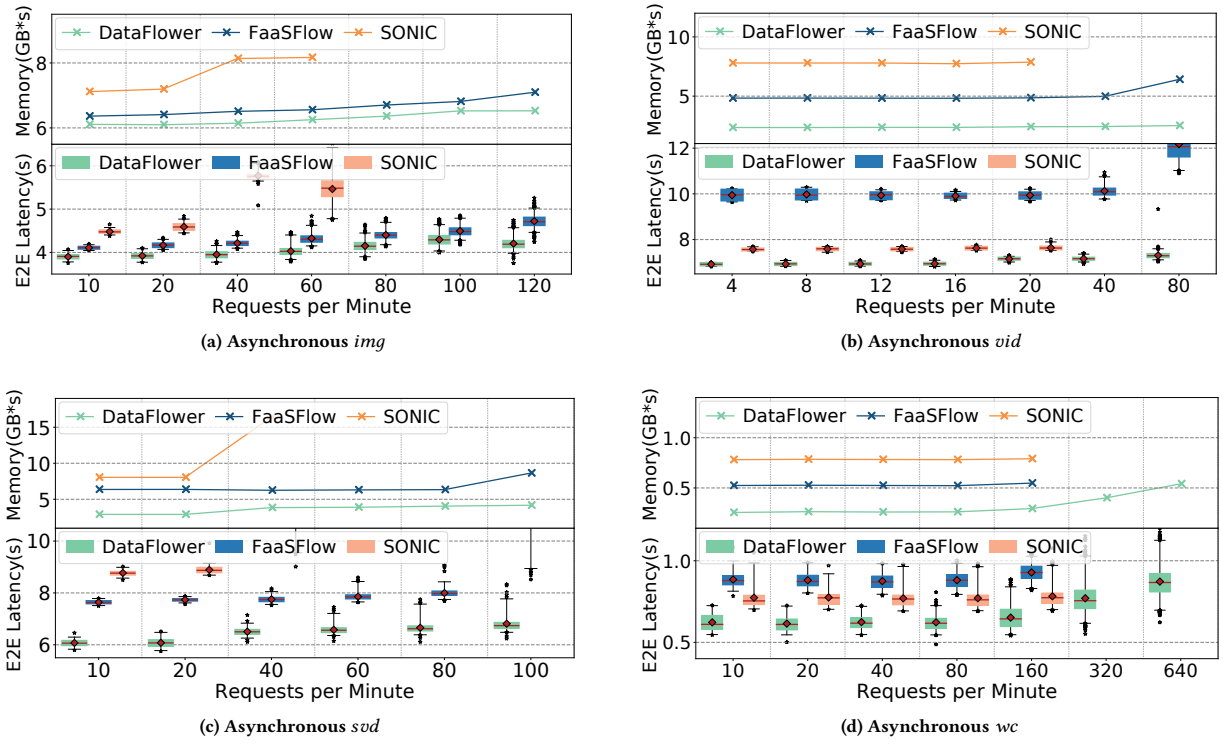
The container used to run a function is configured with the practical specification. Based on the previous study [62], the container network bandwidth increases as the container scales up. Following this pattern, we allocate 0.1 core and 40Mbps network bandwidth for a 128MB-sized container. The resources are allocated proportionally according to the container memory size. We also evaluate the impact of the container specification on the performance of DataFlower in Section 8.7.

We compare DataFlower with sota serverless workflow systems, FaaSFlow [54] and SONIC [56]. In FaaSFlow, a decentralized scheduling pattern is used to reduce the scheduling overhead in the workflow, and enable data transferring through memory for co-located functions. We implement SONIC by replacing the backend storage in FaaSFlow with local storage. The data to be transferred is persisted in the host, and then each destination function container builds a peer-to-peer connection with the source storage to fetch data in parallel.

We evaluate DataFlower with both synchronous invocations and asynchronous invocations [19], as production serverless systems provide the two patterns. With the synchronous invocation pattern, the requests are generated in a closed-loop, where a new request is generated after the previous request completes. We increase the load in this scenario by increasing the number of client threads that generate the requests. With the asynchronous invocation pattern, the requests are generated in an open-loop with a given load. The results with the synchronous invocation and asynchronous invocation patterns reveal the peak throughput that can be achieved with these systems, and the tail latency at a given load, respectively.

8.2 Response Latency and Peak Throughput

We first report the result with the asynchronous invocation pattern. Figure 10 shows the end-to-end response latencies and the memory resource usage of the benchmarks at different loads. Suppose N GB memory is occupied by t seconds, and the memory resource usage is calculated to be $N \times t$. The metric quantifies the cost of running



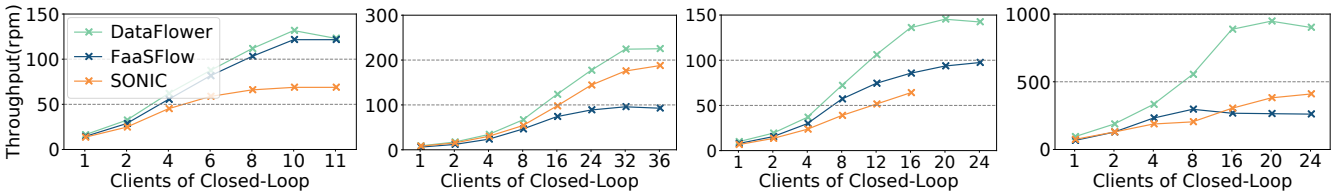
(a) Asynchronous *img*

(b) Asynchronous *vid*

(c) Asynchronous *svd*

(d) Asynchronous *wc*

Figure 10: The end-to-end latency and memory usage of the benchmarks with the asynchronous load generation pattern.



(a) Synchronous *img*

(b) Synchronous *vid*

(c) Synchronous *svd*

(d) Synchronous *wc*

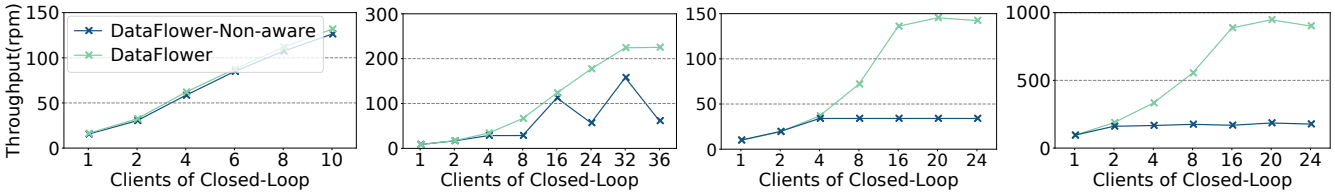
Figure 11: The throughput of the benchmarks when different numbers of clients generate requests synchronously.

the serverless workflow. The missing points/boxplots in the figure mean the benchmark suffers from timeout due to the huge latency.

DataFlower reduces both the average and tail latency of the benchmarks compared with FaaSFlow and SONIC. Specifically, it reduces the 99%-ile latency of the benchmarks by 5.7% to 35.4% compared with FaaSFlow, and by 8.9% to 29.2% compared with SONIC.DataFlower reduces the response latency, because the input data of the function is already prepared on the host. In this way, DataFlower does not need to fetch the data from the backend storage during the function execution as FaaSFlow does. SONIC also directly obtains the data from the source function to its local storage. However, the limited bandwidth of each container results in a long data transfer time. The slow data transfer further incurs the queuing of requests at the containers at high load, without the pressure-aware scaling mechanism in DataFlower.

We can also observe that memory usage reduction is even higher than the reduction of response latency. DataFlower reduces container memory usage by 19.1% to 69.3%, and by 7.4% to 64.1% compared with FaaSFlow and SONIC, respectively. This is mainly because the response latency reduction may have multiple effects on memory reduction for parallel processing, such as map-reduce logic. Each branch container can benefit from memory usage reduction, though the critical execution path can only reflect the end-to-end tail latency.

Figure 11 shows the serving throughput (requests-per-minute, rpm) of benchmarks when different numbers of clients generate requests synchronously. As shown, DataFlower increases the achievable peak throughputs of the benchmarks by 1.03X to 3.8X compared with FaaSFlow, and by 1.29X to 2.42X, compared with SONIC. The throughput is saturated as either CPU or network bandwidth



(a) Pressure-aware for *img* (b) Pressure-aware for *vid* (c) Pressure-aware for *svd* (d) Pressure-aware for *wc*

Figure 12: The achieved throughput of the benchmarks in DataFlower and DataFlower-Non-aware.

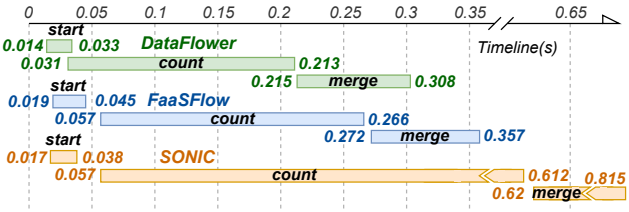
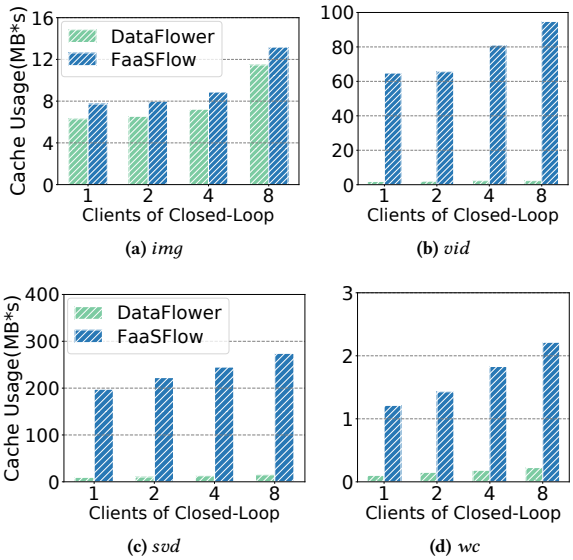


Figure 13: The function triggering timeline of benchmark *wc* with DataFlower, FaaSFlow, and SONIC.



(a) *img* (b) *vid* (c) *svd* (d) *wc*

Figure 14: Average memory usage on the host when caching the intermediate data with DataFlower and FaaSFlow.

becomes the bottleneck when the number of clients increases. We can also observe that *svd* fails with SONIC when there are 20 or more load generation clients in the closed-loops. This is because the application-aware data passing in SONIC can only optimize the data transfer of a single workflow invocation, but cannot recognize the transmission bottleneck of scaling containers for parallel workflow invocations.

DataFlower increases the peak throughputs of serverless workflows because its FLU-DLU abstraction enables the computation-communication overlap.

8.3 Effectiveness of Pressure-aware Scaling

In this experiment, we compare DataFlower with DataFlower-Non-aware, a variant of DataFlower that disables the pressure-aware function scaling. Figure 12 shows their performance with the synchronous load pattern. The achieved throughput drops significantly with DataFlower-Non-aware.

In the four benchmarks, DataFlower and DataFlower-Non-aware perform similar for *img*. This is mainly because the intermediate data between functions in *img* is small. A DLU can always transfer the output of a function to the destination without obvious queuing. On the contrary, for data-intensive functions in *vid*, *svd* and *wc*, the data transfer time of DLU is much longer than the computation time of FLU. Without the pressure-aware function scaling in DataFlower-Non-aware, user requests queue at DLUs. The peak throughputs of these benchmarks are constrained by the data transfer bottleneck.

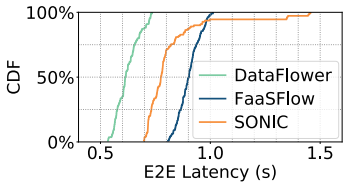
We can also observe that DataFlower-Non-aware achieves relatively high throughput when there are 16 and 32 closed-loop synchronous load clients for *vid*. This is because FLUs in current containers are insufficient to serve the increasing workload, and the serverless platform automatically scales out containers. This partially alleviates the queuing on the DLUs.

The pressure-aware function scaling is able to notice the queuing on the DLUs and scale out the containers, even if the containers are enough in terms of computation ability.

8.4 Early Triggering and Data Caching

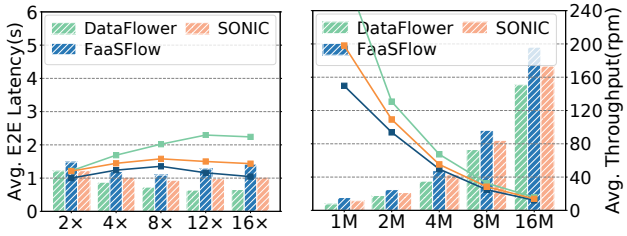
In this experiment, in order to eliminate the impact of network communication on the function triggering, we force all functions of a benchmark to run on a single node. In this way, both DataFlower and FaaSFlow communicate via local memory.

Figure 13 shows the execution timeline of the functions *start*, *count*, and *merge* in the benchmark *wc*. Other benchmarks show a similar result. The function *count* is triggered before the function *start* completes, and *merge* is triggered 2*ms* later once *count* completes with DataFlower. On the contrary, *count* and *merge* are triggered 15*ms* and 6*ms* later than their predecessor completion in FaaSFlow. The late triggering is due to the control-flow paradigm, although with FaaSFlow the intermediate data is also transferred through local memory. On the other hand, functions are triggered much later with SONIC because the function state still has to travel through local VM memory rather than shared memory.



	σ
DataFlower	0.053
FaaSFlow	0.050
SONIC	0.155

Figure 15: Response latency CDF and Standard Deviation (σ)



(a) Fan-out and Fan-in branches **(b) Input Datasize**

Figure 16: The average response latency and throughput of *wc* with different fan-out branches and input data size.

We also evaluate the effectiveness of the intermediate data management technique in the host-container collaborative communication mechanism of DataFlower. Figure 14 shows the average memory usage for caching the intermediate data on the host node per request with DataFlower and FaaSFlow at different loads. Compared to FaaSFlow, DataFlower reduces 19.1%, 90.2%, 94.9% and 97.5% of memory for caching the intermediate data in four benchmarks. FaaSFlow uses large memory on the host node for the intermediate data, because it ignores the lifetime of the intermediate data with the control-flow paradigm. On the contrary, DataFlower recycles the memory space through proactive release and passive expire.

8.5 Efficiency in Handling Bursty Load

Given that serverless is more geared towards bursty workloads, this experiment evaluates the performance of DataFlower in the bursty case. In the experiment, we suddenly increase the load of the benchmark *wc* (from 10 rpm to 100 rpm) using asynchronous invocations. Other benchmarks show similar results. For the 110 user requests invoked in the two minutes, Figure 15 shows the cumulative distribution and the standard deviation (σ) of their latencies respectively.

As observed, FaaSFlow and DataFlower handle bursty workloads more efficiently than SONIC. Moreover, the benchmarks have lower average and 99%-ile latencies with DataFlower, compared with others. DataFlower performs better, because it increases the resource utilization by overlapping the computation (FLU) and communication (DLU). In this way, each function container can execute more requests in a given time, and fewer containers are required to scale out for handling the bursty load. Scale-out time is reduced and the overhead of creating containers with DataFlower is also reduced.

8.6 Adaptiveness to Various Workflows

While serverless workflows may have different structures and inputs, this experiment uses *wc* as an example to show the adaptiveness of DataFlower for various workflows. The DLU of the predecessor will send the data to child functions through different pipe connectors in the FIFO manner. Figure 16 shows the average response latency and the processing throughput of *wc* with different fan-out and fan-in branches and input data sizes. The input data size is fixed to be 4MB in Figure 16(a). The number of fan-out branches is 4 in Figure 16(b).

Observed from Figure 16(a), DataFlower results in much lower latency and higher throughput of the benchmark with all the branch numbers. DataFlower increases the peak throughput by 69.3% and 58.8% compared with FaaSFlow and SONIC, respectively. We can also find that DataFlower performs better with higher branch numbers. When the fan-out branch increases, DataFlower can execute more functions in parallel. The data-availability driven per-function triggering in DataFlower takes advantage of the parallelism, the data is processed earlier and faster. On the contrary, the sequential function triggering of the control-flow paradigm in FaaSFlow and SONIC fails to take advantage of the parallelism.

As shown in Figure 16(b), the latency increases, and the throughput drops when the input data size increases with FaaSFlow, SONIC, and DataFlower. It is reasonable because a larger input data size means higher workload in a single request of *wc*. With small input data (e.g. 1MB), DataFlower increases the throughput by 91.8% and 44.9% compared with FaaSFlow and SONIC. With larger input data (e.g. 16MB), DataFlower increases the throughput by 29.5% and 14.5% compared with FaaSFlow and SONIC. The improvement drops when the input data size increases. With the given number of fan-out and fan-in branches, the workload of a function increases when the input data is larger. When the input data is large, the computation resources (i.e. CPU time) become the performance bottleneck, and the performance gain from the data-flow paradigm in DataFlower becomes smaller.

With the data-flow paradigm, it is beneficial to further reduce the function granularity. While the data-flow paradigm is capable of exploring parallelism, the fine-grained functions can run in parallel to explore the spare resources. On the contrary, the control-flow paradigm cannot support high parallelism and is therefore suitable for coarse-grained functions.

8.7 Impact of Scaling Up Containers

When the number of functions that may run in parallel is small (e.g., the number of fan-out branches is small in *wc*), the response latency can not be reduced by increasing the containers (referred to be scale-out). Increasing the resources allocated to each container may improve the performance in this case (referred to be scale-up).

Figure 17 shows the response latency and the processing throughput of *wc* with the 4MB input and 8 fan-out branches, when we scale up the container's resource. The *x*-axis shows the memory specification allocated to a container. The CPU time and network bandwidth increase linearly with the memory allocation. As observed, the processing throughput increases linearly when we scale up the containers with DataFlower and SONIC. Compared with FaaSFlow and SONIC, DataFlower increases the throughput of *wc* by 148.4% and 11.1% with large containers (640MB memory).

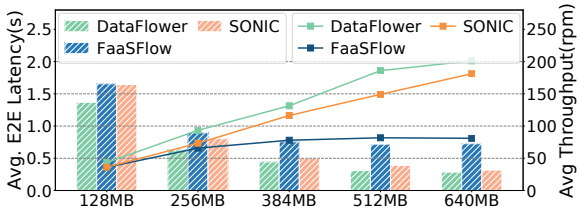


Figure 17: The response latency and throughput of *wc* when scaling up the container's resources.

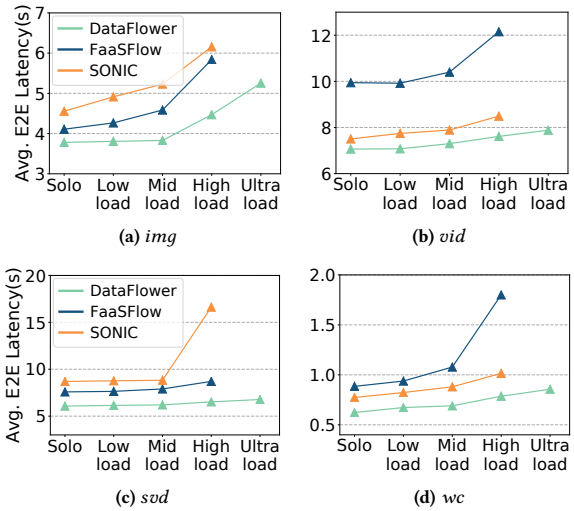


Figure 18: The response latency of the co-located benchmarks in DataFlower, FaaSFlow and SONIC.

When the container scales up, the performance of both function computation and the data transmission increase with DataFlower and SONIC, as they use direct data passing. However, with FaaSFlow, the performance bottleneck occurs in accessing the backend storage when the functions store intermediate data in parallel. In this case, FaaSFlow often cannot benefit from scaling up the containers.

By supporting container scale-out and scale-up, serverless platforms have the potential to support higher loads with the data-flow paradigm. The workflows with many fine-grained functions that can run in parallel tend to benefit more from scaling-out, while the workflows with heavy functions also benefit from scaling-up.

8.8 Colocating Multiple Workflows

In the production deployment, multiple serverless workflows may co-run on the same set of nodes. In this experiment, we co-run all four benchmarks on the three worker nodes to evaluate DataFlower in the co-location scenario. Figure 18 shows the latencies of the co-located benchmarks. In the figure, "Solo" shows the benchmark's latencies when it runs alone, "Low load" indicates when all four benchmarks run with low asynchronous workloads, "Ultra load" shows the latencies when the loads of the benchmarks is maximized, in order to simulate the highly contended environments.

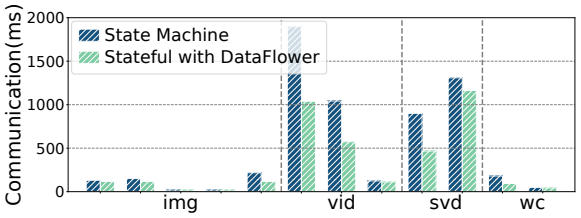


Figure 19: Communication overhead with the state machine and streaming-based functions on DataFlower.

As observed, the benchmarks have the shortest response latency with DataFlower in all the co-location cases. The benchmarks fail at "Ultra load" with FaaSFlow and SONIC. This is because they lack the efficient container scaling policy on heavily overtaxed machines. We can find that none of the benchmarks suffers from more than 2X performance degradation when the workload is high with DataFlower. The performance interference between workflows is limited, as the CPU resources are explicitly allocated to each container, and both FLU and DLU in a container only use its dedicated CPU, memory, and network resources. Meanwhile, cloud providers are also working on encapsulating functions with MircoVM like Firecracker [23], RunD [51], and Kata-container [15], to eliminate turbulent performance due to interference.

8.9 Integrating with Stateful Functions

DataFlower can also be extended and applied to workflows using stateful functions. For example, AWS Step Functions [6] implements stateful serverless workflows by using "state machine" to schedule tasks [18]. Specifically, when a Lambda function λ_a completes, it returns result and the state machine stores the output as a context object. Then the state machine advances to the next state and calls the next Lambda function λ_b by passing the data from the context object. Since the built-in state machine in AWS Step Functions has a quota of 256KB for stateful data [16], we simulate the situation where benchmarks have been deployed in a stateful manner with a state machine running on AWS EC2 to cache unlimited data.

Figure 19 shows the data transfer time of the functions in each benchmark with the above stateful function implementation. "Stateful with DataFlower" shows the case that we use the streaming-based functions in DataFlower to replace the traditional state machine based Step Functions. As observed, the pipe connector in DataFlower reduces up to 47.6% of the function-to-function data transfer time.

Moreover, the compute-communication overlap (aka. FLU-DLU) and data availability-based early triggering techniques are not affected by whether the functions are stateless or stateful. The experiments in Section 8.4 have already showed their effectiveness. Therefore, DataFlower scheme can also improve the performance of serverless workflows based on stateful functions.

9 RELATED WORKS

Data passing in serverless workflows. SONIC [56] improved the transmission efficiency of different fan-out branches by introducing an application-aware data-passing mechanism between functions

in the workflow. Diverse data-passing and storing solutions for serverless functions [44, 49, 62, 74] further optimized the data I/O bottlenecks. Though the data transfer overhead can be partially reduced, the late function triggering due to the control-flow paradigm still exists.

Task-based serverless workflows. Most serverless designs adopt the centralized workflow orchestrator to maintain the execution state, and then use the control-flow to assign tasks [26, 39, 45, 59, 70]. To reduce the scheduling overhead, NightCore [45] used lightweight IPC for functions co-located on the same node. For cross-node function triggering, it relies on the gateway to collect the execution state and assign function tasks. FaaSFlow [54] proposed a decentralized scheduling pattern called WorkerSP to reduce the cross-node scheduling overhead. Other related works like function bundling of a workflow [24, 45, 50, 52, 57, 58, 69] explored the benefits when hosting multiple functions of an application into the dedicated sandbox. They still focused on the traditional control-flow based serverless frameworks, while DataFlower is a data-flow paradigm-based serverless scheme whose function triggering is based on the data-availability.

Dataflow-based workflows. Pegasus [33] is a DAG based workflow system, its trigger logic depends on the completion state of each task. Spark [76] focuses on big data processing and offers data-oriented optimizations such as data-locality aware scheduling. Other similar systems such as Flink [2] and TensorFlow [10] are also dataflow-based workflow systems. However, they hold the assumptions and implementations that 1) tasks can communicate with each other through connections that are both addressable and visible to programmers, 2) executor containers typically have coarse-grained resource allocation to optimize the context accessing, and 3) executor containers with data-denepdence are started in advance rather than based on an event-driven pattern. These distinctions make it difficult to adapt above data-flow systems directly into native serverless.

To enable the data-flow paradigm, SWEEP [46] implemented a workflow management layer on top of serverless frameworks for definition and scalable execution of data processing pipelines. AFCL [64] proposed a high-level programming abstraction to express control-flow and data-flow when constructing a serverless workflow, based on the AWS Lambda backend. However, without co-optimization with the underlying serverless architecture and a customized container scheduling strategy, they cannot take full advantage of the data-flow paradigm like DataFlower in serverless.

10 CONCLUSIONS AND FUTURE WORK

We propose DataFlower, a scheme that achieves the data-flow paradigm for serverless workflows. Specifically, the container that runs a function is abstracted into a FLU and a DLU. The FLU processes the function computation, and the DLU handles all the communication. In this way, the computation and communication overlap with each other, improving the processing throughput. Moreover, a host-container collaborative communication mechanism is proposed to minimize the communication overhead. Experimental results show that DataFlower greatly reduces the response latency, increases the supported throughput, and reduces resource usage.

The data-flow paradigm can also provide an alternative way to prewarm containers based on the data dependencies and availability. With the prior knowledge of the data dependencies, we are designing a policy to warm up a container for a function based on the data-availability instead of predicting function execution patterns in the future work.

ACKNOWLEDGMENT

We sincerely thank our anonymous reviewers, for their helpful comments and suggestions. This work is partially sponsored by the National Natural Science Foundation of China (62232011, 62022057, 61832006), and Shanghai international science and technology collaboration project (21510713600). Quan Chen and Minyi Guo are the corresponding authors.

REFERENCES

[1] 2022. Alibaba Serverless Workflow: Visualization, free orchestration, and Coordination of Stateful Application Scenarios. https://www.alibabacloud.com/product/serverless-workflow.
[2] 2022. Apache Flink: Stateful Computations over Data Streams. https://flink.apache.org/.
[3] 2022. Apache Kafka is an open-source distributed event streaming platform. https://kafka.apache.org/.
[4] 2022. AWS Lambda execution context - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html.
[5] 2022. AWS Lambda: Run code without thinking about servers or clusters. https://aws.amazon.com/lambda/.
[6] 2022. AWS Step Functions: assemble functions into business-critical applications. https://aws.amazon.com/step-functions/.
[7] 2022. Azure Functions: Execute event-driven serverless code functions with an end-to-end development experience. https://azure.microsoft.com/en-us/products/functions/.
[8] 2022. CNCF Serverless Whitepaper. https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview.
[9] 2022. CouchDB. https://couchdb.apache.org/.
[10] 2022. Create production-grade machine learning models with TensorFlow. https://tensorflow.google.cn/.
[11] 2022. Durable Functions is an extension of Azure Functions that lets you write stateful functions in a serverless compute environment. https://docs.microsoft.com/en-us/azure/azure-functions/durable/.
[12] 2022. Fission Workflows: Fast, reliable and lightweight function composition for serverless functions. https://github.com/fission/fission-workflows.
[13] 2022. Function Compute: A secure and stable, elastically scaled, pay-as-you-go, serverless computing platform. https://www.alibabacloud.com/product/function-compute.
[14] 2022. How to Handle Errors in Serverless Applications with AWS Step Functions. https://aws.amazon.com/getting-started/hands-on/handle-serverless-application-errors-step-functions-lambda/.
[15] 2022. Kata Containers - Open Source Container Runtime Software. https://katacontainers.io/.
[16] 2022. Maximum input or output size for a task, state, or execution. https://docs.aws.amazon.com/step-functions/latest/dg/limits-overview.html.
[17] 2022. OpenWhisk: Serverless functions platform for building cloud applications. https://github.com/apache/openwhisk.
[18] 2022. The state machine used in AWS Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-state-machine-data.html.
[19] 2022. Synchronous and asynchronous invocations in OpenFaaS. https://docs.openfaas.com/reference/async/.
[20] 2022. Tutorial for detecting and bluring offensive images using Cloud Functions. https://cloud.google.com/functions/docs/tutorials/imagemagick.
[21] 2022. Use FFmpeg in Function Compute to process audio and video files in Function Compute. https://www.alibabacloud.com/help/doc-detail/146712.htm?spm=a2c63.l28256.b99.313.5c293c94dPLJV1.
[22] Mainak Adhikari, Tarachand Amgoth, and Satish Narayana Srirama. 2019. A Survey on Scheduling Strategies for Workflows in Cloud Environment and Emerging Trends. *ACM Comput. Surv.* 52, 4 (2019), 68:1–68:36. https://doi.org/10.1145/3325097
[23] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA,*

February 25-27, 2020, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[24] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[25] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 263–274. https://doi.org/10.1145/3267809.3267815

[26] Bartosz Balis. 2016. HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows. *Future Gener. Comput. Syst.* 55 (2016), 147–162. https://doi.org/10.1016/j.future.2015.08.015

[27] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful Serverless Computing with Crucial. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 39 (mar 2022), 38 pages. https://doi.org/10.1145/3490386

[28] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita R. Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 153–167. https://doi.org/10.1145/3472883.3486992

[29] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 645–650. https://www.usenix.org/conference/atc18/presentation/boucher

[30] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.* 15, 8 (jun 2022), 1591–1604. https://doi.org/10.14778/3529337.3529344

[31] João Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy H. Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 13–24. https://doi.org/10.1145/3357223.3362711

[32] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 1–15. https://doi.org/10.1145/3419111.3421286

[33] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and R. Kent Wenger. 2015. Pegasus, a workflow management system for science automation. *Future Gener. Comput. Syst.* 46 (2015), 17–35. https://doi.org/10.1016/j.future.2014.10.008

[34] Jack B. Dennis and David Misunas. 1974. A Preliminary Architecture for a Basic Data Flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture, Houston, TX, USA, December 1974*, Willis K. King and Oscar N. Garcia (Eds.). ACM, 126–132. https://doi.org/10.1145/642089.642111

[35] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 467–481. https://doi.org/10.1145/3373376.3378512

[36] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: lambdas on a diet. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 45–59. https://doi.org/10.1145/3419111.3421297

[37] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[38] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 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 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[39] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 386–400. https://doi.org/10.1145/3445814.3446757

[40] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[41] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. *login Usenix Mag.* 41, 4 (2016). https://www.usenix.org/publications/login/winter2016/hendrickson

[42] Jayantha A. Herath, Yoshinori Yamaguchi, Nobuo Saito, and Toshitsugu Yuba. 1988. Dataflow Computing Models, Languages, and Machines for Intelligence Computations. *IEEE Trans. Software Eng.* 14, 12 (1988), 1805–1828. https://doi.org/10.1109/32.9065

[43] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 149:1–149:26. https://doi.org/10.1145/3360575

[44] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 691–707. https://doi.org/10.1145/3477132.3483541

[45] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 152–166. https://doi.org/10.1145/3445814.3446701

[46] Aji John, Kristiina Ausmees, Kathleen Muenzen, Catherine Kuhn, and Amanda Tan. 2019. SWEEP: Accelerating Scientific Research Through Scalable Serverless Workflows. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2019, Companion Volume, Auckland, New Zealand, December 2-5, 2019*, Kenneth Johnson, Josef Spillner, Dalibor Klusácek, and Ashiq Anjum (Eds.). ACM, 43–50. https://doi.org/10.1145/3368235.3368839

[47] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[48] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[49] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[50] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[51] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 53–68. https://www.usenix.org/conference/atc22/presentation/li-zijun-rund

[52] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[53] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. 2021. The Serverless Computing Survey: A Technical Primer for Design Architecture. *CoRR* abs/2112.12921 (2021). arXiv:2112.12921 https://arxiv.org/abs/2112.12921

[54] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 782–796. https://doi.org/10.1145/3503222.3507717

[55] Changyuan Lin and Hamzeh Khazaei. 2021. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 615–632. https://doi.org/10.1109/TPDS.2020.3028841

[56] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[57] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[58] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2 (2022), 26:1–26:28. https://doi.org/10.1145/3530892

[59] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2020. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Gener. Comput. Syst.* 110 (2020), 502–514. https://doi.org/10.1016/j.future.2017.10.029

[60] M. Garrett McGrath and Paul R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino (Eds.). IEEE Computer Society, 405–410. https://doi.org/10.1109/ICDCSW.2017.36

[61] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 115–130. https://doi.org/10.1145/3318464.3389758

[62] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[63] Cristian Ramon-Cortes, Francesc Lordan, Jorge Ejarque, and Rosa M. Badia. 2020. A programming model for Hybrid Workflows: Combining task-based workflows and dataflows all-in-one. *Future Gener. Comput. Syst.* 113 (2020), 281–297. https://doi.org/10.1016/j.future.2020.07.007

[64] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. 2021. AFCL: An Abstract Function Choreography Language for serverless workflow specification. *Future Gener. Comput. Syst.* 114 (2021), 368–382. https://doi.org/10.1016/j.future.2020.08.012

[65] Salvatore Sanfilippo. 2022. Redis: Remote Dictionary Server. https://redis.io/.

[66] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 1063–1075. https://doi.org/10.1145/3352460.3358296

[67] Jiuchen Shi, Jiawen Wang, Kaihua Fu, Quan Chen, Deze Zeng, and Minyi Guo. 2022. QoS-awareness of Microservices with Excessive Loads via Inter-Datacenter Scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*. IEEE, 324–334. https://doi.org/10.1109/IPDPS53621.2022.00039

[68] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. 2023. Nodens: Enabling Resource Efficient and Fast QoS Recovery of Dynamic Microservice Applications in Datacenters. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 403–417. https://www.usenix.org/conference/atc23/presentation/shi

[69] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[70] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: enabling quality-of-service in serverless computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 311–327. https://doi.org/10.1145/3419111.3421306

[71] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 559–572. https://doi.org/10.1145/3445814.3446714

[72] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 39:1–39:16. https://doi.org/10.1145/3302424.3303978

[73] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[74] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2021. Autoscaling tiered cloud storage in Anna. *VLDB J.* 30, 1 (2021), 25–43. https://doi.org/10.1007/s00778-020-00632-7

[75] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2021. Restructuring Serverless Computing with Data-Centric Function Orchestration. *CoRR* abs/2109.13492 (2021). arXiv:2109.13492 https://arxiv.org/abs/2109.13492

[76] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, Erich M. Nahum and Dongyan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[77] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 659–670. https://doi.org/10.1109/ICDE.2017.119

[78] Ge Zheng and Yang Peng. 2019. GlobalFlow: A Cross-Region Orchestration Service for Serverless Computing Services. In *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama (Eds.). IEEE, 508–510. https://doi.org/10.1109/CLOUD.2019.00093

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact includes the prototype implementation (already built into FaaSFlow) of the data-flow paradigm for serverless workflow orchestration, the four workflow benchmarks, and some experimental workflow scripts to evaluate DataFlower.

A.2 Artifact check-list (meta-information)

- **Algorithm:** A congestion control algorithm that is used in the data logic unit (DLU) to aware and relieve the pressure of data transfer.
- **Program:** DataFlower, Real-world application benchmarks, Docker runtime, CouchDB, Kafka, and Redis.
- **Run-time environment:** Ubuntu 20.04 with Docker installed. Detailed software libraries (including docker runtime, CouchDB, Kafka, Redis, and python packages) are all listed and scripted in the artifact.
- **Hardware:** The recommended hardware requirements is {Cores: 16, DRAM: 64GB, Disk: 200GB SSD}.
- **Execution:** To verify the correctness of DataFlower, we provide the corresponding scripts for part of the evaluation (Section 8.2 and Section 8.8). The scripts will send requests and collect the metrics.
- **Metrics:** The end-to-end latency, memory usage, and throughput are collected in Section 8.2. The end-to-end latency is also collected in Section 8.8.
- **Output:** json files.
- **Experiments:** Python scripts.

- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 4 hours
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.10052369

A.3 Description

A.3.1 How to access. The source code of DataFlower are available and maintained on GitHub https://github.com/lzjzx1122/FaaSFlow.

A.3.2 Hardware dependencies. DataFlower requires at least three nodes (one gateway node, one storage node, and one or more worker nodes). For the gateway node and the storage node, the minimal hardware requirements is {Cores: 8, DRAM: 16GB, Disk: 200GB SSD}. For each worker node, the minimal hardware requirements is {Cores: 16, DRAM: 64GB, Disk: 200GB SSD}.

A.3.3 Software dependencies. The artifact requires experiment VMs running Ubuntu 20.04 with Docker installed. We provide an installation script to prepare the software environment (including docker runtime, CouchDB, Kafka, Redis, and python packages) for the worker node, the storage node, and the gateway node.

A.4 Installation

To conduct our experiment, we recommend building an 5-nodes cluster, where 3 nodes are used for performing the function execution (worker nodes), 1 node is used for remote storage and logging (storage node), and 1 node is used for requests generating (gateway node). The `README.md` of the artifact provides more details on conducting experiments of DataFlower.

IP Configurations for each node. *Note: all of the configurations MUST be applied on each node.*

First, reset `WORKER_ADDRS` as the worker nodes IPs in `config/config.py`. The elements number of `WORKER_ADDRS` represents whether the evaluation will be done within a single worker node or among multiple worker nodes. Currently we support the worker number less or equal than three.

Also, reset `GATEWAY_IP` as the gateway node IP, `COUCHDB_IP` as the storage node IP, and `KAFKA_IP` as storage node IP. These settings can be found in `config/config.py`.

Then, reset `COUCHDB_URL` as the storage node IP, `KAFKA_URL` as the storage node IP in `src/container/container_config.py`.

Lastly, reset `KAFKA_ADVERTISED_LISTENERS: PLAINTEXT` as the storage node IP in `scripts/kafka/docker-compose.yml`.

Setting up the worker node. Run `scripts/worker_setup.bash` on each worker node. This script installs docker, Redis, and some python packages, and builds docker images from 4 benchmarks.

Setting up the storage node and the gateway node. First, run `scripts/db_setup.bash` on the storage node and then run `scripts/gateway_setup.bash` on the gateway node. These scripts install docker, Kafka, CouchDB, some python packages.

A.5 Experiment workflow

The entry point of each experiment is the `test/*test.py` script. Detailed instructions on how to start up the proxy on each node and trigger the test scripts are introduced in the `README.md` of our artifact. It first generates the invocations and sends them to the gateway node. Then it collects the corresponding detailed metrics in a `.json` file.

A.6 Evaluation and expected results

The detailed evaluation results will be stored in `test/result/`, while the expected results are in `test/expected_results`. Meanwhile, the `test/*test.py` scripts will also print the key results in the terminal during the experiment.

A.7 Experiment customization

To run DataFlower under one worker node, just specify one IP addr in `WORKER_ADDRS` in `config/config.py`.

To run DataFlower under more than three worker nodes, besides reset `WORKER_ADDRS` in `config/config.py`, the ip route table of each function of each benchmark should also be assigned in `*_sp_ip_idx` in `src/workflow_manager/gateway.py`.