BAASLESS: Backend-as-a-Service (BaaS)-Enabled Workflows in Federated Serverless Infrastructures

Thomas Larcher, Philipp Gritsch, Stefan Nastic, *Member, IEEE*, and Sashko Ristov

***Abstract*—Serverless is a popular paradigm for expressing compute-intensive applications as serverless workflows. In practice, a significant portion of the computing is typically offloaded to various Backend-as-a-Service (BaaS) cloud services. The recent rise of federated serverless and Sky computing offers cost and performance advantages for these BaaS-enabled serverless workflows. However, due to vendor lock-in and lack of service interoperability, many challenges remain that impact the development, deployment, and scheduling of BaaS-enabled serverless workflows in federated serverless infrastructures. This paper introduces BAASLESS – a novel platform that delivers global and dynamic federated BaaS to serverless workflows. BAASLESS provides: i) a novel SDK for uniform and dynamic access to federated BaaS services, reducing the complexity associated with the development of BaaS-enabled serverless workflows, ii) a novel globally-federated serverless BaaS framework that delivers a suite of BaaS-less ML services, including text-to-speech, speech-to-text, translation, and OCR, together with a globally-federated storage infrastructure, comprising AWS and Google cloud providers, and iii) a novel model and an algorithm for scheduling BaaS-enabled serverless workflows to improve their performance. Experimental results using three complementary BaaS-enabled serverless workflows show that BAASLESS improves workflow execution time by up to 2.95× compared to the state-of-the-art serverless schedulers, often at a lower cost.**

***Index Terms*—Backend-as-a-Service (BaaS), federation, optimization, SDK, serverless.**

I. INTRODUCTION

THE emergence of serverless computing has ushered in a transformative paradigm, alleviating developers from the intricacies of infrastructure management [1], [2], [3], [4]. Particularly popular for accommodating compute-intensive applications through serverless workflows [5], [6], [7], [8], [9], [10], [11], serverless typically offloads substantial computation to managed Backend-as-a-Service (BaaS) offerings [12], [13], [14], [15]. For instance, for the *BaaS service OCR* (Optical Character Recognition), AWS Textract and GCP Vision are the *BaaS service deployments* offered by AWS and GCP, respectively. Other BaaS services that we will use in this paper include speech-to-text (*S2T*), text-to-speech (*T2S*), and translation (*TRA*). These AI-based BaaS services are pre-trained, and users do not need to invest in infrastructure and expensive training of AI models but can use them with a serverless paradigm. Unfortunately, similar to other serverless services, cloud providers restrict the size of the input data, which requires the users to create *fork-join* serverless workflows that split the input size into smaller chunks, call multiple instances of the BaaS service, and merge their results at the end. Other examples include *bag-of-tasks*, such as processing multiple audio files from news or customer support. Finally, due to the lack of support for complex BaaS services, such as translating audio files or creating a translated speech from a PDF file, users create composite BaaS services with serverless workflows [16]. We use the term *BaaS-enabled serverless workflows* to denote such serverless applications.

Recently, federated serverless computing [17], [18], [19], [20], [21] and Sky Computing [22], [23] have been gaining in popularity due to the numerous benefits they can bring to serverless workflows. Workflow functions are deployed across multiple cloud providers to reduce cost [24], improve scalability [25], [26], reduce workflow execution time (*makespan*) [17], [19], [27], or increase serverless workflow resilience [28]. However, all these methods are mainly applicable for serverless workflows with isolated functions that do not use external BaaS services. Instead, computing is performed within the functions, and the intermediary results are often transferred through storage [23], [29]. This inherent vendor lock-in, combined with a huge number of (vendor-specific) BaaS services and their configuration options, makes it difficult to reap the benefits of the *federated serverless infrastructure* in practice for BaaS-enabled serverless workflows. Consequently, the federated serverless infrastructure raises three challenges that impact the development, interoperability, scheduling, and optimal execution of BaaS-enabled serverless workflows.

First, while various mature approaches exist to run portable serverless functions with interoperable serverless workflow management systems, there is a lack of methods for portable and interoperable BaaS services. For instance, various executor functions may load the function code from storage and run it as a serverless function (FuncX [30] or Lithops [26]), or simply achieve interoperability at the Function-as-a-Service (FaaS) level by replacing function URLs in the workflow language, such as xAFCL [17]. Unfortunately, these methods have not

Manuscript received 10 January 2024; revised 30 July 2024; accepted 2 August 2024. Date of publication 6 August 2024; date of current version 6 December 2024. This work was supported in part by the Land Tirol, Austria, under Grant F.35499, in part by the MATISSE project, funded by the European Union under Grant 101140216 (KDT Joint Undertaking), and in part by the Österreichische Forschungsförderungsgesellschaft mbH (FFG) under Grant FO999909832. Recommended for acceptance by T. Kosar. *(Corresponding author: Sashko Ristov.)*

Thomas Larcher, Philipp Gritsch, and Sashko Ristov are with the University of Innsbruck, 6020 Innsbruck, Austria (e-mail: t.larcher@student.uibk.ac.at; philipp.gritsch@uibk.ac.at; sashko.ristov@uibk.ac.at).

Stefan Nastic is with the TU Wien, 1040 Vienna, Austria (e-mail: snastic@dsg.tuwien.ac.at).

Digital Object Identifier 10.1109/TCC.2024.3439268

been investigated for BaaS services because functions are seen as black boxes. The existing interoperable infrastructure-based SDKs (Apache Libcloud, jClouds, and pckcloud) are mainly focused on storage infrastructure only. Still, they offer a partial solution for interoperability and restrict the function of using a single storage at a time. As a consequence, if the number of providers is p, developers are required to code $O(p^2)$ functions with different SDKs to be able to call different BaaS service deployments for the same BaaS service to exploit the federated serverless infrastructure. Secondly, providers' vendor lock-in often restricts their FaaS, BaaS, and storage systems to internal communication and prevents cross-provider communications. Even with multiple function deployments, BaaS services that read the input file by reference cannot access the storage of other providers, although access to their buckets may be faster. Unfortunately, the vendor lock-in increases the complexity to $O(p^3)$ to code separate functions on each provider, which access the BaaS service deployment on any provider and store the output data in any storage. For example, let's consider a use case where the data input is stored in AWS S3, and the user may deploy the function, BaaS service, and output storage across AWS and GCP. In such case, the user may select to colocate all three services on AWS. Another option is to keep the function and BaaS service on AWS, but store the output in GCP Storage. For example, if the subsequent function needs to run a BaaS service supported by GCP only. Similar to binary numbers, in the eighth option, the user selects to run all three services on GCP. Finally, the existing schedulers in federated serverless and Sky computing are either optimizing serverless workflows in a single cloud region [8], [10], [11], [31], [32], [33] or across multiple regions of various cloud providers [19], [27], [34], [35]. Unfortunately, these methods federate only the FaaS, thereby finding only a small subset of the solutions for BaaS-enabled serverless workflows or requiring significant costs to configure the schedulers with all parameters. We detail all the aforementioned challenges in Section II.

To address the above challenges, this paper introduces BAASLESS – a novel platform that delivers global and dynamic federated serverless infrastructure to BaaS-enabled serverless workflows, comprising federated FaaS, BaaS, and storage. The main aim of the BAASLESS platform is to enable a novel BaaS-less paradigm that builds on the serverless' core principles, extending them to also shield developers from scheduling, deployment, and management of supporting BaaS services. We describe the BAASLESS approach and the high-level system architecture in Section III. BAASLESS introduces an interoperable SDK for four BaaS services *T2S*, *S2T*, *TRA*, and *OCR*, that seamlessly call the respective BaaS service deployments of AWS and GCP. With the SDK, developers code BaaS-enabled serverless workflows with BaaS services only once, and they can dynamically select the BaaS service deployment. Afterwards, BAASLESS uses its mathematical model and scheduling algorithm to select the specific BaaS service deployment for each BaaS service during runtime, such that the workflow makespan is minimized (Section IV). When needed, BAASLESS transparently moves the intermediary inputs and outputs between workflow functions to the respective storages to overcome vendor lock-in. To evaluate BAASLESS, we conducted a series of experiments (Section V). We first developed a fork-join and a bag-of-tasks BaaS-enabled serverless workflow using the BAASLESS SDK. Using the set of microbenchmarks, we configured the federated serverless infrastructure for AWS and GCP. Further on, the BAASLESS scheduler determined the fastest scheduled deployments of both workflows when the input data was stored in AWS or GCP (Section VI). Finally, we used a MapReduce-based serverless workflow and evaluated BAASLESS framework's weak and strong scaling while increasing the number of available regions (Section VII).

The main components of the BAASLESS platform, which are also the main contributions of this paper, include:

- *A novel globally-federated serverless BaaS framework*, which delivers i) a suite of BaaS-less ML services, including *T2S*, *S2T*, *TRA*, and *OCR*, with ii) globally-federated storage infrastructure, comprising AWS and GCP, and iii) synchronization mechanisms to overcome vendor lock-in.
- *The novel SDK*,[1] which provides uniform and dynamic access to federated BaaS services (including federated storage), enabling the development of BaaS-enabled serverless workflows, i.e., writing FaaS functions without worrying about the management and deployment of the supporting BaaS service deployments.
- *A novel model and algorithm for scheduling BaaS-enabled serverless workflows*, which reduces workflow makespan by up to $2.95\times$ compared to the state-of-the-art serverless schedulers while keeping the costs comparable.

II. MOTIVATING STUDY

We first introduce BaaS services and motivate the need to create BaaS-enabled serverless workflows and run them on a federated serverless infrastructure. We also present the BaaS service constraints due to vendor lock-in and BAASLESS approach to overcome the state-of-the-art limitations.

A. Motivation for BaaS-Enabled Serverless Workflows

Cloud providers offer many AI-based services that are already trained to conduct various human-based recognitions. For instance, the AI-based BaaS service for speech recognition converts given audio into text (*S2T*). Other BaaS services convert text into speech (*T2S*), recognize text from an image (*OCR*), or translate one natural language into another (*TRA*). These and other similar BaaS services are already trained, and developers simply need to call them using SDKs from the respective providers.

To minimize the invocation latency, developers may use a single function to call multiple BaaS services. However, including a new SDK would increase the size of the function deployment package [36]. Unfortunately, the number of BaaS services a function may call is restricted because cloud providers limit the size of the function deployment package to several tens of megabytes. Therefore, developers are forced to create pipelines

[1] https://github.com/FaaSTools/CORE

TABLE I
LIMITATIONS AND CHARGES OF AWS' AND GCP'S BAAS SERVICES

BaaS	AWS	GCP	AWS	GCP
	limit		charge	
S2T	4 h	1 min	¢2.4 per 1 min	
T2S	3,000 chars	5,000 chars	$4 per 1M chars	
OCR	10 MB	20 MB	$1.5 per 1000 units	
Translation	10 kB	100 kB	$15 / $20 per 1M chars	

of serverless functions that call individual BaaS services to create composite BaaS services [16].

Regrettably, a pipeline of composite BaaS services does not solve all issues. For instance, AWS *OCR* restricts the input to a single-page PDF and 10 MB. Similar limitations hold for other BaaS service deployments (see Table I). Therefore, users must split the large input files into smaller chunks before processing them. Moreover, cloud providers do not offer a service to split a large PDF or process multiple files in parallel. To overcome this constraint, users need to use serverless workflows to compose more complex BaaS services and use parallelism to scale the problem size. In that case, users need to invoke the BaaS service multiple times after splitting the input data into smaller parts. We refer the reader to our recent paper [16], where we detail how to develop composite and scalable BaaS services as serverless workflows.

B. BaaS Service Challenges Due to Vendor Lock-In

While building BaaS-enabled serverless workflows may overcome the limitations of individual BaaS services and allow scalability, it opens another challenge that is initiated by the vendor lock-in. Namely, since providers restrict direct communication between serverless functions, developers must build workflows that exchange intermediary results through storage [29]. Unfortunately, developers are restricted to calling BaaS services by reference that points to a file stored in the storage from the same provider. Otherwise, developers must code their functions to download the file to the file system of the function and then call the BaaS service by value. Still, some BaaS service deployments, such as AWS S2T, must be invoked by reference from AWS S3. In general, there are two approaches. The first approach is simpler and it "colocates" the BaaS service with the storage where the input data is stored. The second approach requires considerable development effort to move the input data to the provider whose BaaS service deployment is executed. After the BaaS service finishes, it returns the output to the function by value, which later requires developers to code the function to store the output in storage so that it is accessible by the successor functions.

C. Performance Analysis of BaaS-Enabled Serverless Functions

In line with the challenges, we conducted a set of benchmarks that investigate the effect of a provider on BaaS service execution time. Surprisingly, the measurements identified many cases where cross-provider federated deployments of serverless workflows are faster than colocating workflow functions and BaaS

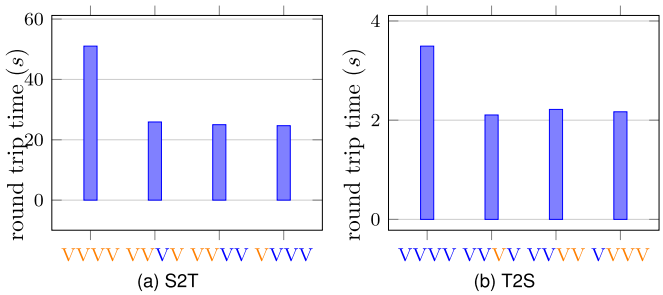


Fig. 1. Significantly lower BaaS service time by migrating BaaS service and data for (a) S2T and (b) T2S.

services within a single provider. We use the notation $<$ XXXX $>$ for a given deployment of the BaaS-enabled function, which denotes the locations of the **input** storage, the **function**, the **BaaS** service, and the **output** storage, respectively from left to right. The marks X may have a value of the region initials (V, L, B, and F for North Virginia, London, Belgium, and Frankfurt, respectively). Since we use the regions located in London and North Virginia of both AWS and GCP, we use orange (lighter in black and white print) and blue color (darker) to distinguish between AWS and GCP, respectively. We also use "–" if the function does not use a BaaS service or storage. For easier memorization, we recommend remembering the abbreviation **IFBO** (**I**nput storage, **F**aaS, **B**aaS service, **O**utput storage). We denote the BaaS service with an italic abbreviation, e.g., (*S2T*) and the respective function with S2T, which calls *S2T*.

We deployed and ran the function S2T with 128MB RAM in the North Virginia regions of AWS and GCP, which converts an audio file (3.8MB) into text (2.2kB). S2T can run the respective AWS and GCP *S2T* BaaS service deployments with the input file stored in AWS S3. Fig. 1 shows that the BaaS service on another provider than the input data storage runs faster than the colocated one. Fig. 1(a) shows that the S2T function in AWS finishes in 51.04 s with the colocated AWS *S2T* (VVVV). However, the VVVV setup, i.e., calling GCP's BaaS service, finishes in 25.9 s, or 1.97× faster.

We also deployed the complementary function T2S, this time in GCP, which uses the input data from GCP storage. We observe in Fig 1(b) that there is no dominating BaaS provider in terms of execution time. The VVVV setup of T2S finishes in 3.49 s. However, although we expected that the colocated BaaS service would run faster than the cross-region one, we observed that the VVVV setup runs 1.65× faster. Notably, this setup is faster than the other two setups VVVV and VVVV. All these three setups can run with the BAASLESS' SDK introduced in this paper.

Summary of observations: The initial performance analysis led us to several crucial observations that motivated our work. First, there is no dominant provider for all BaaS services in terms of performance. Second, the general heuristics that running all cloud services colocated in a single cloud region would improve communication efficiency does not always hold for BaaS-enabled workflows. Finally, there may be a cost-performance trade-off with this approach. However, the potential increase in the monetary costs is negligible compared to the

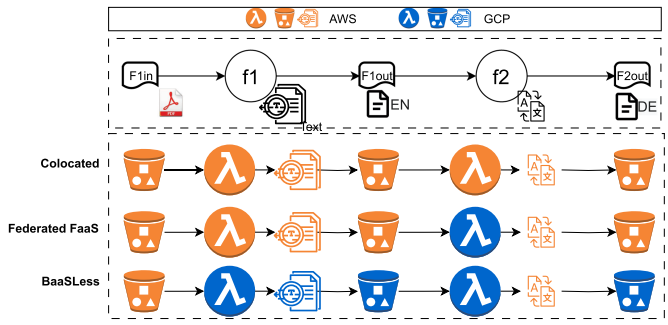


Fig. 2. Example of workflow deployment with federated infrastructure, when input data is stored in AWS S3. The colocated approach minimizes data access time by deploying everything in a single provider region. Federated FaaS selects functions from different providers (`f2` on GCP). Finally, BAASLESS can run the functions and BaaS services and store data in any provider.

significant performance improvement (as discussed in Section VIII-B). These observations lead to the need for a scheduler that distributes cloud services and data across federated serverless infrastructures. This may significantly improve performance for a negligible increase in costs.

D. Problem Statement and Limitations of the State-of-The-Art

Let's consider a BaaS-enabled serverless workflow (Fig. 2) that runs a sequence of two functions, `f1` and `f2`, which call the BaaS services *OCR* and *TRA*, respectively. The top of Fig. 2 shows the workflow function processing steps, their expected inputs and outputs, and invoked BaaS services. The bottom of Fig. 2 shows the state-of-the-art approaches for workflow deployment with the fastest makespan across the orange (lighter in black and white print) AWS and blue (darker) GCP regions. Due to the data flow, there is an additional complexity that should be evaluated, i.e., the output location of `f1` directly affects the deployment and performance of `f2`.

We found two approaches that utilize serverless federated infrastructures. FaDO [27] colocates the functions in the same cloud region where the input data is stored. By colocating, FaDO uses network proximity to minimize the latency of data access between the functions and storage. With this approach, if the input data is stored in AWS, both functions will run on AWS Lambda and use AWS S3. Moreover, AWS BaaS services must be used due to vendor lock-in. Another approach, FaaSt [19], federates FaaS and schedules the functions across different providers using the earliest finish time. Unfortunately, due to the vendor lock-in, it can neither store the data in GCP storage nor dynamically select the BaaS service that loads or stores data from the storage of another provider. With the measured data, we show in Fig. 2 that FaaSt is able to select running the function `f2` on GCP, thereby providing a faster solution than the colocated one.

III. BAASLESS PLATFORM OVERVIEW

To overcome the limitations of state-of-the-art approaches, we introduce BAASLESS, a platform that determines the deployments of functions and attached BaaS services and storages in order to minimize the workflow makespan. Our evaluation shows that although the input data is stored in AWS, BAASLESS decides to deploy the function `f1` from Fig. 2 in GCP, which calls the GCP *OCR* and also stores the output data in GCP storage. Furthermore, the function `f2` also runs on GCP, but it calls the faster BaaS service deployment AWS *TRA*, which stores the result in GCP storage. Note that BAASLESS solves the vendor lock-in by transparently moving the data from S3 to GCP storage (see Section IV-B3).

Fig. 3 illustrates the BAASLESS high-level architecture consisting of two main components: i) BAASLESS BaaS and storage attachment SDKs for federating serverless BaaS and storage infrastructure, and ii) BAASLESS scheduler for specifying the workflow deployment, including the specific BaaS service and storage deployments for each BaaS-enabled function, such that the workflow makespan is minimized.

A. Composing BaaS-Enabled Serverless Workflows

The BAASLESS platform has two internal representations of BaaS-enabled serverless workflows:

1) *Abstract serverless workflow* (Fig. 3 left), which contains all functions, BaaS services, and data flow dependencies but attaches them to no BaaS deployments and storage backends;
2) *Deployed serverless workflow* (Fig. 3 right), which enhances the abstract serverless workflow with the BaaS and storage attachments configured to each function deployed on a cloud provider by the BAASLESS scheduling algorithm.

A developer creates an abstract serverless workflow in two steps: function development and workflow composition. Initially, a developer codes serverless functions and uses the BAASLESS BaaS and storage attachment SDK to attach an *abstract BaaS and storage* to each function. For example, the function in Fig. 3 (left-hand side) is coded to call the abstract BaaS service *S2T*, which receives data from the attached input storage and stores the result in the abstract output storage. While coding, the developer does not need to know about the underlying BaaS service deployment and storage attachment details, such as the storage provider or provider SDK, as BAASLESS provides abstractions and provisions them during runtime. In the second step, the developer composes the functions into a serverless workflow through control and data flow dependencies. BAASLESS uses the Abstract Function Choreography Language [6] (AFCL) because it composes workflows at a high level of abstraction. After composing the abstract serverless workflow with abstract BaaS services, the developer forwards it to the BAASLESS scheduler for the deployment of functions onto concrete computational provider regions, along with BaaS service and storage attachments.

B. BAASLESS Scheduler

To bootstrap the scheduling algorithm in practice, BAASLESS performs a set of initial microbenchmarks to map the state of its federated infrastructure, including federated FaaS, BaaS, and storage. It runs all deployments of the workflow

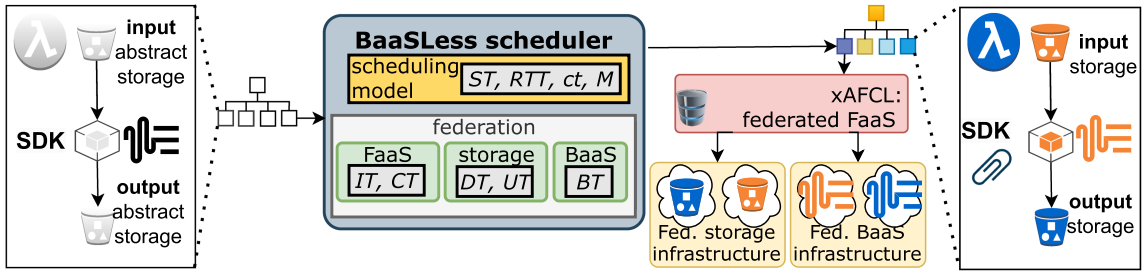


Fig. 3. Overview of BAASLESS architecture.

functions with all possible BaaS deployments. To minimize the microbenchmark, BAASLESS derives a regression function for all the collected data transfer times and file sizes for determining the unidirectional latency and bandwidth between all regions of functions and storage attachments. Further on, BAASLESS decides which storages to attach to each function deployment based on the file size.

Fig. 3 (right-hand side) shows the BAASLESS scheduler downloading the input file from AWS S3 storage, which was stored by some of the predecessor functions. While the input files always have a given known location, the BAASLESS scheduler decides the deployment provider of each function, the BaaS service *S2T* and the storage of the generated intermediary results based on its round-trip time estimation and the storage of its predecessors. In our example, the function is deployed in GCP, the BaaS service (*S2T*) is deployed on AWS, and the GCP storage is attached. The result of this process is the deployed serverless workflow, ready for execution, comprising function deployments, along with deployed BaaS service and attached storage.

IV. SCHEDULING WORKFLOWS WITH BAASLESS

We further introduce the BAASLESS' scheduler that dynamically selects, provisions, and configures suitable BaaS and storage backends in a federated serverless infrastructure to minimize the workflow makespan. We first define the abstract serverless workflow application and deployment models, which underpin the BAASLESS scheduler. Table II summarizes all notations and abbreviations used in the model.

A. BAASLESS Application Model

1) Abstract Serverless Workflow Model: BAASLESS models *BaaS-enabled serverless workflow* applications $BW = (\mathbb{F}, \mathbb{D})$ as a composition of:

- n abstract serverless *functions* $\mathbb{F} = \bigcup_{i=1}^{n} f_i$,
- interconnected in a directed acyclic graph through a set of *data-flow dependencies* $\mathbb{D} = \{(f_i, f_j, d_{ij}) \in \mathbb{F} \times \mathbb{F} \times \mathbb{R}\}$, expressing that the function f_j can start only after all its predecessors f_i completed and generate d_{ij} megabytes of *intermediary result*.

Functions exchange intermediary results implicitly through references and transfer them during their runtime. The size of intermediary results lies in various ranges. For instance, the order of magnitude of audio files may be a few megabytes and beyond, which requires temporary saving in external storage.

TABLE II
NOTATION SUMMARY

Notation	Definition
BW	BaaS-enabled serverless workflow
n	number of functions in a workflow
$\mathbb{F} = \bigcup_{i=1}^{n} f_i$	set of n workflow functions f_i
$\mathbb{D} = \{(f_i, f_j, d_{ij})\}$	a set of *data-flow dependencies*
d_{ij}	megabytes of intermediary result
N_{ij}	the number of files in d_{ij}
SZ_{ij}	the size of all files in d_{ij}
$\mathbb{R} = \bigcup_{j=1}^{R} r_j$	a set of cloud *regions*
r_f	region of a function
r_b	region of the BaaS service
r_s	region of the output storage of a function
c_r	concurrency limit of a service in the region r
$TT(r_f, r_s, dir)$	transfer time of the intermediary result d_{ij}
dir	unidirectional upload or download
$L(r_f, r_s, dir)$	latency between a function and storage
$B(r_f, r_s, dir)$	bandwidth between a function and storage
RTT	round trip time of a function
FD_i	function deployment with fixed, r_f, r_b, r_s
$\mathbb{FD}_i$	the union of all FD_i of a function f_i
ST_i	sync time
IT_i	invocation time
DT_i	download time
UT_i	upload time
CT_i	computation time
BT_i	BaaS service time
$ct(FD_i)$	completion time of FD_i
M_{BW}	makespan of a workflow BW

Each serverless workflow has two special no-op functions with zero round-trip time: $RTT = 0$:

- *start function* f_s with no predecessors: $\nexists (f_i, f_s, d_{is}) \in \mathbb{D}$
- *end function* f_e with no successors: $\nexists (f_e, f_i, d_{ei}) \in \mathbb{D}$.

Additionally, we define two special data flows:

- *workflow input*, representing all data flowing from the start function: $\forall d_{si} : (f_s, f_i, d_{si}) \in \mathbb{D}$;
- *workflow output*, representing all data flowing into the end function: $\forall d_{ie} : (f_i, f_e, d_{ie}) \in \mathbb{D}$;

2) Abstract BaaS-Enabled FaaS Model: Fig. 4 presents our generic *abstract BaaS-enabled FaaS model* of a serverless function. It comprises i) attached *abstract input storage*, ii) *function computation* (e.g., to split large input file), iii) an *abstract BaaS service* (e.g., *S2T*), and iv) attached *abstract output storage*.

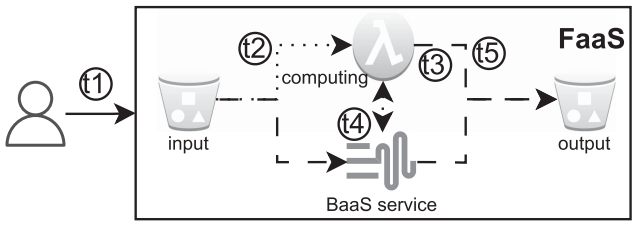


Fig. 4. Abstract BaaS-enabled FaaS model. Dotted lines represent data transfer by value, while the dashed lines by reference.

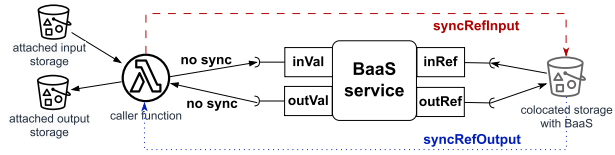


Fig. 5. BaaS service model with I/O data ports *by value* (`inVal` and `outVal`) and *by reference* (`inRef` and `outRef`). BAASLESS introduces `syncRefInput` and `syncRefOutput` primitives to synchronize the attached I/O storage with the storage colocated with the BaaS service.

Fig. 4 shows various data transfer options as defined by the BAASLESS model. We recognize five tasks that need to be finished in general, marked in Fig. 4:

- t1 *invoke the function* from the serverless workflow management system;
- t2 *load data* from the storage to the function file system;
- t3 *run computation* in the function;
- t4 *run the offloaded BaaS service*;
- t5 *deliver the results* to the output storage.

B. BAASLESS Deployment Model

1) Federated Serverless Infrastructure Model: The federated serverless infrastructure comprises several *providers* (e.g., AWS, GCP) offering multiple computational *regions* $\mathbb{R} = \bigcup_{j=1}^{R} r_j$. Each region has an associated *storage backend* as a publicly available object store (e.g., AWS S3, GCP Storage) for saving the intermediary result transferred between workflow functions. Each region also offers a set of BaaS service deployments. Finally, each region offers FaaS (e.g., AWS Lambda or GCP functions). Each region restricts the number of service instances that are running at each point of time. We refer to this metric as concurrency limit c_r of a BaaS service deployment in a given region r.

We model the BAASLESS globally federated serverless infrastructure as a logical overlay FaaS, BaaS, and storage layers of FaaS, BaaS, and storage backends deployed in different providers and regions. The BAASLESS scheduler:

- deploys each abstract function in region r_f;
- selects BaaS service in region r_b; and
- attaches output storage from region r_s,

for each workflow function f_i to minimize makespan.

2) Storage Attachment Model: We use a simplified version of the model presented in [37] to estimate the *transfer time* of intermediary result d_{ij} of N_{ij} files between the function deployment region r_f and the region r_s of the attached storage:

$$TT(r_f, r_s, dir) = N_{ij} \cdot L\left(r_f, r_s, dir\right) + \frac{SZ\left(d_{ij}\right)}{B\left(r_f, r_s, dir\right)} \quad (1)$$

where $L(r_f, r_s, dir)$ and $B(r_f, r_s, dir)$ are the unidirectional *latency* and *bandwidth* between the function deployment and intermediary result storage regions, depending on the *transfer direction* $dir = \{\texttt{down}, \texttt{up}\}$ (download or upload).

We determine the latency and bandwidth between any pair of regions by performing a set of micro-benchmarks using a function that copies files with various files and use a linear regression function to fit the measurement data.

3) BaaS Services Deployment Model: Similar to serverless functions, BaaS services are invoked with input data by value (explicitly). However, for some BaaS services, only a reference to the input or output data is submitted or received. Accordingly, we introduce the BaaS service with input and output ports *by value* (`inVal` and `outVal`) and *by reference* (`inRef` and `outRef`), as presented in Fig. 5. Both *by value* ports (`inVal` and `outVal`) are used to transfer data by value, while `inRef` and `outRef` are used by the BaaS service to load larger files during the service time from the colocated storage in the region r_b.

Due to vendor lock-in, BaaS services deployed on one provider cannot use storages from another provider. To overcome this constraint, *BAASLESS introduces i) a* `syncRefInput` *(dashed line in Fig. 5) and ii) a* `syncRefOutput` *(dotted line in Fig. 5) to synchronize input and output data with the colocated storage in* r_b.

In (2), we use the storage attachment model from (1) to model the *sync time* ST of the BaaS service deployed in region r_b, with colocated storage in r_b, which is invoked by the function f_i from the region r_f.

$$ST_i = x_{\sin} \cdot TT(r_f, r_b, up) + x_{sout} \cdot TT(r_f, r_b, down) \quad (2)$$

The coefficients x_{sin} and x_{sout} have value 1 if the respective `sync` primitives are used and 0 otherwise.

4) Function Deployment Model: We define a *function deployment* as quintuple $FD_i = (f_i, r_f, r_b, r_s, RTT_i)$ that associates function f_i with computation region r_f, attached BaaS service region r_b, attached storage region r_s and expected *round-trip time* RTT_i. We model RTT_i based on the five tasks presented in Fig. 4 and sync mechanisms in Fig. 5:

$$RTT_i = IT_i + DT_i + CT_i + BT_i + ST_i + UT_i, \quad (3)$$

- *invocation time* IT_i necessary to invoke the function using the model presented in [38];
- *download time* $DT_i = \sum_{(f_p, f_i, d_{pi}) \in \mathbb{D}} TT(r_p, r_f, \texttt{down})$ to copy input data to the deployed function FD_i;
- *computation time* CT_i within the function;
- *BaaS service time* BT_i to process the BaaS service;
- *sync time* ST_i for the `sync` primitives; and
- *upload time* $UT_i = TT(r_f, r_s, \texttt{up})$ to copy output data to the attached output storage region r_s.

Note that not all BaaS-enabled functions need to conduct all five tasks presented in Fig. 4. For instance, if a function is

invoked from a trigger within the region, then $IT = 0$. For isolated functions, such as Monte Carlo simulation, $DT = BT = UT = 0$. Finally, functions that only wrap the BaaS services have negligible computation time, or $CT = 0$.

5) Workflow Deployment Model: We model the workflow deployment based on the deployment of its functions, BaaS service, and storage attachments. First, we model the *earliest start time* $est(f_i)$ of a function f_i as the latest completion time of its predecessors $pred(f_i)$:

$$est\left(f_i\right) = \max_{f_p \in pred(f_i)} \Big[ct\left(FD_p\right)\Big], \qquad est\left(f_s\right) = 0. \tag{4}$$

Further on, we model the *completion time* $ct(FD_i)$ of a function f_i deployed in a region $r_f \in \mathbb{R}$ as the earliest start time $est(f_i)$ plus its round-trip time RTT_i:

$$ct\left(FD_i\right) = est\left(f_i\right) + RTT_i. \tag{5}$$

Finally, we express *makespan* M_{BW} of a BaaS-enabled workflow through completion time of the end function f_e:

$$M_{BW} = ct\left(FD_e\right). \tag{6}$$

C. BaaSLess Scheduling Algorithm

The BaaSLess scheduling aims to find the "optimal" function deployment, BaaS implementation, and intermediary result storage regions, for a BaaS-enabled workflow application $BW = (\mathbb{F}, \mathbb{D})$ with a given workflow input. BaaSLess assumes the availability of a *deployment set* $\mathbb{FD}_i = \bigcup_{r_f \in \mathbb{R}} FD_i$ for each workflow function $f_i \in \mathbb{F}$, representing its deployments on all regions $r \in \mathbb{R}$. BaaSLess uses the deployment set to benchmark their round-trip time RTT_i and uses (1) and (3) to determine their invocation IT_i, download DT_i, computation CT_i, BaaS service BT_i, sync ST_i, and upload UT_i times. The benchmarks also provides complete intermediary result size information: $SZ(d_{ij}), \forall (f_i, f_j, d_{ij}) \in \mathbb{D}$.

1) Scheduling Algorithm: Algorithm 1 shows the BaaSLess scheduling algorithm as an extension of the heterogeneous earliest finish time (HEFT [39]) algorithm that considers BaaS service location and temporary storage of intermediary results. The algorithm has three input parameters: i) a BaaS-enabled workflow BW, ii) a federated infrastructure with R regions, and iii) the deployment sets of all functions $\mathbb{FD}_i, \forall f_i \in \mathbb{F}$ (with the corresponding invocation, download, computation, service, and upload time benchmark information). Firstly, line 2 sorts all workflow functions according to their B-rank [39], representing the critical path to the end of the workflow based on their benchmarked round-trip time. Line 3 initializes the workflow deployment plan with an empty set. Lines 4 to 31 iterate the functions according to their B-rank order to deploy them on the "best" region, select the best region of the BaaS service, and dynamically attach storage.

For each function (line 4), the algorithm first initializes the completion time, RTT, and calculates the earliest start time of the function in lines 5 to 7, respectively. Further on, the algorithm traverses the function deployments of the function f_i in lines 8 to 29. For each deployment, it loads invocation time IT, computation time CT, and calculates the download time with (1). These parameters are known since the regions of the input files and the deployment is known at this step.

Further on, the algorithm iterates over each BaaS region (lines 12 to 28). Inside the loop, the algorithm loads the BaaS service time and computes synchronization time for `syncRefInput` with (2), if interoperability is needed for the input.

Finally, the algorithm iterates over each storage for the output (lines 17 to 27). It first computes synchronization time for `syncRefOutput` with (2), if the interoperability is needed for the output. Furthermore, since the function deployment and output storage regions are iterated (known), the algorithm computes upload time UT with (1). Afterwards, it uses all computed and loaded parameters and computes RTT with (3). This value is then applied in (5) together with the earliest start time that was already computed in line 7 for this function. At each nested iteration, the algorithm saves the deployments with the earliest completion time (lines 24 to 26) and adds the fastest one to the workflow deployment in line 30. Line 32 returns the final workflow deployment, ready for execution.

2) Complexity: The BaaSLess scheduling algorithm conceptually iterates over four nested loops: once across all n workflow functions and trice across all deployment, BaaS, and storage regions R, leading to an acceptable polynomial complexity of $O(n \cdot R^3)$.

Algorithm 1: BaaSLess Scheduling Algorithm.

```
   Input  : BW = (F, D), F = ⋃_{i=1}^{n} f_i;      // Serverless workflow
   Input  : R = ∪_{r=1}^{R} r;     // Federated infrastructure regions
   Input  : FD_i = ⋃_{r=1}^{R} FD_i, ∀f_i ∈ F;    // Function deployments
   Output: D_BW = ∪_{f_i∈F} FD_ir;                // Workflow deployment
1  Function BaaSLess (BW, R, FD):
2      F ← B − Rank(F);    // Reorder functions with B-rank
3      D_BW ← ∅ ;          // Initialize workflow deployment
4      for i ← 1 to n do                          // Iterate functions
5          ct_min ← ∞;                            // Initialization
6          RTT ← 0;                               // Initialization
7          est_max ← est(f_i);  // calculate est based on Eq. 4
8          for r_f ← 1 to R do  // Iterate func. deployment set
9              IT ← IT_i ;              // load invocation time
10             CT ← CT_i ;              // load computation time
11             DT ← TT(r_f, r_s, down) ; // Calculate download
                time with Eq. 1
12             for r_b ← 1 to R do   // Iterate service regions
13                 BT ← BT_i ;     // load BaaS service time
14                 if x_sin = 1 then
15                     ST ← TT(r_f, r_b, down) ; // latency for
                        syncRefInput primitive based on
                        Eq. 2
16                 end
17                 for r_s ← 1 to R do  // Iterate out. regions
18                     if x_sout = 1 then
19                         ST ← TT(r_f, r_b, up) ;    // latency
                            for syncRefOutput primitive
                            based on Eq. 2
20                     end
21                     UT ← TT(r_f, r_s, up) ;      // Calculate
                        upload time with Eq. 1
22                     RTT ← RTT_i ;       // based on Eq. 3
23                     ct(FD) ← ct(FD_i) ;  // based on Eq. 5
24                     if ct(FD_i) < ct_min then
25                         FD_min ← (f_i, r_f, r_b, r_s, RTT);
                            // Save fastest deployment
26                     end
27                 end
28             end
29         end
30         D_BW ← D_BW ∪ FD_min;  // Add funct. deployment
31     end
32     return D_BW;              // Return workflow deployment
33 return
```

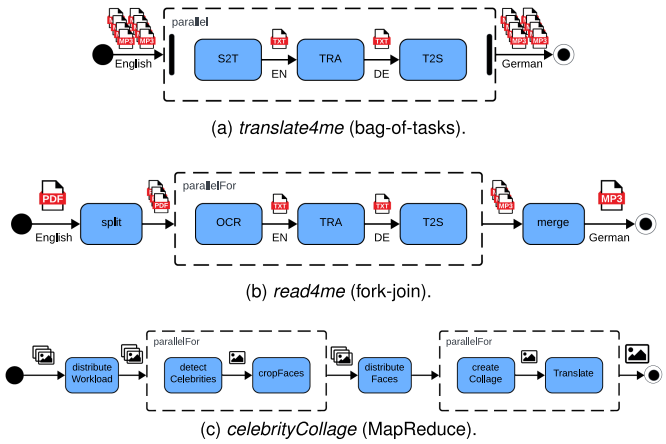


Fig. 6. Experimental BaaS-enabled serverless workflows.

V. Testing Methodology

We conduct a comprehensive evaluation to minimize external threats to validity and to generalize BaaSLess evaluation. We evaluate BaaSLess on six regions of AWS and GCP hosted in the US and Europe. We are using three complementary BaaS-enabled workflows (fork-join, bag-of-tasks, and MapReduce). Further on, we validate our model with a detailed analysis with low concurrency. Finally, we evaluate BaaSLess for weak and strong scaling.

A. Diverse Benchmark BaaS-Enabled Serverless Workflows

Fig. 6 presents the BaaS-enabled workflows that tackle real-life problems:

1) *translate4Me* solves a real-life problem of translating a speech from one to another language, for which no BaaS service exists. *translate4Me* is a bag-of-task workflow, which implicitly translates audio files without explicit BaaS service for that purpose. Each audio file is transcribed into plain text by `S2T`, whose output is translated into target language by `TRA`, which is converted to natural sounding speech by `T2S`. With five audio files of 1 min, *translate4Me* runs 15 functions in total.
2) *read4me* is a fork-join workflow that creates natural-sounding speech in multiple languages from a PDF document, such that visually impaired people can "hear" documents instead of reading them. The function `split` cuts the input PDF file into single-page PDFs. In parallel for each such PDF, `OCR` extracts the text, which is translated into the target language by `TRA`, and then converted into natural-sounding speech by `T2S`. Finally, `merge` concatenates the audio files into a single file and stores it in storage. The workflow input contains a PDF with 5 pages stored in AWS or GCP storage, leading to a total of 17 functions.
3) *celebrityCollage* analyzes images (e.g., screenshots of a movie) and returns a collage of faces per each detected celebrity (actor), including information about the actor. *celebrityCollage* is a MapReduce-based workflow which identifies celebrities on various images. The map phase crops and groups the faces per celebrity, while in the reduce phase, the faces of the same celebrity are combined into a collage. Additionally, the reduce phase retrieves a summary text in English from Wikipedia for each celebrity, translates it in German, and stores it in cloud storage. More details for the workflow can be found in our recent paper [40]. For BaaSLess' evaluation, we used four values for the problem size of the first parallel loop, represented as the number of input images. The number of celebrities determines the size of the reduce phase. The problem sizes of 10 and 20 input images lead to 23 celebrities, while the problem sizes of 30 and 40 input images result in 34 celebrities for the second parallel loop.

Workflow diversity. The selected three workflows complement each other in many aspects. *read4me* represents a fork-join class of workflows, *translate4me* is a bag-of-task workflow, while *celebrityCollage* is a MapReduce-based workflow. Therefore, only *translate4me* is embarrassingly parallel workflow, while *read4me* can scale up to some problem size, after which `merge` will be much slower than the parallelism with the parallel loops implemented with the `parallelFor` construct of AFCL. Although both workflows run `TRA` and `T2S`, *translate4me*'s functions consume by up to 52 % larger inputs than the *read4me*'s ones. Finally, the types of invocations are also different. While all three BaaS services of *read4me* are invoked synchronously, *translate4me* calls *S2T* asynchronously, and the BaaSLess SDK uses busy waiting to inform the function when the BaaS service finishes, so that the workflow can continue with its execution immediately after the BaaS service finishes. *read4me* calls *OCR* by reference for the colocated BaaS service and storage to reduce the number of data transfers and by value otherwise. On the other side, *translate4me* calls the AWS *S2T* always by reference, even when input data is stored in GCP when the sync mechanisms are used to move data to AWS S3. While *read4me* and *translate4me* scale up to the providers' concurrency limit, for *celebrityCollage*, both the Map and Reduce phase go beyond the concurrency limit. We use the first two workflows to evaluate BaaSLess with low concurrency (Section VI), while *celebrityCollage* for high concurrency across multiple cloud regions (Section VII).

B. Workflow Deployments in Federated Infrastructures

Due to the high costs for BaaS services as they are managed, we separated our evaluation into two parts, still retaining the reliability of results. We executed the scheduled workflows six times, similar to other recent related work [38], [41]. We omitted the first execution to eliminate the cold start effect. Despite running six repetitions, functions in parallel loops are executed up to even 200 times for the *celebrityCollage* workflow.

1) Deployment Setup for Low Concurrency: We used a federated FaaS, BaaS, and storage testbed comprising AWS and GCP regions in North Virginia. We deployed all functions of the *read4me* and *translate4me* with the minimum possible memory of 128MB.

2) Deployment Setup for High Concurrency: For the experiments with high concurrency, the evaluation setup comprised

TABLE III
NETWORKING PARAMETERS FOR THE TESTBED FOR LOW CONCURRENCY

Setup	download		upload	
	L (ms)	B (MB/s)	L (ms)	B (MB/s)
VV-V	67	27.6	318	66.5
VV-V	125	58.5	253	45.6
VV-V	452	14.6	481	14.7
VV-V	296	17.3	366	15.9

TABLE IV
BAAS SERVICE EXECUTION TIME IN SECONDS

Setup	*translate4me*			*read4me*		
	S2T	*TRA*	*T2S*	*OCR*	*TRA*	*T2S*
-VV-	42.47	0.29	0.48	2.46	0.29	0.59
-VV-	18.87	0.08	1.09	1.16	0.03	1.53
-VV-	43.02	0.84	1.03	3.01	0.84	1.14
-VV-	19.15	0.26	1.37	1.44	0.31	1.81

four regions in Europe, i.e., AWS London and Frankfurt, along with GCP London and Belgium. All functions of *celebrityCollage* were deployed in all four regions and assigned with2 GB to differ from the smaller memory and the continent of the other two workflows. We selected memory larger than 1.5 GB in order to maximize bandwidth for the functions [42], [43].

C. Related Work Comparison With Federated Testbed

We compare BAASLESS with two state-of-the-art schedulers in federated FaaS. FaDO [27] and Seti et al. [34] move the code closer to data, i.e., colocate the function, BaaS service and data in a single provider region, thereby trying to minimize data access time. FaaSt [19], on the other side, uses the earliest finish time approach and prefers the function deployment that finishes earliest in federated FaaS. Unfortunately, this approach roams only the functions while it colocates the storage and BaaS services in the same provider region as the input data. Related work supports colocation and FaaS federation but not BaaS and storage federation across providers. Contrary, BAASLESS federates FaaS, BaaS, and storage, which increases the search space for optimal workflow deployment.

VI. EVALUATION FOR LOW SCALABILITY

This section evaluates BAASLESS scheduler for low scalability with the *translate4me* and *read4me* workflows.

A. Microbenchmarks

We first conducted a set of microbenchmarks to determine the federated testbed parameter setup. We ran the colocated workflows in the regions of North Virginia of AWS and GCP. We also measured BaaS service time (BT) from the function by adding timestamps before and after the call of the service. We also executed the same workflows but by migrating the functions to the other provider to determine the service time for those function deployments. Finally, we used linear regression functions to determine networking parameters L and B between the regions of both providers from all functions from their upload and download times. Tables III and IV show the determined parameter setup for the federated testbed and both BaaS-enabled workflows.

TABLE V
TRANSLATE4ME SCHEDULE: <IFBO> (INPUT STORAGE, FAAS, BAAS, OUT. STORAGE)

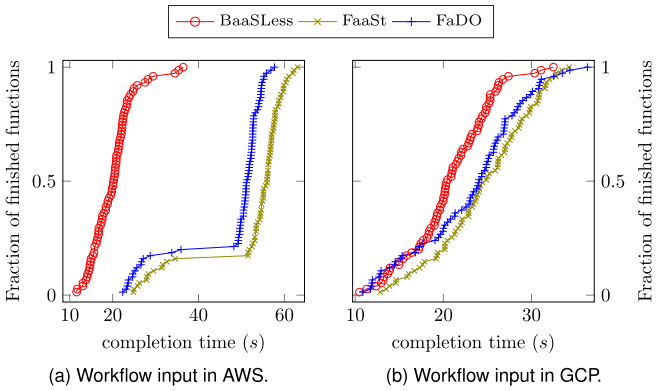
Function	workflow input on AWS S3			workflow input on GCP		
	BaaSLess	*FaaSt*	*FaDO*	*BaaSLess*	*FaaSt*	*FaDO*
S2T	VVVV	VVVV	VVVV	VVVV	VVVV	VVVV
TRA	VVVV	VVVV	VVVV	VVVV	VVVV	VVVV
T2S	VVVV	VVVV	VVVV	VVVV	VVVV	VVVV

GCP functions transfer files faster from and to GCP storage instead of S3 since VV-V dominates VV-V for both directions (lower latency, higher bandwidth). On the other hand, AWS functions download faster from GCP storage but upload larger files faster (Apply respective L and B for VV-V vs. VV-V in (1)). Another interesting observation is that BaaS services on both providers are invoked faster from AWS functions (VV-V vs VV-V in Table IV). The reason is the higher internal latency and lower bandwidth for VV-V compared to VV-V (Table III). Finally, by applying these values in (3), we determined the compute time CT for all workflow functions.

B. Evaluation With the translate4me Workflow

Using the federated testbed illustrated in Section V-B1 and the results of the microbenchmarks, we first evaluate the three schedulers with the *translate4me* workflow. Table V represents the 4-tuple **IFBO** <**I**nput storage, **F**aaS, **B**aaS service, **O**utput storage> for each function of *translate4me* achieved by the three evaluated schedulers BAASLESS, FaaSt, and FaDO, when workflow input is stored in AWS S3 and GCP storage.

Translate4Me schedules when workflow input is in AWS S3. FaDO colocates the function S2T on AWS Virginia, to call the AWS BaaS service *S2T*, and to store the output data in AWS S3, i.e., IFBO is VVVV. FaDO also uses the same schedules for the other two functions VVVV. Since the input data is in AWS, FaaSt searches from two IFBO schedules, i.e., VVVV and VVVV. FaaSt selects the former because VV-V dominates VV-V in Table III and -VV- dominates -VV- in Table IV. It is worth to note that FaaSt cannot find the faster schedule -VV- from Table IV because of lack of interoperability. However, FaaSt selects the schedule VVVV for the function TRA because the advantage of VV-V over VV-V in Table III and -VV- over -VV- in Table IV is smaller compared to faster function compute time and lower IT on GCP compared to AWS [38]. For T2S, FaaSt selects the colocated schedule VVVV, similarly to S2T. While FaaSt supports federated FaaS and finds a better schedule than the colocated FaDO, BAASLESS can schedule any of the three parts of FBO on any provider, selecting from 8 schedules (VVVV, VVVV,..., VVVV). Although the input data is in AWS S3, BAASLESS can find an even faster schedule VVVV for the function S2T, using its interoperability SDK. Furthermore, since the output data of S2T is scheduled on GCP, BAASLESS searches for a schedule for TRA such that the input data is stored in GCP. In this case, it is VVVV. Similarly, BAASLESS schedules the last function, T2S, with the same schedule as TRA.



(a) Workflow input in AWS. (b) Workflow input in GCP.

Fig. 7. Distribution of completion times for *translate4me*.

Translate4Me schedules when workflow input is in GCP storage. Using the same criteria as in the previous case, the three schedulers select the presented schedules on the right-hand side of Table V when the workflow input is in GCP. FaDO schedules all parts of IFBO on GCP because it colocates them with the storage. FaaSt selects the same colocated schedules because the functions with colocated storage finish faster on GCP. Once BAASLESS selects GCP storage for the output data of S2T, the subsequent functions are scheduled the same as when the workflow input is stored in S3.

Translate4Me's distribution of completion time when workflow input is in AWS S3. Fig. 7(a) presents the completion time of *translate4Me* functions scheduled by the three schedulers BAASLESS, FaaSt, and FaDO when workflow input is in AWS S3. We observe that all functions scheduled by BAASLESS finish earlier than the other two schedules. A huge benefit of the schedule VVVV for the function S2T can be seen compared to the FaaSt's and FaDO's schedule VVVV, which are the first twenty-five data points for BAASLESS. However, we observed strange behavior of the colocated AWS schedules VVVV for the function S2T. Although we measured a service time of 42.47 s, five calls to the AWS *S2T* BaaS service finished in the interval of only [22.04 s, 28.98 s] for the FaaSt schedule and even faster [19.68 s, 22.42 s] for FaDO, although they run the same schedule VVVV. We believe that this paradoxical behavior to run significantly faster is due to some caching and internal scheduling within AWS due to the spawn start [44]. Nevertheless, since the parallel loops finish when the last function finishes, these executions do not affect the overall workflow makespan. We observe similar gradients of all three curves for the next two functions because the functions run in a few seconds, and the benefit of BAASLESS is smaller compared to S2T. The BAASLESS scheduler achieved the lowest makespan of 27.68 s, ahead of FaDO, which completed within 55.79 s (2× slower), and FaaSt with 60.33 s (2.18× slower) when the workflow input resides in AWS S3. Note that although we expected FaaSt's schedule to run faster, FaDO's paradoxical executions were faster than FaaSt's, despite the schedules being the same.

Translate4Me's distribution of completion time when workflow input is in GCP storage. Similar conclusions can be derived from Fig. 7(b), which shows the completion time when the workflow input is stored in GCP storage. While the first 25 executions have similar completion times since all three schedulers run the same schedule VVVV, BAASLESS reported earlier finish time for the remaining executions because it found the better schedule VVVV compared to the colocated schedules VVVV. BAASLESS reported the lowest makespan of 27.64 s, which is comparable to the experiment with the input data on AWS S3. FaDO and FaaSt were slower by 15.97 % and 19.85 %.

TABLE VI
READ4ME SCHEDULE

Function	workflow input on AWS S3			workflow input on GCP		
	BaaSLess	*FaaSt*	*FaDO*	*BaaSLess*	*FaaSt*	*FaDO*
split	VV–V	VV–V	VV–V	VV–V	VV–V	VV–V
OCR	VVVV	VVVV	VVVV	VVVV	VVVV	VVVV
TRA	VVVV	VVVV	VVVV	VVVV	VVVV	VVVV
T2S	VVVV	VVVV	VVVV	VVVV	VVVV	VVVV
merge	VV–V	VV–V	VV–V	VV–V	VV–V	VV–V

<Ifbo>: (input storage, faas, baas, output storage).

C. Evaluation With the read4Me Workflow

Read4Me schedules when workflow input is in AWS S3. Table VI presents the schedules of the three evaluated schedulers when the workflow input of the *read4me* workflow is stored in AWS S3 and GCP storage. Since *read4me* is a fork-join workflow, it starts with the split function, which does not run a BaaS service. FaDO again colocates the function and the output storage. The same is true for FaaSt since GCP functions have higher latency to AWS S3 compared to AWS Lambdas. However, due to the lower latency for upload and download, BAASLESS schedules the output storage on GCP to achieve the earliest finish time. Furthermore, the schedules of the remaining functions coincide with *translate4me*'s schedule. Finally, similar to split, FaDO and FaaSt scheduled merge colocated VV-V, as it also does not run any BaaS service. Notably, BAASLESS scheduled the output to AWS S3 because the latency may be more important than the bandwidth when a function downloads five files, based on (1). This may be emphasized even more if the workflow is executed for a larger PDF file.

Read4Me schedules when workflow input is in GCP storage. FaDO schedules all parts of IFBO on GCP. FaaSt can find the faster schedule VV-V for the function split, which is the fastest of all eight schedules that BAASLESS evaluates. *read4me*'s functions that run BaaS services are scheduled similar to *translate4me*'s functions. Finally, for merge, FaDO colocates all parts of <IFBO> as for the other functions. FaaSt schedules the faster schedule VV-V in front of VV-V, while BAASLESS schedules the fastest schedule VV-V, which is only possible using BAASLESS SDK.

Read4Me's distribution of completion time when workflow input is in AWS S3. Fig. 8(a) presents the completion time of all functions of the *read4me* workflow, based on the schedules in Table VI, when workflow input is stored in AWS S3. While we do not observe a significant difference for the split function, the next three functions that run BaaS services finish earlier with the BAASLESS schedule. Overall, BAASLESS achieved the lowest

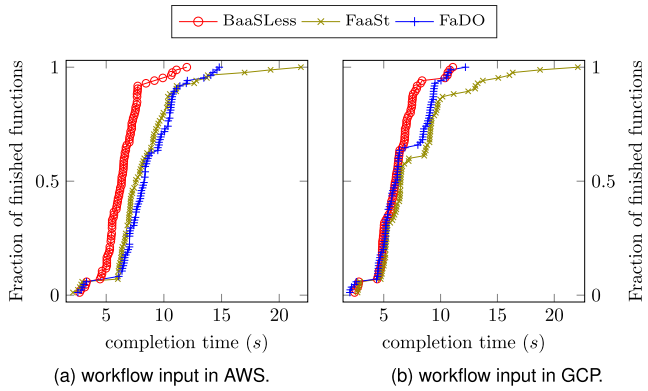


Fig. 8. Distribution of completion times for *read4me*.

makespan of 10.88 s, in front of FaDO's 14.24 s or 30.94 % longer, and FaaSt's 14.97 s, or 37.64 % slower. Notably, once again, FaaSt's schedule leads to a longer completion time. We analyzed the results in more detail and determined a huge outlier for one instance of the function TRA, whose download time was 5.9 s, or 13.66× longer than the other instances of *TRA* of that execution. Without this outlier, FaaSt achieves 13.24 s on average, which is faster than FaDO, as expected, due to the better schedules on the function TRA.

Read4Me's distribution of completion time when workflow input is in GCP. BAASLESS reported the lowest makespan for the experiments when the workflow input is in GCP, as well (Fig. 8(b)). The main improvements are observed for TRA and T2S. BAASLESS achieved similar makespan of 10.7 s as when the workflow input is in AWS S3 due to the similar schedule. FaDO was slower for 2.9 %, while FaaSt 45.7 %, on average. We also observed several outliers, this time for the function OCR for two executions with the FaaSt schedule. Note that although BAASLESS and FaDO had the same schedule, we did not observe outliers for either of them.

VII. EVALUATION FOR HIGH SCALABILITY

We analyze high concurrency based on the setup in Section V-B2. Since we use up to $R = 4$ regions and the minimum concurrency limitation of GCP's TRA is 10, we vary the problem size $N \in \{10, 20, 30, 40\}$. For both weak and strong scaling, we ordered the regions as follows: L, L, F, and F. L is the colocated region in which the input data is stored. L is geographically closest to L, followed by F, and F, which are closer to Innsbruck, from where we run the experiments. For easier readability, we will use the notation $< N/R >$.

A. Microbenchmarks

We ran the *celebrityCollage* workflow in the four evaluated regions in Europe with colocated storage to evaluate BAASLESS for weak scaling. Using our xAFCL serverless workflow management system [17], we measured the overall round trip time RTT of each function. With white-box testing, we additionally measured the BaaS service time BT for the functions detectCelebrities and Translate, as well as the transfer times DT and UT for all functions. Further on, we applied all these times in (3) and computed $CT + IT$. Finally, we applied these values in our SimLess simulator [38] to simulate the scheduled versions of all three workflows for weak and strong scaling.

TABLE VII
RELEVANT RTT DERIVED FROM THE MICROBENCHMARKS FOR THE FUNCTIONS cropFaces AND createCollage, WHICH ARE NEEDED FOR HIGH CONCURRENCY

Function	*FaDO*	*+FaaSt*			*+BaaSLess*						
	LL-L	LL-L	LF-L	LB-L	LL-L	LL-L	LB-L	LL-L	LL-L	LF-L	LB-L
crop	6.05	19.37	11.81	23.55	8.67	3.22	4.07	-	-	-	-
collage	2.58	7.46	7.26	13.46	-	-	-	6.93	2.44	6.11	2.87

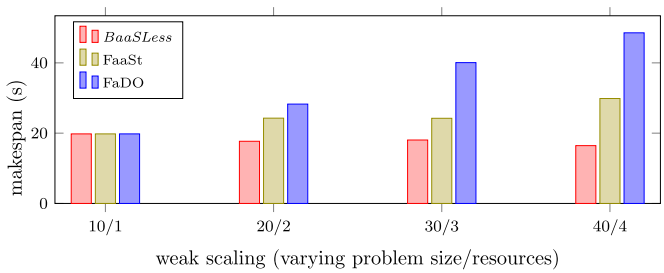


Fig. 9. Makespan achieved with BAASLESS, FaaSt, and FaDO schedulers for weak scaling experiments using *celebrityCollage*.

Table VII presents the calculated values of the relevant values for RTT for the two functions cropFaces and createCollage. Similar to low concurrency, FaDO colocates the functions, BaaS services, and output storage, leading to the single schedule LL-L. By utilizing federated FaaS, FaaSt gains access to three additional regions to schedule the functions, i.e., the schedules LL-L, LF-L, and LB-L. Notably, due to the lack of storage interoperability, FaaSt colocates the output storage as the input storage. As expected, the colocated function achieves the lowest round trip time. However, the second-best AWS Frankfurt is in front of GCP London, which is geographically closer. The reason is the 2× higher bandwidth for download and 1.67× lower latency to upload of LF-L compared to LL-L.

Apart from the four options that FaaSt uses, BAASLESS searches among up to $O(4^4)$ schedules (four parameters with four regions each) for each function. For simplicity, Table VII presents a selection of schedules that are relevant for the evaluation. For the function cropFaces, BAASLESS calculates two schedules LL-L and LB-L, which are faster than all four schedules that FaaSt can consider. The main reason is the $1.78\times - 2.12\times$ higher latency of LL-L compared to the other two cross-regional schedules. Similarly, BAASLESS finds three cross-regional schedules LL-L, LF-L, and LB-L, which achieve lower RTT than the respective colocated storages for the createCollage function. Notably, these three schedules are possible because the schedules LL-L and LB-L stored data in GCP storage in London.

B. Evaluation for Weak Scaling

Fig. 9 illustrates the makespan of all three evaluated schedulers for weak scaling. All three schedulers produce the same

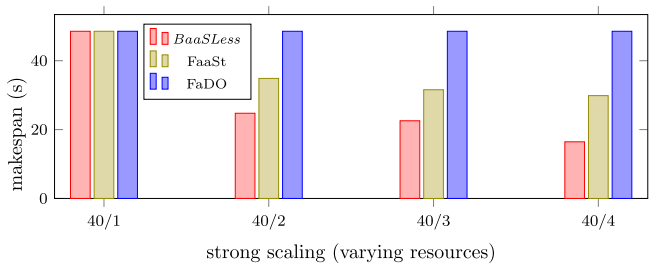


Fig. 10. Makespan achieved with BAASLESS, FaaSt, and FaDO schedulers for strong scaling experiments using *celebrityCollage*.

schedule when they schedule within a single region only, thereby they all achieve the same makespan of 19.79 s. However, for the other three experiments for weak scaling, BAASLESS' schedules are up to 1.82× and even 2.95× faster compared to FaaSt and FaDO, respectively. BAASLESS even reduces makespan to 16.45 s for the weak scaling $< 40/4 >$ compared to $< 10/1 >$, which represents a superlinear speedup [45]. The superlinear speedup is achieved because of the reduced data access time for the scaled experiments, which is supported by the interoperable BAASLESS' SDK. The main reason for the significantly higher makespan of FaDO's schedules is the lack of scalability. Namely, since the problem size is 40 images, but the total number of concurrent executions is restricted to 10, FaDO's schedule runs both parallel loops as a sequence of four parallel loops of ten iterations. FaaSt, on the other side, can utilize all four regions for the functions but not for the output storage.

The main advantage of BAASLESS is observed for the functions `cropFaces` and `createCollage`, which download images from and upload to the storage. Namely, starting from $< 20, 2 >$, BAASLESS can find the faster schedule LL-L to run the function and colocate the cropped faces (output) in GCP London, despite that the input data is stored in AWS London. This cross-regional setup needs 3.22 s instead of colocated LL-L, which needs 6.05 s. With it's federated storage, BAASLESS' $< 40, 4 >$ schedule runs the `cropFaces` function within $RTT = 6.44$ s, which is only 6.54 % longer than the baseline experiment $< 10, 1 >$. On the other side, FaaSt searches from four options only, all of which are scheduled to access the storage in AWS, thereby reaching enormous $RTT = 18.14$ s for $< 40, 4 >$. Similarly, for the `createCollage` function, FaaSt reduces RTT from 7.73 s to 7.46 s utilizing the federated FaaS aproach. However, BAASLESS finds an even faster schedule that achieves $5.74\,\text{s} = 2 \cdot 2.87\,\text{s}$.

C. Evaluation for Strong Scaling

Fig. 10 presents the evaluation of the three schedules using the SimLess simulator. Given that strong scaling always submits the maximum problem size of 40, FaDO always generates the same schedule on the single colocated region LL-L, which is the same as the other schedulers for the experiment $< 40, 1 >$, thereby all achieving the highest makespan of 48.55 s. However, BAASLESS and FaaSt reduce makespan for strong scaling, but BAASLESS overperforms FaaSt for all three experiments. For the experiment $< 40, 4 >$, BAASLESS reduces makespan by 2.95×, while FaaSt reduces makespan 1.63× compared to $< 40, 1 >$. The reasons for BAASLESS's better strong scaling are similar to weak scaling, i.e., running the same two functions *cropFaces* and *createCollage* with the BAASLESS' interoperable SDK.

VIII. DISCUSSION

A. Related Work

FaaS federation. FaaSt [19] is the state-of-the-art scheduler that distributes workflow functions across federated FaaS. While FaaSt leverages the computation resources of federated FaaS, it ignores the network proximity of storage and performance of BaaS services. XFaaS [46] and WISEFUSE [47] re-group the workflow by fusing a sequence of functions in a single function to reduce invocation latency. Additionally, both works determine the optimal placement of the fused functions in federated FaaS. While this approach may reduce data access time to storage and invocation delay, it introduces many other challenges, such as increased cost due to higher memory requirements, or longer duration because of larger package size. Moreover, many BaaS services use their `inVal` and `outVal` data ports, which require that the function loads the input data and stores the output data in storage. xAFCL [17] considers network proximity between functions within a parallel loop and scatters entire iterations in federated FaaS while FaDO [27] colocates functions with storages. Zion [48] creates pipelines and runs them in a dataflow manner. However, the evaluation showed that BaaS-enabled serverless workflows benefit not only from FaaS federation, but also from BaaS and storage federation.

Storage federation. Onedata [49] splits the files and places them in various storages in a multi-cloud environment, using improved block-based data transfer. Lithops [26] allows the sharing of files among multiple functions of a parallel loop. Locus [50] federates cheap but slow storage and fast but expensive storage to achieve a cost-performance tradeoff. While these systems allow global data access and benefit from storage federation, they do not support FaaS and BaaS.

BaaS composition and federation. In general, BaaS services are neglected by researchers. SLAM [15] profiles the call graph of serverless applications, including BaaS services, in order to determine the memory configuration. However, SLAM only profiles without replacing the faster BaaS service on another provider. The very recent paper ProPack [51] determined a huge overhead while scaling the number of functions that use compute-intensive BaaS services inside functions. The authors determine the trade-off between packaging and fewer functions, which reduces scalability but also concurrency overhead. We believe that a mixture of offloading the BaaS services or running locally inside the function will improve the performance.

QoS-aware placement. Serverless workflows enable the distribution of their functions across the edge and the cloud. However, users are challenged to determine the optimal setup that satisfies their requirements because the setup depends on both application structure and runtime environment. Sigurleifsson et al. [52] used a modular fuzzy analytical approach for multi-objective optimization, which considers data locality, cost, and performance. However, the authors model a pipeline (a sequence) of functions without considering a real DAG and

concurrency limitations, as required by BaaS-enabled serverless workflows. Courier [53] is a platform that delivers serverless functions across federated serverless infrastructures to optimize their execution time. However, it mainly works for isolated functions or functions that access to a single storage, while BaaSLess supports BaaS-enabled serverless workflows. Orthogonal to our work, Bocci et al. [54] introduced a placement technique that prevents information leakage through side channels for BaaS-enabled functions.

Dynamic scheduling. While BaaSLess scheduler uses an offline (static) scheduling algorithm, several online schedulers were recently introduced. StepConf [55] is a dynamic scheduler for serverless workflows, which utilizes intra- and inter-function parallelism to speed up the execution and minimize cost. Similar to BaaSLess, StepConf schedules the workflow functions step by step. However, its model supports isolated functions within a single cloud region without supporting BaaS-enabled serverless workflows. AsyFunc [56] utilizes the concept of lightweight shadow functions which take over part of computing within a burstable period. O-RDC [57] is an online scheduling algorithm that considers a trade-off between startup latency and resource utilization in the edge-cloud continuum, similar to a ski-rental problem. λDNN [58] is a cost-efficient serverless resource provisioning framework, which checkpoints the states of the functions in storage and loads them to avoid the maximum duration constraint. Unlike it, BaaSLess approach is to asynchronously offload the computing to BaaS services. Finally, λGNN [59] is a dynamic scheduler that builds a workflow from a Graph Neural Networks operations dynamically during runtime through sharing the graph and allocating the resources in a fine-grained manner. While all aforementioned approaches are appropriate in open-source platforms such as Knative or OpenWhisk, they are not applicable in public cloud providers as inter-function communication is forbidden.

B. Monetary Cost Consideration for Execution

While serverless computing introduces cloud resources that are fully managed by the cloud providers, it still comes with a higher cost per resource than the respective Infrastructure-as-a-Service, such as EC2 [60]. Liu and Niu [61] introduce a concept of future function, that is, an auction-based pricing model that offers discounts to the users but at the same time boosts profit for the providers. While this approach is suitable for FaaS, whose pricing model is based on invocation and duration price, the pricing model for BaaS service is based on the problem size, which is two magnitudes larger than the FaaS cost, and a magnitude higher than data transfers between cloud regions. On the other side, the prices for the same BaaS service offered by various providers do not significantly differ between providers.

While BaaSLess allows dynamic BaaS-enabled serverless workflow deployments during runtime to optimize execution time, it does not consider the potential increase of the costs due to external traffic between providers and potentially higher costs for the faster BaaS services or faster functions. However, we did not include the cost (See Table I) in our optimization because the only BaaS service deployment with higher cost (GCP *TRA*) is also slower than AWS *TRA* (see Tables V and VI). The costs for one execution of *read4me* and *translate4me* scheduled with BaaSLess are 55.7¢ and 26¢, respectively. These costs are mainly due to the charges for the BaaS services, from 99.1% for the *read4me* workflow with input data in GCP up to 99.997 % for the *translate4me* workflow with input data in GCP. Compared to the other two schedulers, BaaSLess reported up to 26 % lower costs for input data in GCP storage, but also up to 0.84 % higher costs when input data is stored in AWS S3. Still, these results may differ for other BaaS services.

C. Threats to Validity

To better identify BaaSLess limitations, we rely on the methodology introduced by Wohlin et al [62], who classify the validity tests in four categories: construct validity, internal validity, external validity, and reliability.

Construct validity. We minimized this class of threat by constructing our methodology with two widely-used cloud providers, four cloud regions, three complementary workflows, and we evaluated BaaSLess scheduler with two other approaches in federated serverless infrastructures.

Internal validity. We evaluated BaaSLess scheduler with a given problem size, including a constant number of input files with a constant file size within the range of megabytes. However, we used the simplified model for networking transfer time in (1), which ignores the TCP's slow-start [37]. For smaller files, TCP's slow-start may increase latency 10 to 15 times and reduce bandwidth by 20 %. Still, this threat evenly affects all evaluated schedulers.

External validity. We evaluated the BaaSLess scheduler with four regions of AWS and GCP. Due to different network proximity, the results may differ if we used other regions. However, we selected US regions to achieve higher network delay between University of Innsbruck, across the sea. Additionally, workflows with more complex structures and other BaaS services may not lead to the same conclusion.

Reliability. Our evaluation was conducted from the University of Innsbruck. However, experiments conducted from other locations in the world will lead to different results and therefore researchers must measure and adapt the parameters in all schedulers. We repeated our experiments five times to minimise the reliability threat, ignoring the cold start.

IX. Conclusion and Future Work

This paper introduced BaaSLess, a novel platform that minimizes the makespan of BaaS-enabled serverless workflows by providing and integrating a dynamic and globally federated BaaS and storage infrastructure across multiple cloud regions to workflow functions. To the best of our knowledge, BaaSLess is the first platform that exposes common APIs to the developer that hide the heterogeneity of SDKs for BaaS services. BaaSLess alleviates the vendor lock-in constraints with various sync primitives. Finally, the BaaSLess scheduler dynamically configures the BaaSLess SDK in each function to deploy the BaaS service and to attach the storage backend that minimizes the overall makespan.

BaaSLess improves the makespan by up to 2.18× compared to state-of-the-art schedulers, which are specifically designed

for federated serverless infrastructures. The speedup is initiated mainly by selecting the optimal deployment of BaaS services, attached storage locations and function deployments. Even higher speedup is achieved for strong and weak scaling, where BAASLESS achieves speedup of $3\times$ and $1.54\times$, respectively, when scaled with four cloud regions compared to the experiments with a single region. As a comparison, this leads to an improvement of $2.45\times$ compared to related work.

We will extend BAASLESS in four directions:

i) *BaaS service time and cost models:* We will investigate how runtime and cost are affected by input and output location and problem size (e.g., audio length), BaaS service implementation, and region, in order to create mathematical models that estimate performance and cost for all different setups;
ii) *Multi objective optimization:* we will extend the BAASLESS heuristics to automatically apply the mathematical models for cost and runtime of other BaaS services;
iii) *BaaS services composition:* We will extend our notion of BaaS-enabled workflows into *BaaS-workflows* by composing the BaaS services directly without FaaS latency; and
iv) Extend the BAASLESS SDK for other programming languages (e.g., Python and Node.js), BaaS services (e.g., sentiment analysis), and providers (e.g., Azure).

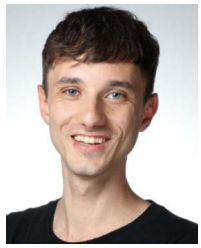
ACKNOWLEDGMENT

The master student Mika Hautz, affiliated with the University of Innsbruck, supported the authors in conducting the measurements for the *celebrityCollage* workflow.

REFERENCES

[1] I. Baldini et al., *Serverless Computing: Current Trends and Open Problems*. Berlin, Germany: Springer, 2017, pp. 1–20.
[2] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Serverless computing on heterogeneous computers," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Lausanne, Switzerland, 2022, pp. 797–813.
[3] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Virtual, USA, 2021, pp. 152–166.
[4] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," 2019, *arXiv: 1902.03383*.
[5] E. Paraskevoulakou and D. Kyriazis, "ML-FaaS: Toward exploiting the serverless paradigm to facilitate machine learning functions as a service," *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 3, pp. 2110–2123, Sep. 2023.
[6] S. Ristov, S. Pedratscher, and T. Fahringer, "AFCL: An abstract function choreography language for serverless workflow specification," *Future Gener. Comput. Syst.*, vol. 114, pp. 368–382, 2021.
[7] A. Arjona, P. G. López, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," *Future Gener. Comput. Syst.*, vol. 124, pp. 215–229, 2021.
[8] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proc. Symp. Cloud Comput.*, 2020, pp. 1–15.
[9] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless ERA," in *Proc. 21st Int. Middleware Conf.*, Delft, The Netherlands, 2020, pp. 280–295.
[10] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proc. Symp. Cloud Comput.*, 2020, pp. 311–327.
[11] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *Proc. 20th USENIX Symp. Networked Syst. Des. Implementation*, Boston, MA, USA, 2023, pp. 1489–1504.
[12] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019.
[13] S. Eismann et al., "Serverless applications: Why, when, and how?," *IEEE Softw.*, vol. 38, no. 1, pp. 32–39, Jan./Feb. 2021.
[14] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, 2022, Art. no. 10s.
[15] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt, "SLAM: SLO-aware memory optimization for serverless applications," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2022, pp. 30–39.
[16] T. Larcher and S. Ristov, "Scale composite baas services with AFCL workflows," in *Proc. Workshops Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, Denver, CO, USA, 2023, pp. 2033–2041.
[17] S. Ristov, S. Pedratscher, and T. Fahringer, "xAFCL: Run scalable function choreographies across multiple FaaS systems," *IEEE Trans. Services Comput.*, vol. 16, no. 1, pp. 711–723, Jan./Feb. 2023.
[18] Y. Babuji et al., "Federated function as a service for escience," in *Proc. IEEE 17th Int. Conf. eScience*, Los Alamitos, CA, USA, 2021, pp. 251–252.
[19] S. Ristov and P. Gritsch, "FaaSt: Optimize makespan of serverless workflows in federated commercial FaaS," in *Proc. Int. Conf. Cluster Comput.*, Heidelberg, Germany, 2022, pp. 182–194.
[20] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona, "Toward multicloud access transparency in serverless computing," *IEEE Soft.*, vol. 38, no. 1, pp. 68–74, Jan./Feb. 2021.
[21] Z. Li et al., "funcX: Federated function as a service for science," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 4948–4963, Dec. 2022.
[22] I. Stoica and S. Shenker, "From cloud computing to sky computing," in *Proc. Workshop Hot Topics Operating Syst.*, 2021, pp. 26–32.
[23] Z. Yang et al., "SkyPilot: An intercloud broker for sky computing," in *Proc. Symp. Networked Syst. Des. Implementation*, Boston, MA, USA, 2023, pp. 437–455.
[24] J. J. Durillo, R. Prodan, and J. G. Barbosa, "Pareto tradeoff scheduling of workflows on federated commercial clouds," *Simul. Modelling Pract. Theory*, vol. 58, pp. 95–111, 2015.
[25] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. Symp. Cloud Comput.*, Santa Clara, USA, 2017, pp. 445–451.
[26] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez, "Outsourcing data processing jobs with lithops," *IEEE Trans. Cloud Comput.*, vol. 11, no. 1, pp. 1026–1037, Jan.–March 2023.
[27] C. P. Smith, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict, "FaDO: FaaS functions and data orchestrator for multiple serverless edge-cloud clusters," in *Proc. IEEE 6th Int. Conf. Fog Edge Comput.*, 2022, pp. 17–25.
[28] S. Ristov, D. Kimovski, and T. Fahringer, "Faascinating resilience for serverless function choreographies in federated clouds," *IEEE Trans. Netw. Serv. Manage.*, vol. 19, no. 3, pp. 2440–2452, Sep. 2022.
[29] V. Yussupov, J. Soldani, U. Breitenbücher, and F. Leymann, "Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA," *Softw.: Pract. Experience*, vol. 52, no. 6, pp. 1454–1495, 2022.
[30] R. Chard et al., "FuncX: A federated function serving fabric for science," in *Proc. Int. Symp. High- Perform. Parallel Distrib. Comput.*, Stockholm, Sweden, 2020, pp. 65–76.
[31] S. Nastic et al., "Polaris scheduler: Edge sensitive and SLO aware workload scheduling in cloud-edge-IoT clusters," in *Proc. Int. Conf. Cloud Comput.*, 2021, pp. 206–216.
[32] T. Pusztai et al., "Polaris scheduler: SLO- and topology-aware microservices scheduling at the edge," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput.*, 2022, pp. 61–70.
[33] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer, "Serverless workflows for containerised applications in the cloud continuum," *J. Grid Comput.*, vol. 19, no. 3, pp. 1–18, 2021.
[34] B. Sethi, S. K. Addya, J. Bhutada, and S. K. Ghosh, "Shipping code towards data in an inter-region serverless environment to leverage latency," *J. Supercomput.*, vol. 79, no. 10, pp. 11585–11610, Mar. 2023.
[35] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google cloud functions," *Future Gener. Comput. Syst.*, vol. 110, pp. 502–514, 2020.

[36] T. Larcher and S. Ristov, "Profiling java serverless functions package size in federated FaaS," in *Proc. Int. Workshop Algorithms, Models Tools Parallel Comput. Heterogeneous Platforms*, Limassol, Cyprus, 2023, pp. 330–341.
[37] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *J. Parallel Distrib. Comput.*, vol. 74, no. 10, pp. 2899–2917, 2014.
[38] S. Ristov, M. Hautz, C. Hollaus, and R. Prodan, "SimLess: Simulate serverless workflows and their twins and siblings in federated FaaS," in *Proc. ACM Symp. Cloud Comput.*, San Francisco, CA, USA, 2022, pp. 323–339.
[39] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.
[40] S. Ristov, S. Brandacher, M. Hautz, M. Felderer, and R. Breu, "CODE: Code once, deploy everywhere serverless functions in federated FaaS," *Future Gener. Comput. Syst.*, vol. 160, pp. 442–456, 2024.
[41] H. Casanova et al., "Developing accurate and scalable simulators of production workflow management systems with WRENCH," *Future Gener. Comput. Syst.*, vol. 112, pp. 162–175, 2020.
[42] A. Wang et al., "InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache," in *Proc. 18th USENIX Conf. File Storage Technol.*, Santa Clara, CA, USA, 2020, pp. 267–281.
[43] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2018, pp. 133–145.
[44] S. Ristov, C. Hollaus, and M. Hautz, "Colder than the warm start and warmer than the cold start! experience the spawn start in FaaS providers," in *Proc. Workshop Adv. Tools, Program. Languages, Platforms Implementing Evaluating Algorithms Distrib. Syst.*, Salerno, Italy, 2022, pp. 35–39.
[45] S. Ristov, R. Prodan, M. Gusev, and K. Skala, "Superlinear speedup in HPC systems: Why and when?," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, Gdansk, Poland, 2016, pp. 889–898.
[46] A. Khochare, T. Khare, V. Kulkarni, and Y. Simmhan, "XFaaS: Cross-platform orchestration of FaaS workflows on hybrid clouds," in *Proc. IEEE/ACM Int. Symp. Cluster Cloud Internet Comput.*, 2023, pp. 498–512.
[47] A. Mahgoub et al., "WISEFUSE: Workload characterization and DAG transformation for serverless workflows," in *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, pp. 1–3, 2022.
[48] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in *Proc. Middleware Conf.*, Las Vegas, Nevada,USA, 2017, pp. 121–133.
[49] M. Orzechowski, M. Wrzeszcz, B. Kryza, Ł. Dutka, R. G. Słota, and J. Kitowski, "Global access to legacy data-sets in multi-cloud applications with onedata," in *Proc. Parallel Process. Appl. Math.: Int. Conf.*, Gdansk, Poland, 2023, pp. 305–317.
[50] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. Symp. Netw. Syst. Des. Implementation*, Boston, MA, USA, 2019, pp. 193–206.
[51] R. Basu Roy, T. Patel, R. Liew, Y. N. Babuji, R. Chard, and D. Tiwari, "ProPack: Executing concurrent serverless functions faster and cheaper," in *Proc. Int. Symp. High- Perform. Parallel Distrib. Comput.*, Orlando, FL, USA, 2023, pp. 211–224.
[52] B. Sigurleifsson, N. Ahmed, A. Verdet, M. Hamdaqa, M. Sabri, and I. Pelletier, "An approach for modeling the operational requirements of FaaS applications for optimal deployment," *Inf. Softw. Technol.*, vol. 161, 2023, Art. no. 107242.
[53] A. Jindal, J. Frielinghaus, M. Chadha, and M. Gerndt, "Courier: Delivering serverless functions within heterogeneous FaaS deployments," in *Proc. 14th IEEE/ACM Int. Conf. Utility Cloud Comput.*, Leicester, U.K., 2021, pp. 1–10.
[54] A. Bocci, S. Forti, G.-L. Ferrari, and A. Brogi, "Declarative secure placement of FaaS orchestrations in the cloud-edge continuum," *Electronics*, vol. 12, no. 6, 2023, Art. no. 1332.
[55] Z. Wen, Y. Wang, and F. Liu, "StepConf: SLO-aware dynamic resource configuration for serverless function workflows," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1868–1877.
[56] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "AsyFunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *Proc. ACM Symp. Cloud Comput.*, Santa Cruz, CA, USA, 2023, pp. 324–340.
[57] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1069–1078.
[58] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, "DNN: Achieving predictable distributed DNN training with serverless architectures," *IEEE Trans. Comput.*, vol. 71, no. 2, pp. 450–463, Feb. 2022.
[59] H. Hu, F. Liu, Q. Pei, Y. Yuan, Z. Xu, and L. Wang, "Grapher: A resource-efficient serverless system for GNN serving through graph sharing," in *Proc. ACM Web Conf.*, Singapore, 2024, pp. 2826–2835.
[60] A. Eivy and J. Weinman, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 6–12, Mar./Apr. 2017.
[61] F. Liu and Y. Niu, "Demystifying the cost of serverless computing: Towards a win-win deal," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 1, pp. 59–72, Jan. 2024.
[62] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.



Thomas Larcher received the bachelor degree in computer science from the University of Innsbruck, Austria, in 2021. He is currently working toward the master's degree in software engineering with the University of Innsbruck, Austria. His main research interests include serverless computing, performance modeling, optimization, and cloud federation.

Philipp Gritsch is currently working toward the PhD degree with the University of Innsbruck, Austria. His research interests include performance modeling and optimization of distributed systems, in particular workflow applications and serverless computing.

Stefan Nastic (Member, IEEE) is an assistant professor with TU Wien, Austria. He is also a founder and managing director of IntelliEdge GmbH. His research interests include serverless computing, edge-cloud continuum, AI and edge AI, and reliability engineering. He has a track record as a lead researcher, consultant, and technical coordinator, working on various research and commercial projects for over a decade. Nastic holds a PhD in programming, provisioning, and governing IoT cloud systems from TU Wien, Austria.

Sashko Ristov received the PhD degree in computer science from Ss. Cyril and Methodius University, Skopje, North Macedonia. He is an assistant professor with the University of Innsbruck, Austria. His main research interests include serverless computing, cloud engineering, and cloud federation. He received the IEEE Cloud Summit best paper award, in 2022.