Floki: A Proactive Data Forwarding System for Direct Inter-Function Communication for Serverless Workflows

Anna Maria Nestorov
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Catalunya, Spain
anna.nestorov@bsc.es

Josep Lluís Berral
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Catalunya, Spain
josep.ll.berral@upc.edu

Claudia Misale
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
c.misale@ibm.com

Chen Wang
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
chen.wang1@ibm.com

David Carrera
Barcelona Supercomputing Center
Barcelona, Catalunya, Spain
david.carrera@bsc.es

Alaa Youssef
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
asyousse@us.ibm.com

ABSTRACT

Serverless computing emerges as an architecture choice to build and run containerized data-intensive pipelines. It leaves the tedious work of infrastructure management and operations to the cloud provider, allowing developers to focus on their core business logic, decomposing their jobs into small containerized functions. To increase platform scalability and flexibility, providers take advantage of hardware disaggregation and require inter-function communication to go through shared object storage. Despite data persistence and recovery advantages, object storage is expensive in terms of performance and resources when dealing with data-intensive workloads. In this paper, we present Floki, a data forwarding system for direct and inter-function data exchange proactively enabling point-to-point communication between pipeline producer-consumer pairs of containerized functions through fixed-size memory buffers, pipes, and sockets. Compared with state-of-practice object storage, Floki shows up to 74.95× of end-to-end time performance increase, reducing the largest data sharing time from 12.55 to 4.33 minutes, while requiring up to 50,738× fewer disk resources, with up to roughly 96GB space release.

CCS CONCEPTS

• **Information systems** → **Data management systems**.

KEYWORDS

Containers, Communication, Serverless, Orchestration, Kubernetes

1 INTRODUCTION

Serverless computing has received a significant uptick in attention over the last few years, both in academia and industry. In the serverless paradigm, the operational concerns of the infrastructure are fully managed by the cloud provider, relieving the user from challenging decisions, e.g., instance types, cluster size, and load balancing strategy. Thus, the serverless paradigm lets more users approach the cloud, allowing them to be completely focused on their domain of expertise. Serverless attracts many users also for its fundamental principles, such as the 'pay-per-use' cost model and transparent auto-scaling based on incoming requests.

Among its limitations [4, 6, 10], no network addressability and the lack of efficient data sharing between functions prevent a vast number of data-intensive workloads to benefit from serverless. Not supporting direct inter-function communication, major serverless platforms take disaggregation to an extreme, imposing functions to exchange data only through shared object storage. In the context of data-intensive workloads, represented as Directed Acyclic Graphs (DAGs) and characterized by a considerable amount of intermediate data transfers, shared object storage becomes a bottleneck for efficient inter-function communication due to its high-latency access. Indeed, as demonstrated in [16], directly using a serverless platform for data-intensive workloads leads to highly inefficient executions. The slow data transfers between functions make the CloudSort benchmark to be up to 500× slower when executed on AWS Lambda with S3 instead of on a cluster of Virtual Machines (VMs). Recent studies tackle this problem by implementing optimized exchange operators [13, 15], using multi-tier storage combining slow with fast storage or solely remote in-memory storage [7, 8, 16], exploiting per-node caches [2, 19], co-locating functions on a single container [1, 5, 9, 18], handling external storage on long-running VMs [3, 22], or circumventing the network constraints [21]. However, these methods either use domain-specific optimizations, require two copies of data over the network, are not fully transparent to the user, break the advantage of fine-grained scaling, or use non-serverless components.

In this paper, we present *Floki*, a system enabling direct inter-function communication in Kubernetes-based environments by proactively forwarding data based on the workflow. It creates point-to-point data channels exploiting conventional pipes and TCP sockets, for intra-node and inter-node data transmission, allowing data to be transferred directly from producer to consumer functions in a fully transparent fashion, minimizing data copying over the network. Floki offers workflow-oriented data communication, increasing performance while minimizing resource requirements without imposing any constraint on function placement. Specifically, its flexibility and observability allow creating data channels adapting to the specific function scheduling of the underlying orchestration framework.

The main contributions of this paper are:

- A proactive workflow-based data forwarding system enabling point-to-point data transfers without additional overheads, such as state-of-practice storage overheads.
- The design of a full communication stack, allowing non-colocated functions to share data as they are hosted on the same node through local read-write operations.

The final publication is available at ACM via http://dx.doi.org/10.1145/3565384.3565890

- An in-memory mechanism for transferring volatile data, capable of dealing with arbitrary intermediate data sizes efficiently and scalable on the data volume.
- A benchmark of Floki on the principal communication patterns in distributed systems, i.e., one-to-one, fan-out, fan-in, and all-to-all, with data transfers between 1MB and 16GB.

We envision Floki to be leveraged by container-based platforms and users for high-performance volatile intermediate data exchange, as an alternative solution for message passing.

2 RELATED WORK

The serverless paradigm takes advantage of hardware disaggregation, using the data center as a pool of independent resources connected through high-speed networks. While there are several available container orchestration frameworks, Kubernetes has become the leading platform and *de-facto* cross-cloud standard for automatic management of containerized applications. Knative introduces serverless capabilities for Kubernetes clusters, managing stateless services for deployment and autoscaling. However, in these platforms communication latency becomes a bottleneck for data-intensive applications or when parallelizing, making data transfers through shared object storage directly proportional to the scale factor. Also, although many streaming services exist, none represents a good match for intermediate data communication between non-colocated functions. For example, Apache Kafka requires storing redundant copies of a large amount of data when working with large data sets, by introducing an additional full-stack service. Finally, KubeFlow MPI ports MPI on Kubernetes within a specific use case: making it easy to run allreduce-style distributed machine learning training. However, its data communication is mainly driven by the user.

Efficient intermediate data sharing between functions represents a key challenge for chained function execution, especially when dealing with data-intensive workloads [3, 4]. Different approaches have been proposed to optimize data exchange in serverless workflows, e.g., Lambada [13] and Starling [15], however focusing solely on database analytics and using domain-specific optimizations. Technologies like Locus [16] and Pocket [8] leverage in-memory storage for analytics and intermediate data sharing, but require resource demand information from the user at submission time. In contrast, our solution does not need any information on resource demand. Also, as shown in [7, 16, 17], intermediate data sizes can consistently vary during the workload execution, resulting in the well-understood problem of potential performance degradation and/or resource underutilization [11, 20]. All these works involve indirect communication, demanding two serial data copies over the network in the critical path: one from producer function to shared storage and one from shared storage to consumer function. Contrarily, Floki always requires only one data copy for each inter-function communication (from source to destination node).

Works like Crucial [2], Cloudburst [19], OFC [14], SONIC [12], and SAND [1] focus on inter-function data sharing, disaggregation and physical co-location, or data locality exploitation. However, they need static provisioning of per-node cache resources or focus on small data sizes. Faasm [18], Nightcore [5], and Faastlane [9] co-locate workload functions on a single container to minimize data sharing latency thanks to shared memory access, requiring memory over-provisioning to ensure containers run multiple functions and extra services for concurrent executions during peak usage.

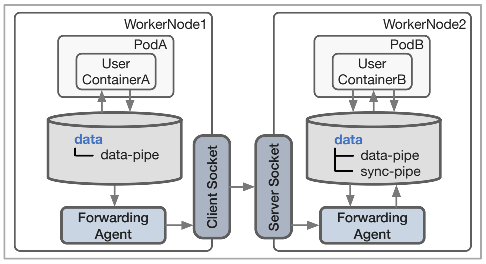

Figure 1: Floki's architecture.

Finally, Boxer [21] improves Lambada by enabling inter-function direct communication using TCP connections, deploying their subsystem alongside each function. Floki establishes TCP connections on the host namespace; therefore, it does not require deploying additional connection-specific components in the serverless platform.

3 FLOKI ARCHITECTURE

In this section, we tackle the shared object storage bottleneck problem by presenting Floki, a system that proactively enables faster point-to-point data sharing by exploiting local resources and TCP socket connections. The system requires two inputs: First, the DAG describing the workflow where nodes represent functions and arcs represent data dependencies between functions; Second, the mapping between functions and cluster nodes. In the current version of Floki, we assume the two inputs are given, and the user containers read/write data sequentially. We further discuss these assumptions in Section 5. In Floki, we achieve the following design goals: 1) Deal with arbitrary complex data structures, 2) Proactive data transfer between functions, 3) Fast and direct inter-function communication, 4) No constraint on functions placement. We highlight how we achieve these specific design goals in the remaining of this section.

Floki's architecture transmits data in a fully volatile manner relying on pipes and TCP sockets for intra-node and inter-node data transmission, respectively. In Floki functions run concurrently, while in the naïve shared object storage communication functions run sequentially based on data dependencies. Data is transmitted and stored as byte arrays, allowing Floki to deal with arbitrary complex data structures independently from the programming language (Goal 1). Floki's architectures solve the centralized storage bottleneck by offering direct communication between functions, where data exchanges are managed on a producer-consumer functions pair level, minimizing data copying over the network. Direct communication between functions implies the following advantages. First, the number of concurrent read/write operations is reduced as resources are shared among a lower number of functions, i.e., the ones co-placed on a given node, or of exclusive use. Second, the I/O and CPU usage are lower. Finally, thanks to multi-threading, a producer function data can be sent in parallel to multiple consumer functions, and a consumer function can receive concurrently multiple data.

Floki's architectures consider five key components: the *data-* and *sync-pipe*, the *client* and *server sockets*, and the *forwarding agent*. To proactively transfer data between producer-consumer function pairs, based on the specific workload DAG and function-node mappings,

Algorithm 1 Floki's forwarding agent algorithm.

```
 1: procedure RECVDATA($sSocket$,$listObjsToRecv$)
 2:     AquireLock(dataPipe)
 3:     for all objName ∈ listObjsToRecv do
 4:         dataSize = RecvAndWriteSize(sSocket, dataPipe)
 5:         RecvAndWriteData(sSocket, dataPipe, dataSize)
 6:     end for
 7:     ReleaseLock(dataPipe)
 8: end procedure
 9: procedure SENDDATA($cSocket$,$listObjsToSend$)
10:     for all objName ∈ listObjsToSend do
11:         if currentObjName == objName then
12:             SendDataSize(cSocket, sizeBuffer)
13:             for k = 1 to $\lceil outDataSize/packetSize \rceil$ do
14:                 SendDataPacket(cSocket, dataBuffer)
15:             end for
16:         end if
17:     end for
18: end procedure
19: procedure FORWARDINGAGENT( )
20:     producers = GetSocketsProducersNames(sSockets)
21:     SendOwnName(cSockets)
22:     WaitReady(syncPipe)
23:     for i = 1 to #sSockets do
24:         thsIn[i] = thread(RECVDATA, sSockets[i], listObjsToRecv)
25:     end for
26:     WaitThreadsEnd(thsIn)
27:     for j = 1 to #cSockets do
28:         thsOut[j] = thread(SENDDATA, cSockets[j], listObjsToSend)
29:     end for
30:     for all outObjName ∈ outObjsNames do
31:         currentObjName = outObjName
32:         outDataSize = ReadDataSize(dataPipe, sizeBuffer)
33:         for k = 1 to $\lceil outDataSize/packetSize \rceil$ do
34:             ReadDataPacket(dataPipe, dataBuffer)
35:         end for
36:     end for
37:     WaitThreadsEnd(thsOut)
38: end procedure
```

Floki creates all the necessary components and relative connections immediately after its submission (Goal 2).

Data- and sync-pipe: The two pipes, exposed on a local Persistent Volume (PV), allow to exchange data between the user container and the local *forwarding process*. While the *data-pipe* represents the data communication channel, the *sync-pipe* synchronizes Floki with the user container letting the *forwarding agent* to write on the *data-pipe* only when the user container is ready to receive[1].

Client and server sockets: These are the key components in charge of transmitting data between pairs of nodes. We choose to implement TCP sockets guaranteeing features such as error checking, ordered data delivery, and enabling uniquely identified connections between two endpoints, i.e., combining client and server sockets.

Forwarding agent: This component, instantiated into a process in the host namespace, represents Floki's architecture core component. At a high level, the primary purpose of this component is to drive inter-node communication, forwarding data directly from the producer to the consumer function (Goal 3). More precisely, its role is threefold. First, it creates and sets up the required TCP connections. Second, it supplies the necessary input data to the user container. Third, it forwards the data produced by the user container to the following functions in the chain. Since the *forwarding agent* mainly performs write/read memory buffers operations, we expect its overhead to be negligible. To proactively set up the communication infrastructure for the specific workflow, it internally stores the function name to which it refers and the ordered lists of data object names to receive/send for each of the previous/following functions in the workflow. In addition, it stores the mapping between the following functions in the workflow and the IP address of the nodes to which they have been scheduled to account for the underlying scheduler functions placement (Goal 4). Based on this information, it automatically derives the necessary number of *server* and *client socket* connections, i.e., $\#sSockets$ and $\#cSockets$.

Algorithm 1 shows the pseudo-code of the *forwarding agent*, whose top-function is represented by the FORWARDINGAGENT procedure. Once the $\#sSockets$ *server* and $\#cSockets$ *client socket* connections are opened and set up, the *forwarding agent* receives producer functions' names getting the correspondence with the *server socket* connections (line 20). Storing for each producer the list of data object names to receive allows the *forwarding agent* to know the number and the names of the data objects transmitted on each *server socket*. Then, the *forwarding agent* sends the related function name on all *client sockets* (line 21). Since the *forwarding agent* starts before the workflow is deployed, to respect the coordination of the pipe operations, the *forwarding agent* waits for the user container ready signal eventing that it is running and ready to read data from the *data-pipe* (line 22). Once received, the *forwarding agent* creates the $\#sSockets$ input threads (lines 23-25).

The input threads alternately write on the *data-pipe*, sending first the data size (line 4) and then the data content read in a packet-based fashion from the corresponding *server socket* (line 5). The input threads' *data-pipe* write operations follow the order declared in the stored list of data objects to receive. To handle *data-pipe* contention, the threads' write operations are synchronized by acquiring (line 2) and releasing (line 7) a lock. When all input threads finish (line 26), the output threads, in charge of sending the user container-produced data on the *client sockets*, are created (lines 27-29).

For each produced data object, the *forwarding agent* reads the data object size $outDataSize$ (line 24) and the data object content from the *data-pipe* (lines 32-35). To guarantee output threads access to both data object size and content, the *forwarding agent* read operations store them in $sizeBuffer$ and $dataBuffer$ shared memory buffers. Finally, each output thread sends the buffers on the *client socket* (lines 12-15) if the current received data object, i.e., $currentObjName$, belongs to its list of objects to send (line 11).

[1]Prevents the write operation from receiving a broken pipe signal when the read file descriptor referring to the pipe read end is not opened.

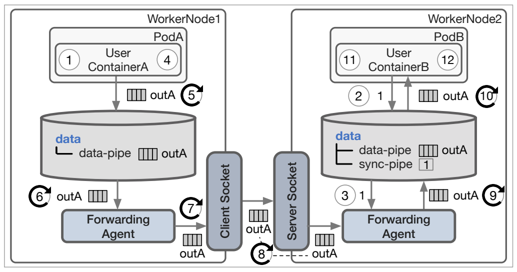

Figure 2: A step-by-step example of Floki.

Figure 2 illustrates how Floki works step-by-step with a simple example of a two functions workflow. The first function reads the workflow input stored in the object storage and creates the intermediate output *outA*. In contrast, the second function reads the intermediate data object *outA* and computes the workflow output *out*, saving it in the object storage. For data transferred in a packet-based fashion, in Figure 2 we highlight the operations performed multiple times with circular arrows. While the first function reads the workflow input from the object storage (step 1), the second function sends the ready signal on the *sync-pipe* to the local *forwarding agent* (step 2), eventing it is up and running and waiting to read data on the local *data-pipe*. During the intermediate data object *OutA* computation (step 4), the *forwarding agent* on the second node reads the ready signal (step 3) from the *sync-pipe* and waits for the first packet on the *server socket*. Once the first function ends to compute the intermediate data object *outA*, it first writes *outA* size packet and then iteratively writes *outA* content in packets on the local *data-pipe* (step 5). The local *forwarding agent* reads the packets from the *data-pipe* (step 6) and sends them to the *client socket* (step 7). On the consumer side, packets are read from the *forwarding agent* (step 8) and written to the local *data-pipe* (step 9). Finally, packets are read from the second function (step 10), which, once received *outA*, computes the workflow output *out* (step 11) and stores it in the object storage (step 12).

4 EXPERIMENTAL RESULTS

To evaluate our approach, we analyze the impact of different pipe and socket buffer sizes on data communication latencies, and we evaluate Floki in terms of performance and resource usage impact.

4.1 Experimental Setup

To prevent us from benchmarking cloud vendors' specific environments, the experiments are run on an on-premise cloud-prepared environment. Experiments are executed on a virtualized Kubernetes cluster composed of one master, representing the Kubernetes control-plane, and 7 worker nodes on which functions are deployed. A MinIO server, a widely used high-performance object storage, outside the Kubernetes cluster but inside the infrastructure, is considered in the experiments as shared storage. In-memory key-value stores, such as Redis and Memcached, are not considered since they break one of the serverless advantages by requiring users to select instance types in terms of network, compute, and memory resources to satisfy their application requirements. The master runs in a Linux-based VM with 32GB of memory and 16 virtual cores, while the workers run in a Linux-based VM with 128GB of memory and 16 virtual cores. The MinIO server runs bare-metal on a node featuring an Intel® Xeon E5-2620 CPU running at 2.00GHz, interfacing with two 1.6TB Intel® DC P3608 SSDs through NVMe. The VMs are synchronized in the millisecond range. The Kubernetes cluster is mapped on 8 physical nodes residing in the same rack and featuring either an Intel® Xeon Silver 4114 CPU running at 2.20GHz or an Intel® Xeon E5-2630 v4 CPU running at 2.20GHz. Physical nodes are connected through a 10Gbps Brocade VDX6740 network switch. The experimental evaluation reports average results computed over 10 sequential runs for reliability.

4.2 Pipe and Socket Buffer Sizes Analysis

In this analysis, we want to analyze the impact of different buffer sizes on fixed-size data communication over pipes and sockets and find the optimal buffer sizes. The experiments are based on two types of functions: a producer function writing data in packets on a channel and a consumer function reading data in packets from a channel. The evaluation considers different buffer sizes, ranging from 1 system page, i.e., 4KB, to 128 system pages, i.e., 512KB, with the kernel imposed constraint of a power-of-two increment. The lower range limit, i.e., 1 system page, represents the size for which the kernel guarantees pipe writes operations to be atomic. Within Floki, a *data-pipe* is of exclusive use of a single producer at a time; thus, the pipe buffer size can be increased without affecting writes operations atomicity.

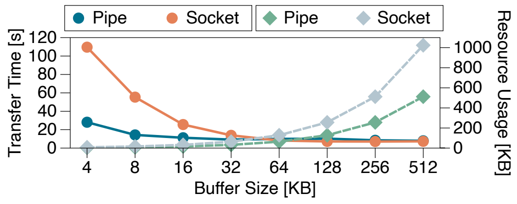

Figure 3: Average transfer time (circles) and resource usage (diamonds) for 16GB data with pipe and socket buffer sizes ranging from 1 to 128 pages (4KB to 512KB).

Figure 3 shows the average transfer times of a 16GB data object with different pipe and socket buffer sizes, and the related resource usage. Note that the socket resource usage is always twice the corresponding buffer size: TCP allocates twice the requested buffer size and uses the extra space for administrative purposes and internal kernel structures. While the difference between the pipe and socket transfer times is significant for small buffer sizes, the transfer times are comparable for big buffer sizes. In particular, as highlighted in Figure 3, 16 system pages buffer size, i.e., 64KB, represents the optimal size, reducing and balancing the pipe and socket transfer times. Increasing the buffer size would provide comparable transfer times while using more resources. Therefore, the following experiments and evaluations consider a buffer size of 16 system pages.

4.3 Performance Evaluation

We conduct a series of experiments to evaluate the performance of Floki in terms of end-to-end times. Targeting data-intensive workloads, the evaluation considers data sizes ranging from 1MB to 16GB with a 2× increment. To measure the end-to-end times, we register the timestamp before each producer function starts to write data and the timestamp after each consumer function finishes reading data. Thus, given N producers and K consumers functions with TS_{p_i} and TS_{c_j} as their timestamps, we derive the end-to-end time T_{E2E} as:

$$T_{E2E} = max(TS_{c_1},..,TS_{c_K}) - min(TS_{p_1},..,TS_{p_N}) \quad (1)$$

As Equation (1) shows, the end-to-end time T_{E2E} accounts for possible not fully concurrent operations by considering the minimum of the producers timestamps TS_{p_i} and the maximum of the consumers timestamps TS_{c_j}.

Figure 4 shows Floki end-to-end time speedups over the object storage solution baseline (horizontal constant solid line). Floki always significantly outperforms the object storage solution in all the analyzed patterns. Contrarily to a naïve solution relying on shared object storage, producers and consumers functions are deployed and run concurrently. The benefits of the volatile data share are more

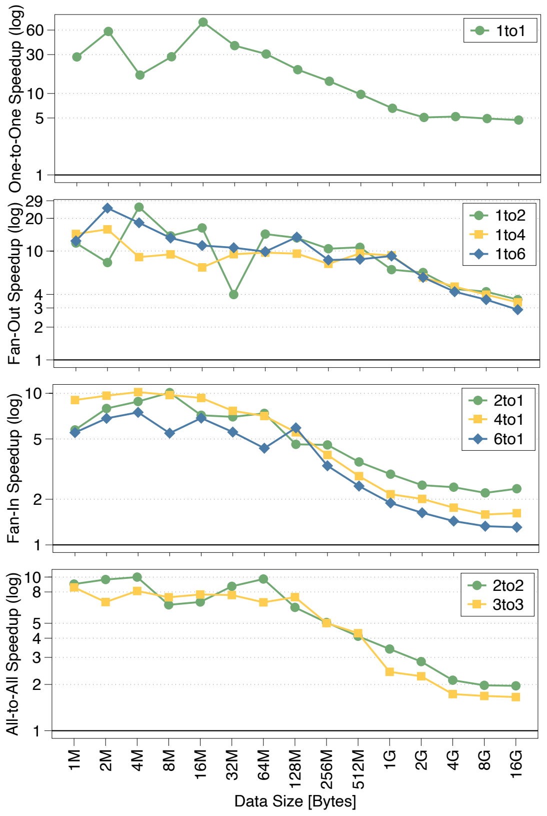

Figure 4: Floki's end-to-end speedups over the object storage solution in the analyzed distributed systems patterns.

visible with small data sizes, i.e., from 1MB to 256MB, for which a higher performance increase is obtained. It is worth noting that, since data objects are read sequentially from the consumer functions, with multiple producers, functions using Floki gain smaller performance than those achieved with a single producer. Floki reduces the end-to-end time up to: 74.95× in the one-to-one pattern; 25.34×, 15.83×, and 24.83× in the fan-out pattern; 10.11×, 10.18×, and 7.49× in the fan-in pattern; 9.99× and 8.11× in the all-to-all pattern. Overall, considering the impact of Floki in terms of end-to-end time, the most significant time-savings are reached with a data size of 16GB, featuring the largest data transfer latency. In particular, the higher time reductions are achieved in the 1to6 pattern, where communication latencies are reduced from 753*s* to 260*s* on average. In other words, Floki allows saving 8.22 minutes on data sharing latency over the object storage baseline requiring 12.55 minutes.

4.4 Resource Usage Evaluation

Resource usage is crucial in serverless environments, where resources are billed with a pay-as-you-go model. We want to estimate and compare the resource usage of Floki to the object storage solution, representing the baseline, in the four considered patterns, i.e., one-to-one, fan-out, fan-in, and all-to-all. To assess the gap of resource requirements between varying data sizes, we choose two extreme cases just for comparison, i.e., 1MB and 16GB. In the following, *DS* represents the data size, *T* the total amount of functions composing the specific pattern, *P* and *C* the number of producers and consumers functions, and *PBD* and *SBD* the pipe and socket local buffer sizes (i.e., 64KB and 128KB), accordingly. The object storage resource usage estimation does not consider the necessary internal buffers to write and read the data object/file since their sizes are negligible compared with the analyzed data sizes. Being the object storage shared among the different functions, we derive the related resource usage $RU_{ObjStorage}$ as:

$$RU_{ObjStorage} = (P * DS)_D \tag{2}$$

The required disk space is proportional to the number of producers functions *P*. Therefore, from a resource usage perspective, there is no difference among the patterns composed of the same number of producers and a different number of consumers. For instance, the fan-in pattern with three producers and one consumer would require the same disk space as the all-to-all pattern with three producers and three consumers. Differently, Floki represents a significantly less expensive resource usage solution. Considering only the memory space needed to hold the local buffers to perform the pipe and socket operations, we derived Floki resources usage RU_{Floki} as:

$$RU_{Floki} = (T * PBD + 2 * P * C * SBD)_M \tag{3}$$

We evaluate Floki resource usage following the presented analysis. By applying Equations (2) and (3), the resource-saving is evaluated by dividing the resource usage of the object store baseline for the Floki resource usage. For example, when sharing 1MB, Floki saves $\frac{1MB}{384KB} = 2.67\times$ of resources compared to the baseline. Floki always demands a significantly lower amount of resources compared to the object storage solution. More precisely, each function composing the workflow only requires 64KB of memory for the pipe buffer and 128KB for each client/server socket buffer, allowing resource-saving to scale linearly with the data size increase. For example, considering the simple one-to-one pattern, the 16GB saving differs from the 1MB saving by a factor of 16,384×, equivalent to the difference between the two data sizes. Overall, Floki achieves up to 50,738× of resource-saving, translating into a memory allocation of roughly 1.9MB instead of an object storage allocation of 96GB.

5 DISCUSSION

Envisioning Floki as part of Knative and Kubernetes-based workflows frameworks, we briefly discuss the made assumptions and their integration limitations.

Assumptions: As introduced in Section 3, Floki requires two inputs: the functions-nodes mapping and the workload DAG. In the currently available computation frameworks, the workload DAG is provided by the user in the form of a configuration file (e.g., JSON file) or a high-level description. Further steps are required to either facilitate the DAG specification in case of complex workflows or to remove the user *from the loop*. Instead, the mapping between functions and cluster nodes can be easily retrieved by inspecting the underlying Kubernetes scheduler.

Interface: Given the current POSIX *write*, writing a pipe requires reading and writing data in batches. Adapting an application would require substituting such *write* functions with *looped writes*, where a simple library function provided as API by Floki could interface such change without changing the programming model.

Fault-Tolerance, Multy-Tenancy, and Data Recovery: Fault-Tolerance, multi-tenancy, and data recovery represent crucial attributes of cloud and serverless computing. Floki needs more effort to be fault-tolerant. Hard multi-tenancy in Kubernetes environments can be achieved through complex namespaces, resource quotas, access control, and virtual cluster configuration, and Floki indirectly guarantees security in a multi-tenant environment. Concerning data recovery, in case of a component failure, intermediate data must be re-computed by re-running the entire workflow. Re-computing only failed functions could lower the overhead; thus, we believe per-function data recovery deserves further investigation.

Porting on Kubernetes-based Workflow Frameworks and Knative: Even though Floki is currently at its first maturity stage, we target to port the proposed solution to Kubernetes-based workflows frameworks and Knative. To integrate Floki with Knative, two main features are required. First, following the underlying orchestration platform feature, it is necessary to enable Knative functions to mount local volumes. Second, on top of the existing Knative Custom Resource Definitions (CRDs) providing sequential and parallel functions invocations, a more general workflow CRD has to be built. While porting Floki to Knative is more complex, the porting on Kubernetes-based workflows frameworks, e.g., Argo, would only require to automatically create the system components.

6 CONCLUSIONS

Executing data-intensive pipelines on serverless requires efficient inter-container data sharing, overcoming state-of-practice storage high-latency access. We tackle this problem by presenting Floki, a system designed for fast point-to-point data sharing. We benchmark Floki on the principal distributed systems communication patterns, considering data transfers from 1MB to 16GB. Performance-wise, when compared to object storage baseline, Floki improves end-to-end time performance up to 74.95×, reducing the largest data-sharing time from 12.55 to 4.33 minutes while requiring up to 50,738× fewer disk resources, with up to roughly 96GB disk space release. Even though Floki is at its first stage of maturity, we aim to integrate it in Kubernetes-based workflow frameworks and Knative to enable high-performance intermediate data exchange.

ACKNOWLEDGMENTS

This work was partially supported by the Ministry of Economy of Spain under contract TIN2015-65316-P, the Ministry of Science under contract PID2019-107255GB-C21/AEI/10.13039/501100011033, and PID-126248OB-I00, and the Generalitat de Catalunya under contract 2014SGR1051.

REFERENCES

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *USENIX Conference on Usenix Annual Technical Conference*, ATC '18, 2018.

[2] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *International Middleware Conference*, Middleware '19, 2019.

[3] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *ACM Symposium on Cloud Computing*, SoCC '19, 2019.

[4] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018.

[5] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, 2021.

[6] 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. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, 2019.

[7] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *European Conference on Computer Systems*, EuroSys '22, 2022.

[8] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, 2018.

[9] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *USENIX Annual Technical Conference*, ATC '21, 2021.

[10] Pedro García López, Marc Sánchez Artigas, Simon Shillaker, Peter R. Pietzuch, David Breitgand, Gil Vernik, Pierre Sutra, Tristan Tarrant, and Ana Juan Ferrer. Servermix: Tradeoffs and challenges of serverless data analytics. *CoRR*, 2019.

[11] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using qoop. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, 2018.

[12] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *USENIX Annual Technical Conference*, ATC '21, 2021.

[13] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, 2020.

[14] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. Ofc: An opportunistic caching system for faas platforms. In *European Conference on Computer Systems*, EuroSys '21, 2021.

[15] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services. *CoRR '19*, 2019.

[16] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *USENIX Conference on Networked Systems Design and Implementation*, NSDI '19, 2019.

[17] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing*, SoCC '12, 2012.

[18] Simon Shillaker and Peter Pietzuch. *FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing*. ATC '20. 2020.

[19] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *CoRR*, abs/2001.04592, 2020.

[20] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *USENIX Conference on Networked Systems Design and Implementation*, NSDI'20, 2020.

[21] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. Boxer: Data analytics on network-enabled serverless platforms. In *CIDR*, 2021.

[22] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *ACM Symposium on Cloud Computing*, SoCC '19, 2019.