Future Generation Computer Systems 160 (2024) 442–456

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

CODE: Code once, deploy everywhere serverless functions in federated FaaS

Sashko Ristov [a,*], Simon Brandacher [a], Mika Hautz [a], Michael Felderer [a,b,c], Ruth Breu [a]

[a] University of Innsbruck, Department of Computer Science, Innrain 52, 6020, Innsbruck, Austria
[b] German Aerospace Center (DLR), Institute for Software Technology, Linder Hoehe, 51147, Cologne, Germany
[c] University of Cologne, Department of Mathematics and Computer Science, Albertus Magnus Platz, 50923, Cologne, Germany

ARTICLE INFO

Keywords:
Automation
Domain specific language
Function-as-a-Service
Infrastructure-as-a-Code
Interoperability
Portability

ABSTRACT

Infrastructure-as-Code (IaC) frameworks empower developers to swiftly define and provision their infrastructure with a single click. However, the domain-specific languages (DSLs) utilized for coding the infrastructure often lean towards provider specificity rather than being application-centric. This results in increased developer effort, as they are compelled to duplicate data when deploying serverless functions across diverse regions and providers within federated FaaS environments. To mitigate this challenge, we introduce *CODE*, a framework engineered to streamline the deployment of functions in federated FaaS settings. *CODE* facilitates automatic deployment directly from the storage of any provider, eliminating the need for additional development effort to upload or copy deployment packages between disparate providers. Aligned with the guiding principle of "code once, deploy everywhere", *CODE* adopts a three-level hierarchy: function → providers → cloud regions. This architectural approach dramatically reduces the lines of code (LoC) in IaC scripts by up to 9.23× when contrasted with prevailing IaC frameworks such as Terraform and Serverless Framework. Additionally, *CODE*'s unified storage interface slashes LoC by up to 81.8%, both within *CODE* itself and when coding functions that use storage from providers such as AWS and GCP. In our comprehensive evaluation, we assessed the correlation between deployment package size and deployment time for various functions within a real-world serverless workflow across four regions of AWS and GCP. Our findings indicate that AWS deployment packages are significantly larger, often in the tens of megabytes, compared to GCP. Despite the larger size, AWS deploys these packages up to 6× faster than GCP.

1. Introduction

Function-as-a-Service (FaaS) has emerged as a prominent platform for scientific computing [1] and edge [2] and cloud-based data analytics [3]. Typically orchestrated in the form of serverless workflows [4], serverless functions are executed using specialized serverless workflow management systems. Some of these systems are tailored for event-driven workflows [5], while others focus on real-time data processing using data flow streams [6]. Additional systems, like xAFCL [7], facilitate complex batch processing across multiple cloud *regions* and *providers*, establishing a concept known as *federated FaaS* [8]. This federation often circumvents the limitations inherent in a single region of a provider, such as the cap on concurrent function executions (e.g., up to 1,000 concurrent functions in a single region [3, 7]), and ensures resilience against massive failures in a single region [9]. In Section 2, utilizing two representative serverless workflows, we illustrate that even with low concurrency, serverless workflows may experience considerably longer runtimes. Furthermore, we demonstrate that FaaS federation can significantly reduce the makespan of data-intensive serverless workflows, particularly when input data is distributed globally.

State-of-the-art approaches. While many serverless workflow management systems may run workflows across multiple FaaS providers [5, 7,10,11], several challenging steps must be performed beforehand to prepare the setup for execution. These steps include *coding* the functions for each specific provider, *packaging* the code with dependencies into a *deployment package* [12], and *deploying* it across multiple regions of various providers in a federated FaaS environment. Two distinct approaches exist to automate deployment. First, state-of-the-art frameworks like Terraform[1] and Serverless Framework[2] automate deployment across various providers. These frameworks, known as Infrastructure-as-a-Code (*IaC*), require developers to write scripts to

* Corresponding author.
E-mail addresses: sashko.ristov@uibk.ac.at (S. Ristov), simon.brandacher@student.uibk.ac.at (S. Brandacher), mika.hautz@student.uibk.ac.at (M. Hautz), michael.felderer@dlr.de (M. Felderer), ruth.breu@uibk.ac.at (R. Breu).

[1] https://www.terraform.io/
[2] https://www.serverless.com/framework

https://doi.org/10.1016/j.future.2024.06.017
Received 7 July 2023; Received in revised form 3 June 2024; Accepted 9 June 2024
Available online 10 June 2024

define their infrastructure and then run the framework to provision it. IaC scripts typically need to specify at least the function name, provider, region, assigned memory, timeout (maximum duration), and runtime. They may also include additional provider-specific configurations, such as security roles on AWS, project names on GCP, or namespaces on IBM. Another type of framework supporting all three steps – coding, packaging, and deploying – are *FaaSifiers*, such as Node2FaaS [13] and M2FaaS [14]. These tools convert parts of monolithic applications into functions, packaging all dependencies into deployment packages on the local file system before deploying the functions across different providers. While IaC frameworks allow deployment packages to be stored in provider storage, FaaSifiers extract code from monolithic applications, package it with its dependencies locally, and deploy it from there.

State-of-the-art limitations and challenges. More than 60 % of functions rely on managed cloud services, such as storage [15], which increases their deployment package size. For these functions, developers must upload the deployment package to the storages of the target providers, often to each target region, increasing the manual effort required to copy the package to all necessary locations. Our initial investigation of IaC frameworks revealed additional challenges in deploying a function across multiple regions of different providers in federated FaaS environments. While IaC frameworks simplify deployment by allowing infrastructure to be defined using a domain specific language (DSL), these DSLs are primarily provider-centric. IaC frameworks assume that functions will be deployed in a single region of a single provider. This hierarchical structure of *provider → region → function* necessitates redundant coding of function parameters for each target region. Given that top providers like AWS and GCP have more than 30 regions each, scripting deployments for multiple regions can become a tedious and time-consuming task, especially since serverless workflows often consist of multiple functions.

Code contribution. This paper introduces *CODE*, a framework designed to overcome the limitations of existing IaC frameworks and facilitate the deployment of functions in federated FaaS environments with minimum effort. *CODE* builds upon our recent work, GoDeploy [16],[3] which employs a three-level hierarchy placing the function at the highest level, followed by providers and their respective regions. *CODE* extends the capabilities of our *GoDeploy* deployer with the integration of another Go library *GoStorage*. *GoStorage* offers a unified set of APIs for provisioning, accessing, and interacting with AWS and GCP storage. This abstraction of different cloud storage solutions allows developers to use a single command to upload deployment packages or copy them between storages of different providers, with dynamic selection of source and destination storage providers. This approach significantly reduces the time required to learn various provider SDKs and the lines of code (LoC) that need to be written. Moreover, as a standalone library, GoStorage also reduces the LoC for functions that use it to copy files between any storage region of AWS S3 and GCP Storage.

Paper contributions. Apart from the extension with the *GoStorage* library, this paper contributes significant insights and comprehensive evaluations that are valuable for the community. First, we introduce and evaluate two serverless workflows that benefit from federated FaaS. Second, we conducted an evaluation of *CODE* against Serverless Framework, Terraform, and the recent M2FaaS FaaSifier. Third, for each framework, we derived a general model for the required LoC and assessed them across various federated FaaS scenarios, scaling the number of providers and their regions across continents. Lastly, we examined several parameters influencing the deployment of functions using *CODE*. Overall, the paper contributions include:

[3] This paper achieved the best paper award.

- Publicly Available Framework in Go with *GoDeploy*[4] and *GoStorage*[5] repositories, providing tools for seamless deployment in federated FaaS environments;
- Publicly available serverless workflow `celebrityCollage`,[6] a data-intensive, MapReduce-based workflow;
- *CODE* requires up to 9.23× less LoC compared to Terraform and 8.44× less LoC compared to Serverless Framework for coding the deployment of a function across all regions of AWS and GCP;
- *CODE* introduces `copy(srcURI, destURI)`, a single command enabling dynamic selection of storage providers without needing to rewrite or redeploy functions;
- The *GoStorage* library reduces LoC by up to 81.8 % compared to using individual provider SDKs; and
- Despite AWS deployment packages being significantly larger (by an order of magnitude) compared to GCP, AWS deploys these packages up to 6× faster. This was observed for both Go and Python functions.

Paper outline. The rest of the paper is structured into five sections. Section 2 presents the motivating use cases for deploying functions of a workflow application across multiple regions of various providers and addresses the need for high-frequency deployment in federated FaaS environments. In Section 3, we describe the overall architecture and implementation details of *CODE* and its two libraries *GoDeploy* and *GoStorage*, including their DSL and developer APIs for storage provisioning, access, and interaction. The results of the evaluation with three state-of-the-art frameworks, as well as the impact of different deployment package size and deployment time are discussed in Section 4. Section 5 discusses related work in automatic deployment of functions and how *CODE* advances beyond state-of-the-art approaches, as well as *CODE* limitations and threats to validity. Finally, Section 6 concludes the paper and presents several research areas that this paper opens for future work.

2. Motivating use cases for deployment of serverless workflows in federated FaaS with high frequency

This section outlines the numerous benefits that federated FaaS brings to serverless workflows. To demonstrate these advantages, we conducted a series of preliminary experiments using two motivating use cases - data-intensive serverless workflows (Fig. 1). These experiments highlight the constraints imposed by individual providers when operating within a single region, emphasizing the necessity of a federated FaaS environment. Additionally, we report the benefits of distributing computing tasks across cloud regions to efficiently process distributed data. Finally, we discuss the need to deploy serverless workflows in federated FaaS with high frequency.

2.1. Use case 1: scalable serverless workflows suffer from spawn start and concurrency overhead

Montage [17] is widely used scientific workflow by many researchers [5,6,18–20]. Details for Montage can be found in our recent paper [21]. We are interested in the three parallel loops `mProjectPPs`, `mDiffFits`, and `mBackgrnds` (see Fig. 1(a)). We used two versions to show the deviation in performance when scaling the problem size in a single region. The version 0.25 runs 30, 141, and 30 instances in the parallel loops, while the version 2.0 738, 2,095, and 430, respectively. We deployed Montage in AWS and GCP Frankfurt with 2 GB RAM for each function of the parallel loops. With this setup we minimize network instability by avoiding multiple functions to

[4] https://github.com/FaaSTools/GoDeploy
[5] https://github.com/FaaSTools/GoStorage
[6] https://github.com/AFCLWorkflows/celebrityDetection

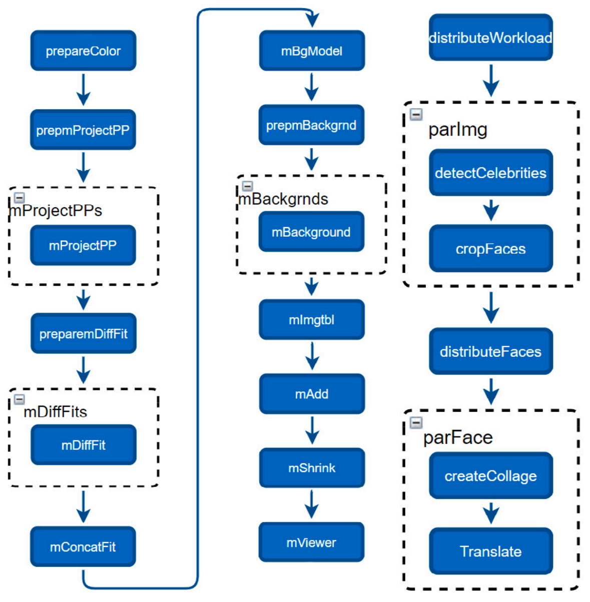

Fig. 1. Motivating serverless workflows. (a) Montage is a scientific workflow used in astronomy for processing and analyzing astronomical images and (b) `celebrityCollage` is a data-intensive workflow based on MapReduce technology, which recognizes celebrity faces in input images and generates a separate collage for each identified celebrity, accompanied by a translated summary.

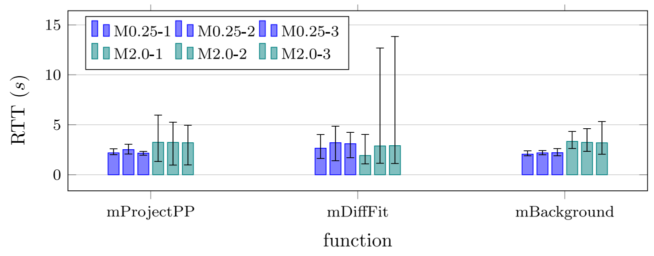

Fig. 2. Preliminary evaluation of Montage 0.25 and 2.0 versions on AWS. Each function `mProjectPP`, `mDiffFit`, and `mBackground` is nested in a parallel loop.

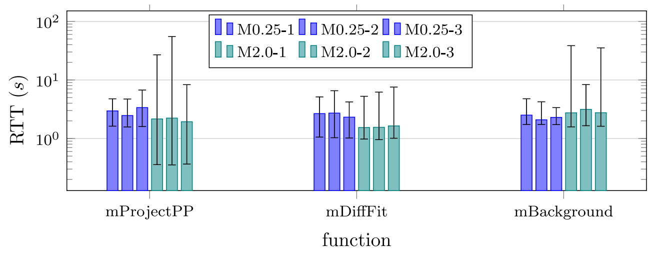

Fig. 3. Preliminary evaluation of Montage 0.25 and 2.0 versions on GCP. Each function `mProjectPP`, `mDiffFit`, and `mBackground` is nested in a parallel loop.

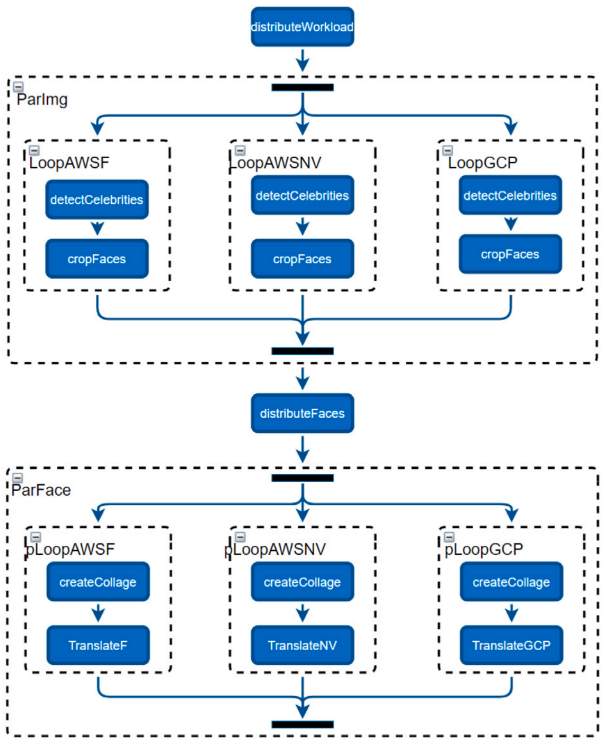

Fig. 4. The transformed `celebrityCollage` serverless workflow to run in a federated FaaS environment. Functions nested in parallel loops must be deployed across multiple regions to achieve scalability. The parallel constructs `ParImg` and `ParFace` can have an arbitrary number of sections, for example, 68 for each AWS and GCP region.

share the underlying VM [22]. We executed both versions of Montage four times and ignored the first execution due to the cold start. The workflow on AWS was executed with concurrency limit of 1,000, while on GCP with 100 to avoid the failures that we observed for higher number of concurrent functions [7]. Figs. 2 and 3 illustrate the average, minimum and maximum values of the round trip time RTT when the functions `mProjectPP`, `mDiffFit`, and `mBackground` of the parallel loops run on AWS and GCP, respectively.

Observation 1: FaaS elasticity does not automatically provide scalability. The parallel loops of Montage 2.0 run by up to 9.59× slower on AWS and by up to 30.23× slower on GCP, compared to the version 0.25. The main reason are concurrency overhead and spawn start when spawning numerous functions [23–25] and not the bandwidth constraints between the functions and colocated storage.

2.2. Use case 2: Data-intensive workflows with distributed input data

`Celebritycollage` workflow. We additionally developed and benchmarked a data intensive `celebrityCollage` serverless workflow (Fig. 1(b)), which processes images that are scattered across multiple cloud regions. `celebrityCollage` is a MapReduce-based workflow, which contains two parallel loops to speed up image processing. The first parallel loop `parImg` iterates over all images to group the cropped faces in a separate folder per celebrity, while the second parallel loop `parFace` merges all images of each celebrity in a collage with a summary text translated in German.

`Celebritycollage` adaptation for running in federated faas. We adopted approaches from recent research [26,27] to distribute the work across regions closer to the input data. Additionally, we transfer intermediary data generated during the mapper phase (`cropFaces`) across regions, which can significantly reduce data transfer time. To achieve such scalability, the workflow must be transformed so that functions nested in parallel loops are deployed across different regions. Fig. 4 illustrates an implementation of `celebrityCollage` designed to distribute the workload across two AWS regions and one GCP region.

Experiment. We used 200 input images distributed evenly, with 100 in AWS S3 Frankfurt and 100 in AWS S3 North Virginia. Each input image is 5 MB and contains faces of 8 celebrities. Celebrities with names starting from `A` to `M`, have their cropped faces stored in AWS S3 Frankfurt, while those with names from `N` to `Z` have their cropped faces stored in

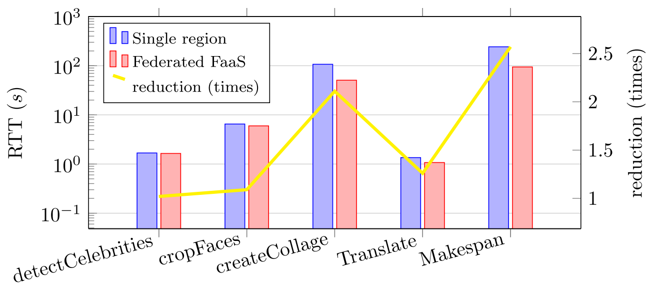

Fig. 5. Preliminary evaluation of the `celebrityCollage` workflow that processes 100 images stored in AWS S3 North Virginia and 100 images stored in AWS S3 Frankfurt. Single region denotes all functions run exclusively in AWS North Virginia. Federated FaaS denotes that functions are distributed across both regions, running closer to where the images are stored. The latter significantly reduces data access time, leading to 2.57× lower makespan.

AWS North Virginia. Two implementations of the `celebrityCollage` workflow were executed. In the first implementation, all 200 functions ran in a single region - AWS Frankfurt. In the second implementation, functions were colocated with the images: 100 functions in AWS Frankfurt and 100 in AWS North Virginia. This latter implementation follows a MapReduce-based approach, where the mapper functions `cropFaces` map the faces based on the celebrity name. The reducer functions `createCollage` and `Translate` then work locally with the colocated storage. To optimize throughput, we set the concurrency of both loops to 25 in accordance with the throughput limits of AWS Rekognition and AWS Translate, which restrict the number of requests per second.

Observation 2: Data-intensive serverless workflows with distributed input data benefit from federated FaaS as it reduces data access time. Fig. 5 presents the *RTT* of the four functions that are part of the `parallel` loops, as well as the overall makespan for both implementations of the `celebrityCollage` workflow. We observe that all four functions benefit from federation, which reduced the overall makespan by 2.57×, despite using the same amount of resources. The primary reason for this improvement is the reduced access time due to the colocation of functions with the input and intermediary data.

2.3. Other benefits from federated FaaS

Apart from addressing the challenges discussed in the previous two sections, federated clouds offer various benefits to users. Serverless workflow applications execute computations and download data faster on AWS, while data downloads are quicker on GCP [21]. By employing single-objective [28] or multi-objective [29] optimization, users can choose to run their workflows either more cost-effectively or faster. Additionally, the recently introduced concept of Sky computing [30,31] enables users to move large datasets between cloud providers and still process the data more cheaply, despite the high costs associated with transferring data out of the source cloud provider.

Federated FaaS may overcome the provider concurrency limitation of running 1,000 concurrent functions per user in a single region.[7] While the cloud users can extend the concurrency limit,[8] it remains restricted to a few thousand concurrent functions, which is often insufficient for the required scalability of, for example, 800,000 tasks [18]. Many researchers reported significant speedup by exploiting multiple regions, either by distributing the serverless workflow functions across several regions of multiple providers [7,32], or by offloading execution from edge to cloud providers whenever edge resources are overloaded [33], or by distributing files across storages and collocate functions closer to the data [26].

[7] https://cloud.google.com/functions/docs/configuring/concurrency
[8] https://repost.aws/knowledge-center/lambda-concurrency-limit-increase

2.4. The motivation for multiple deployments of a single function with high deployment frequency in federated FaaS

The deviceless edge computing approach introduces vertical offloading of the computing to more powerful edge servers and horizontal offloading on other similar edge devices [34]. Given that both edge servers and edge devices have limited capacity [35–37], functions must be dynamically redeployed after existing ones are deleted, necessitating adjustments to IaC deployment scripts.

Microservice architecture is essentially an architectural style that can be implemented with serverless functions [38]. The numerous microservices are often orchestrated using state machines [39] and deployed as serverless workflows [40]. Since one microservice may comprise one or multiple functions, DevOps engineers must create adapted IaC scripts for deploying these functions in federated FaaS for each subset of updated microservices. Some reports[9] indicate that certain companies perform up to 23,000 deployments per day. The challenges mentioned above necessitate multiple deployments of the same function across different regions and memory setups. These deployments are referred to as "twins" and "siblings", respectively. [24]. Unfortunately, such deployments generate significant overhead in IaC because users must redundantly specify information about the same function multiple times. The evaluation results in Section 4.2 demonstrate that *CODE* saves up to 9.23× the LoC compared to state-of-the-art IaC frameworks.

3. *CODE* system architecture

This section presents the *CODE*'s system architecture and implementation details for its two main modules, *GoStorage* and *GoDeploy*. Additionally, it outlines the *CODE*'s programming model and introduces a novel approach to IaC DSL. Unlike traditional IaC tools such as Terraform and Serverless Framework, which typically prioritize the provider at the top of their hierarchy, this new approach places the function at the apex of a three-level hierarchy.

3.1. CODE programming model

CODE uses the design principle "*code once, deploy everywhere*", which raises several requirements. *CODE* recommends that the function is developed as a local function or method only once (e.g., `DoWork()`), which is then called from various handler methods, separately for each provider. With this programming model, the users need to create a single deployment package for federated FaaS and then simply specify the specific handler method for the target provider during the deployment. Afterwards, developers should bundle all dependencies in a deployment package as a zip file so that the function can run properly, regardless of the target provider and region. Note that sometimes the developers may need to build a jar instead of a zip, or a container, and store them on the storage of the target provider.

Once a developer prepares the deployment package, including all dependencies, they need to use the *CODE* DSL to configure where and how to deploy the function. The developer utilizes *CODE*'s convenient hierarchy - *function* → *provider* → *region* - to specify multiple regions for each provider with minimum development effort for each function.

3.2. CODE system architecture

Fig. 6 presents the system architecture of *CODE*, which comprises two modules *GoDeploy* and *GoStorage*. The former introduces a CLI, DSL, and federated deployer, while the latter provides a set of developer APIs for federated storage infrastructure.

[9] https://www.altexsoft.com/blog/dataops-essentials/

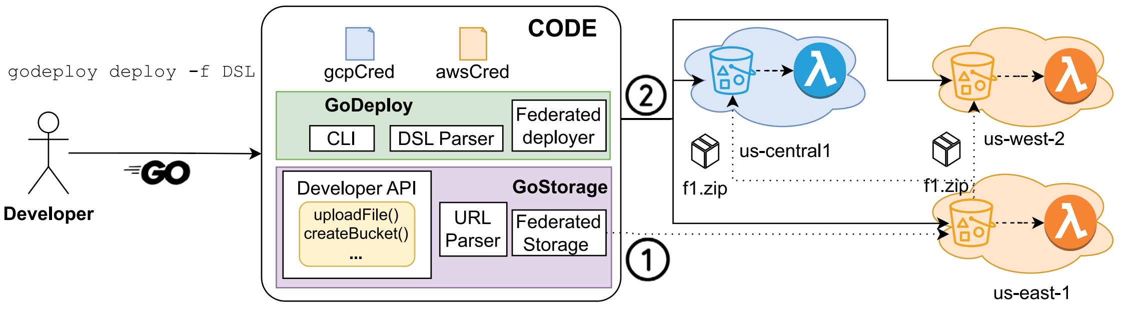

Fig. 6. *CODE* architecture comprising two modules (i) *GoStorage*, which introduces a federated storage interface designed to manage the background copying of function deployment packages across storage systems in target regions. GoStorage can also function independently as a library, enabling functions to dynamically select source and target cloud storage; and (ii) *GoDeploy* which deploys all packages copied by GoStorage into a federated Function-as-a-Service (FaaS) environment.

```
type CredentialsHolder struct {
    AwsCredentials    *aws.Credentials
    GoogleCredentials *google.Credentials
}
```

Fig. 7. `CredentialsHolder` definition.

3.2.1. *GoDeploy*

CLI. *CODE* requires credentials for each provider to be stored in separate files in the current working directory. The current implementation supports two providers AWS and GCP. A custom `CredentialsHolder` type needs to be created to store the credentials necessary for invoking operations via the corresponding SDKs, as defined in Fig. 7. This type stores pointers to provider-specific credential implementations, which are then used to configure the SDK clients for supported FaaS providers. After setting up the credentials, the developer can deploy functions using a single `deploy` command as an argument in the YAML-based DSL input file.

DSL parser. This module parses the YAML-based configuration file with the three-level hierarchy to determine the configuration setup for all functions to be deployed in federated FaaS. This includes details such as the function name, the providers where the function should be deployed, the regions for each provider, the amount of memory to be assigned, the maximum allowed duration of the function, the runtime environment, and the handler method for each provider. The developer can specify a single location for the deployment package (*f1.zip*) of a function for all deployments, across different providers. This package can be stored on the local file system or any storage service of the supported providers. For instance, the deployment package *f1.zip* can be stored on S3 in AWS North Virginia (*us-east-1*). The details of the DSL are presented in Section 3.4.1.

Federated deployer. Based on the parsed provider configuration, the federated deployer deploys the function in all specified regions where the deployment package is stored. For example, if `f1.zip` is already stored on AWS S3 in North Virginia, the federated deployer deploys it immediately. However, if `f1.zip` is not present in AWS S3 in `us-west-2` (Oregon) and in GCP storage in `us-central1` (North Virginia), the federated deployer uses the *GoStorage* module to copy the deployment package `f1.zip` to those cloud storages in the background, transparently to the user (step 1 in Fig. 6). Once the copy is complete, the federated deployer utilizes the specific SDKs for each provider to deploy the function in those two regions (step 2). This approach significantly reduces the development effort required to copy deployment packages of a single function to each region of federated FaaS environment.

3.2.2. *GoStorage*

Developer API. *GoStorage* offers a set of developer APIs for access and provisioning the storage on AWS and GCP, such as `uploadFile`, `downloadFile`, or `createBucket`. *APIs* do not require any change of the code and redeployment to be able to replace the storage. Implementation details and the usage of the developer APIs are presented in Section 3.3.1.

URL parser. The developer APIs of *GoStorage* use the URI parser to determine the SDK of the corresponding provider. Table 1 presents the supported URIs for AWS and GCP. The URI parser identifies the provider and its region from the provided URI for the deployment package of each function. *GoStorage* uses `URI 1` from Table 1 for AWS URLs because it contains the AWS region of the bucket, along with the bucket name and the key (the file). In contrast, GCP does not require developers to specify the region as it supports multi-region. Implementation details and the usage of the URI parser are presented in Section 3.3.2.

Federated storage interface. One of the main innovations of *CODE* over state-of-the-art IaC frameworks is its ability to copy the deployment package from any storage to all target regions and providers in the background. For instance, in the example shown in Fig. 6, if the developer specifies deploying a function in AWS' *us-west-2* (Oregon) and GCP's *us-central1* (North Virginia), the federated storage interface automatically copies the deployment package to the storages in both regions.

3.3. *GoStorage software architecture*

The *GoStorage* module abstracts the functionalities of different providers using the `Provider` interface, as defined in Fig. 8. It serves as a stand-alone library usable by any Go application, including functions, enabling dynamic selection of target storages supported by various cloud providers. Currently, *GoStorage* implements the `Provider` interface for two providers AWS and GCP. However, extending support to other providers is straightforward by implementing the `Provider` interface with the corresponding SDK of those providers. *CODE* primarily utilizes `copyFileWithinProvider()` and `uploadFile()` of *GoStorage* for copying the deployment package to another target storage and uploading the deployment package to the target storage, respectively. Additionally, other methods provided by *GoStorage* can be leveraged by software practitioners for managing storages across various providers. For instance, users can create a new bucket, copy the deployment package of a function to that bucket, deploy the function from that bucket, delete the function deployment package, and finally, delete the bucket of the target storage for each provider.

Most methods of the `Provider` interface utilize one or two objects of the `GoStorageObject` type, a custom object type designed to

Table 1
Supported URIs for AWS and GCP storages.

URI	Provider	Scheme	Authority	Path
1	AWS	`https://`	`s3.region-code.amazonaws.com`	`/bucket-name/key-name`
2	GCP	`https://`	`storage.cloud.google.com`	`/bucket-name/key-name`
3	GCP	`http://`	`storage.cloud.google.com`	`/bucket-name/key-name`
4	GCP	`gs://`		`/bucket-name/key-name`

```
type Provider interface {
  createBucket(bucketName string, region string)
  deleteBucket(target GoStorageObject, deleteIfNotEmpty bool)

  copyFileWithinProvider(source GoStorageObject, target GoStorageObject)
  copyBucketWithinProvider(source GoStorageObject, target GoStorageObject)

  uploadFile(target GoStorageObject, sourceFile string)
  downloadFile(source GoStorageObject, targetFile string)
  listFilesInBucket(source GoStorageObject) []string

  deleteFile(target GoStorageObject)
}
```

Fig. 8. *GoStorage* `Provider` interface.

```
type GoStorageObject struct {
    Bucket          string
    Key             string
    Region          string
    IsLocal         bool
    LocalFilePath   string
    ProviderType    ProviderType
}
```

Fig. 9. `GoStorageObject` definition.

encapsulate all necessary information for the storage location, including the bucket and file name, region, and provider (`AWS` or `GCP`). Fig. 9 presents the definition of the `GoStorageObject struct`. This object contains details about the object's location of the object, which can be either `local` or `remote`. If the `GoStorageObject` points to a local file, it also includes the local file path `LocalFilePath`. Given that GCP supports multi-region cloud storage deployments, developers can optionally utilize the region `key` for `GoStorageObject` objects pertaining to GCP. `ProviderType` is an enumeration consisting of `AWS` and `GCP` in the current implementation.

3.3.1. GoStorage usage

Gostorage initialization. At first, developers need to initialize *GoStorage*, as presented in Fig. 10. They can use the utility function `LoadCredentialsFromDefaultLocation()` of the `goStorage` object to load the credentials file and retrieve the parameters needed for creating the credentials types. Subsequently, developers should construct `CredentialsHolder` type and utilize it to instantiate the `goStorage` object.

Download a file from and upload a file to a storage with gostorage. *GoStorage* facilitates the downloading of files from and uploading files to storage. Developers can accomplish these tasks by providing the source and destination paths. For example, in Fig. 11, we demonstrate how *GoStorage* downloads a remote object from AWS S3 to the local file system and subsequently uploading the same file to the storage of a different cloud provider. Notably, neither AWS' nor GCP's SDKs are directly employed for interacting with their storages; instead, a unified interface, `CopyFromString()`, is utilized. Developers simply need to specify the file path of the local file system and the path of the remote location. It is worth noting that while the paths in Fig. 11 are hardcoded for clarity, they can be dynamically adjusted without necessitating rebuilding and redeploying.

Copy a file (e.g., a function deployment package) between storages of different providers. If developers need to copy a file from a source storage to a destination storage, regardless whether the source and destination storages belong to the same or to different providers, they can use the same `CopyFromString()` method, as presented in Fig. 12.

3.3.2. GoStorage storage URI parser

The examples provided in the previous Section 3.3.1, demonstrating (1) uploading a file to a storage, (2) downloading a file from a storage, and (3) copying files between storages, are intended for software practitioners to easily utilize the *GoStorage* module as a stand-alone library. However, in the context of *CODE*, the source and target storages are dynamically determined by the DSL parser when parsing the provided storage URIs from the input DSL for deployment. *GoStorage* identifies whether the URI corresponds to AWS or GCP by parsing the information from `parseUrlToGoStorageObject(...)`. Based on the determined storage provider, either `parseAWSUrl(...)` or `parseGoogleUrl(...)` is called. If the URI does not match with AWS or GCP, *GoStorage* examines whether the object is a file with a given file path from the local file system. If so, *GoStorage* sets the `IsLocal` flag and the `LocalFilePath` field. Otherwise, *GoStorage* raises an error and exits.

3.4. GoDeploy software architecture

CODE introduces *GoDeploy* to abstract the deployment of providers in Federated FaaS. The current version of *GoDeploy* supports deployment on two providers AWS and GCP.

3.4.1. GoDeploy domain-specific language

GoDeploy offers developers a YAML-based DSL to encode the necessary information for each function to be deployed in federated FaaS. *GoDeploy* supports the deployment of multiple functions at once. Each function can be deployed across multiple regions of AWS and GCP, all from a single deployment package, which can be stored either in the local file system where *CODE* runs, or on AWS S3 in any region, or in any region of GCP storage. The location of the deployment package does not affect where the function should be deployed. In particular, *GoDeploy* parses the location of the deployment package and, based

```
//Loading Credentials
awsCredentials, googleCredentials := gostorage.LoadCredentialsFromDefaultLocation()

credentialsHolder := gostorage.CredentialsHolder{
    AwsCredentials:    awsCredentials,
    GoogleCredentials: googleCredentials,
}

//Initializing GoStorage with credentials
goStorage := gostorage.GoStorage{Credentials: credentialsHolder}
```

Fig. 10. *GoStorage* initialization.

```
//Download from AWS
goStorage.CopyFromString("https://srcBucket.s3.../file", "/tmp/file")

//Upload to GCP
goStorage.CopyFromString("/tmp/file", "gs://destBucket/file")
```

Fig. 11. An example of how to download and upload a file with *GoStorage*.

```
// AWS -> AWS
goStorage.CopyFromString("https://srcBucket.s3.../file", "https://destBucket.s3.../

// AWS -> GCP
goStorage.CopyFromString("https://srcBucket.s3.../file", "gs://destBucket/file")

// GCP -> GCP
goStorage.CopyFromString("gs://srcBucket/file", "gs://destBucket/file")

// GCP -> AWS
goStorage.CopyFromString("gs://srcBucket/file", "https://destBucket.s3.../file")
```

Fig. 12. An example of how to copy a file between two different storages with *GoStorage*, regardless if they belong to one or two providers.

on the URI, selects the appropriate SDK of AWS or GCP to deploy the function.

Fig. 13 presents an example of *GoDeploy*'s DSL when the developer utilizes the recommended programming model of *CODE* to prepare a single deployment package for multiple providers. In this example, the deployment package of the function `function1` is stored on some region of AWS S3 and needs to be deployed in two AWS regions `us-east-1` (North Virginia) and `us-west-2` (Oregon) with the `AWSHandler` handler method. The same deployment package also needs to be deployed on GCP's region `us-east1` (South Carolina) with the `GoogleHandler` handler method and the Go runtime, as shown in Fig. 6. While the IaC frameworks Terraform and Serverless Framework place the provider at the top level of the hierarchy, *GoDeploy* places the function at the top of the hierarchy. The developer specifies in which providers and their regions to deploy a single deployment package only after defining the parameters of the function. In general, *GoDeploy*'s DSL introduces a three-level hierarchy starting from deployment package, then the providers, and finally regions for each provider where the function should be deployed. Note that the developer does not need to manually copy the deployment package of function `function1` from AWS S3 to GCS cloud storage in North Virginia because the deployment package will be copied by *CODE* in the background.

Level 1 (top): Function. The infrastructure coding begins with the `functions` keyword, which serves as the root for defining multiple function parameters. Each function definition consists of several keys. The location of the deployment package, to be deployed, is specified in the `archive` field. The string can point to either a local file path of the *zip* file or a URI of GCP storage or AWS S3. The `name` field, set to *function1*, designates the name of the deployed function across all locations. If the specified `name` already exists in any of the regions specified later in the DSL, the existing function will be updated with the new deployment package. The subsequent fields `memory`

```
functions:
  - archive: "https://bucket.s3.region.amazonaws.com/../.."
    name: "function1"
    memory: 128
    timeout: 60
    providers:
      - name: "AWS"
        handler: "example.AWSHandler"
        regions:
          - "us-east-1"
          - "us-west-2"
        runtime: "go1.x"
      - name: "Google"
        handler: "example.GoogleHandler"
        regions:
          - "us-central1"
        runtime: "go116"
  - archive: "https://storage.cloud.google.com/bucket/../.."
    name: "function2"
    ...
```

Fig. 13. Example of deployment script in *GoDeploy* DSL using the *CODE* programming model.

and `timeout` determine the memory allocation in megabytes and the timeout duration in seconds, respectively, for the function.

Level 2 (middle): Providers. The second level in the hierarchy is focused on the providers and begins with the key `providers`. After specifying the archive in level 1, developers can designate the provider on which the deployment package should be deployed. To do this, developers fill in the `name` of the provider (AWS or GCP), the `handler` method, and the `runtime`. Ideally, developers can deploy the same deployment package on multiple providers. In such case, developers do not need to reenter the same information from level 1. Instead, they continue filling in the values for the second provider under a new field called `name`. When filling in the handler, developers use the file name (e.g., `example`), followed by a separator character ".", and the handler method (e.g., `AWSHandler`) that is invoked when the function is called. Although all top providers support common runtime environments, their names may differ. For example, developers need to specify `go116` and `go1.x` for GCP and AWS Go runtime environments, respectively. Therefore, we place the value for `runtime` at level 2, instead of level 1.

Level 3 (bottom): Regions. After specifying each provider, developers need to specify its regions in the third level of the hierarchy. This step must be repeated for each provider. This level is the simplest, as developers use the `regions` parameter to specify a list of strings containing the regions for the parent provider. Developers must use the standardized code names of AWS and GCP regions, such as `us-east-1` for AWS North Virginia and `us-east4` for GCP North Virginia. Once all entries in the three layers have been coded for the function `function1`, developers repeat the same procedure for the next function `function2`, and so on.

DSL without the code programming model. In Section 3.1, we recommended that developers should use the *CODE* programming model and create a single deployment package of a function across all providers. This approach reduces the amount of code that needs to be written in *CODE*. However, some developers may prefer not to adopt the programming model proposed in Section 3.1. For example, they may prefer smaller deployment packages and exclude dependencies for providers not in use. In this alternative approach, developers need to create a total of p deployment packages for a single function, one for each of the p providers. However, the advantage of this approach is that the deployment packages per provider are smaller, as they do not include unnecessary dependencies. If developers choose not to use the recommended *CODE* programming model, they will need to write more LoC in the *CODE* DSL, as illustrated in Fig. 14. Additionally, maintenance becomes more complex, as developers must manage p deployment packages for each function. As shown in Fig. 14, apart from the common lines for the GCP provider (the last 5 lines), developers must re-enter the provider-specific LoC to specify the URI of the archive for the second provider — GCP, as well as memory, timeout, and the predefined key `providers`. The LoC for both programming models are evaluated in Section 4.2.1, revealing that both approaches result in the same complexity for the DSL when the number of regions where the function needs to be deployed is large.

3.4.2. GoDeploy DSL parser implementation

GoDeploy utilizes several internal structures to accurately parse the input DSL and convert it into equivalent objects. For this purpose, a *data transfer object* [41] (*DTO*) was used, which represents data in a form that is easily transferable. The structure of a `DeploymentDto` object corresponds to the definition for a function in the input DSL file, as presented in Fig. 15. The `‘mapstructure:‘‘<KEY>’’‘` identifiers are necessary for deserializing the deployment file and populating the correct fields in the `struct`.

Utilizing the *CODE*'s innovative three levels of abstraction, a single function can be deployed across multiple providers. Consequently, `DeploymentDto` includes a list of providers, represented by the `Provider` structure in *GoDeploy*, as illustrated in Fig. 16. Each instance of the `Provider` structure corresponds to a provider block of a function within the input DSL file.

Once *CODE* parses the DSL file and creates the objects of `DeploymentDto` and `Provider` structures, it generates objects of the `Deployment` structure, as shown in Fig. 17. With this approach, *GoDeploy* converts the list of `Provider` objects and the list of `region` strings for each `Provider` object into separate objects `deployment`, which can then be processed and deployed simultaneously, as described in the following section.

```
functions:
  - archive: "https://bucket.s3.region.amazonaws.com/../.."
    name: "function1"
    memory: 128
    timeout: 60
    providers:
      - name: "AWS"
        handler: "example.AWSHandler"
        regions:
          - "us-east-1"
          - "us-west-2"
        runtime: "go1.x"
  - archive: "https://storage.cloud.google.com/bucket/../.."
    name: "function1"
    memory: 128
    timeout: 60
    providers:
      - name: "Google"
        handler: "example.GoogleHandler"
        regions:
          - "us-central1"
        runtime: "go116"
```

Fig. 14. Example of deployment script in *GoDeploy* DSL without using the recommended *CODE* programming model. In this case, the developer creates a separate deployment package (archive) for each provider, which requires to repeat the lines from level 1 (`archive`, `name`, `memory`, and `providers`). The other lines (levels 2 and 3) are anyway needed.

```
type DeploymentDto struct {
    Archive     string      `mapstructure:"archive"`
    Name        string      `mapstructure:"name"`
    MemorySize  int32       `mapstructure:"memory"`
    Timeout     int32       `mapstructure:"timeout"`
    Providers   []Provider  `mapstructure:"providers"`
}
```

Fig. 15. `DeploymentDto` structure definition, which corresponds to the function in *CODE*'s DSL.

```
type Provider struct {
    Name     ProviderName  `mapstructure:"name"`
    Handler  string        `mapstructure:"handler"`
    Regions  []string      `mapstructure:"regions"`
    Runtime  string        `mapstructure:"runtime"`
}
```

Fig. 16. `Provider` structure definition, which corresponds to the provider in *CODE*'s DSL.

```
type Deployment struct {
    Archive          string
    Name             string
    MemorySize       int32
    Timeout          int32
    Runtime          string
    Provider         ProviderName
    HandlerFile      string
    HandlerFunction  string
    Region           string
    Bucket           string
    Key              string
}
```

Fig. 17. Internal `Deployment` structure definition, whose objects contain all necessary information that are needed to deploy a function.

3.4.3. Parallel deployment

Using the *CODE* DSL, users can specify that a single function be deployed across multiple regions of several providers. Given that some providers have more than 30 regions and users may need to deploy multiple functions in Federated FaaS, a single deployment script could specify hundreds of deployments. To minimize the overall deployment

```
if shared.ProviderAWS == deployment.Provider {
    go aws.Deploy(&waitGroup, deployment, credentials)
}

if shared.ProviderGoogle == deployment.Provider {
    go google.Deploy(&waitGroup, deployment, credentials)
}
```

Fig. 18. Parallelized functions deployment with goroutines.

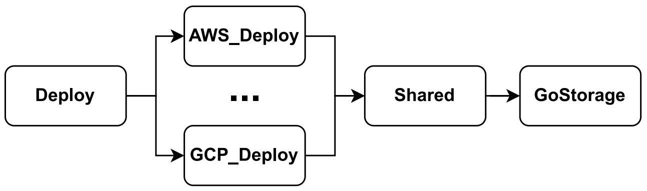

Fig. 19. *CODE* integrates *GoStorage* into *GoDeploy*.

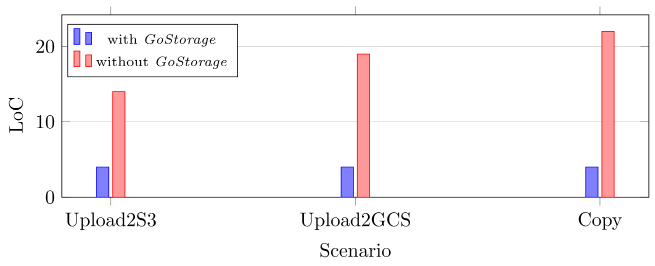

Fig. 20. Evaluation of *CODE* internal federated storage interface with and without *GoStorage* with LoC for three scenarios (1) to upload a deployment package to AWS S3, (2) upload a deployment package to GCP, and (3) copy a deployment package between AWS S3 and GCP storage.

time, the *GoDeploy* module utilizes *goroutines*, creating a lightweight thread for each deployment separately.

The current implementation of *GoDeploy* supports deployment on AWS and GCP. However, *CODE* can be easily extended by adding new constants for the `ProviderName` (e.g., `ProviderIBM`) and implementing `IBM_Deploy` to enable deployment across three providers. Depending on the selected cloud provider, *GoDeploy* calls either `go aws.Deploy(..)` or `go google.Deploy(..)` to initiate the function's deployment on the corresponding provider in a separate thread, as shown in Fig. 18. The `&waitgroup` variable tracks the state of the parallel threads and ensures that all deployments finish properly.

3.5. *CODE integrates GoStorage into GoDeploy*

Fig. 19 illustrates the internal structure of *CODE*. The general module `deploy` manages most of the setup, credential management, and DSL validation. The abstraction is then divided into provider-specific implementations. The current implementation includes the `AWS_Deploy` and `Google_Deploy` modules, which can be easily extended with similar modules for other providers. These modules leverage the internal shared logic of the `Shared` module, which contains utility functions, error handling, and constants. Finally, the `Shared` module utilizes the *GoStorage* library, which transparently copies deployment packages of the functions from either the local file system or the source storage to the target storage in the regions where the functions are to be deployed.

4. *CODE* evaluation

This section evaluates the benefits of *CODE* and its two modules *GoStorage* and *GoDeploy*. Given that LoC is a relevant metric that is often used by other researchers [4,42–44], we first assess the benefit of *GoStorage* as an internal module of *CODE* for federated storage interfaces in Section 4.1. This evaluation focuses on its ability to upload or copy deployment packages to AWS S3 and GCP storage. In Section 4.2, we then evaluate *GoDeploy*'s DSL in terms of deployment script length measured in LoC, and compare it with DSLs of state-of-the-art IaC frameworks such as Terraform and Serverless Framework, as well as the recent M2FaaS FaaSifier [14]. Additionally, we evaluate the correlation between the deployment package size and deployment time on AWS and GCP for several functions that use *GoStorage* in Section 4.3. We also repeat the assessment for the functions of our motivating workflow, `celebrityCollage`, written in Python, in Section 4.4.

4.1. *CODE benefit from GoStorage*

Evaluation methods. We evaluate the usage of *GoStorage* in *CODE* in terms of LoC that the developer needs to write with *GoStorage* and compare it with providers' SDK without *GoStorage*, which is the original version of *GoDeploy* [16]. We evaluate *GoStorage* with the three scenarios in which *CODE* uses *GoStorage* internally. Specifically, we measure the LoC needed to (1) upload the deployment package of a function on AWS S3, (2) upload the deployment package of a function on GCP storage, and (3) copy the deployment package of a function between AWS S3 and GCP storage.

Code reduction with gostorage. Fig. 20 presents the evaluation of how much LoC is reduced by *GoStorage* compared with the state-of-the-art approaches with the corresponding storage client SDKs in Go for AWS S3 and GCP storage. We observe that *GoStorage* significantly reduces LoC, especially when the storage client needs to use SDKs from both AWS and GCP to copy the deployment package. Regardless of the evaluated scenario, *GoStorage* requires the same number of LoC (four) to initiate the `gostorage` object and use the single method `goStorage.CopyFromString()`. With *GoStorage*, *CODE* reduced LoC by 71.4% compared to the AWS S3 SDK and up to 81.8% compared to using both storage SDKs in Go (for copying).

4.2. *GoDeploy DSL evaluation*

Methods. We evaluate the usage of *CODE* in terms of DSL's LoC with two experiments that scale the number of regions of an individual provider and the number of regions in federated FaaS. For this purpose, we derive equations for each evaluated framework for deployment. We compare *CODE* with the two state-of-the-art IaC frameworks Terraform and Serverless Framework, as well as the FaaSifier M2FaaS, which also offers a DSL for deploying a code block of a monolith as a function across multiple providers and their regions.

Fairness. Apart from automatic deployment of functions in federated FaaS, Terraform and Serverless Framework support deployment of different resources, such as buckets, message queues, security roles, API gateways, etc. Also, they support integration with GitHub actions or automatic creation of deployment packages (packaging). These features require more LoC per function than *CODE* and M2FaaS. Therefore, for fair comparison, we evaluate both IaC frameworks for LoC needed only to deploy functions.

4.2.1. CodeDSL

CodeDSL with the code programming model. *CODE*'s approach, prioritizing the serverless application over the cloud provider, initially reduces the LoC in the deployment script. For a single function, *CODE* needs 5 LoC to specify the location of the deployment package (`archive`), the name (`name`), allocated memory (`memory`), timeout (`timeout`) and finally, a fixed line for the predefined key `providers`. Further on, if the developer uses the *CODE* programming model and creates a single deployment package for multiple providers, then the same deployment package is deployed on multiple providers. In this case, the *CODE* DSL requires only 4 LoC per provider to specify: its name, the handler that is called when the function is invoked, runtime, and the predefined key `regions`. Finally, for each region of the provider, *CODE* needs a single LoC. Based on the above-mentioned complexity analysis, in Eq. (1) we model the total number of LoC needed to be written for deployment of a function in federated FaaS. The total number of LoC is affected by the number of providers p and the average number of regions r per provider on which the function needs to be deployed.

$$LoC^{w}_{CODE} = [5 + p \cdot (4 + r)] \quad \sim \quad p \cdot r \tag{1}$$

CodeDSL without the code programming model. If the developers choose to create a separate deployment package per provider, they need to write five more LoC per each provider, as presented in Fig. 14. We generalize this approach in Eq. (2). Although this approach requires $5 \cdot (p - 1)$ LoC more than packaging with *CODE* programming model, the complexity remains the same $p \cdot r$ for large number of regions per provider.

$$LoC^{w/o}_{CODE} = p \cdot (9 + r) \quad \sim \quad p \cdot r \tag{2}$$

4.2.2. Terraform DSL

Terraform's DSL uses a declarative way to describe functions that should be deployed, usually in multiple files, one for each provider. It uses a proprietary language, with three basic elements: blocks, arguments, and expressions. Blocks usually represent the configuration of an object (e.g. a resource), arguments appear in blocks to assign a value to a name, while expressions represent values. Terraform supports deployment of functions to all top providers. The total number of LoC of Terraform is given in Eq. (3). For a single function, Terraform requires developers to write 2 LoC to define the region. Additionally, they need 2 LoC to specify the region in the function, and yet another 7 LoC to define the needed parameters of the function. The entire procedure needs to be repeated for each provider.[10]

$$LoC_{Terraform} = 11 \cdot p \cdot r \tag{3}$$

4.2.3. Serverless framework's DSL

Serverless Framework uses an imperative two level hierarchy regions and functions. Developers need to compose scripts for each provider separately. They also need to code a separate file per region of that provider. Within the script, developers need to specify the service, provider, provider name, region, runtime, memory, duration, deployment package, function, and function name. The total number of LoC required by the Serverless Framework is given in Eq. (4). For a single function to be deployed on p providers and r regions per provider, Serverless Framework requires developers to write 2 LoC per provider in the compose file. Additionally, in each deployment script per region, developers need to write: 2 LoC for the provider, 1 LoC to specify the region, and 6 LoC for the function parameters.

$$LoC_{Serverless} = 9 \cdot p \cdot r + 2 \cdot p \quad \sim \quad 9 \cdot p \cdot r \tag{4}$$

4.2.4. M2FaaS DSL

M2FaaS uses a JSON-based DSL to describe where and how to deploy annotated code-block as a function on two supported providers AWS and IBM. M2FaaS does not require all deployment parameters because some of them are predefined within the FaaSifier. Among others, the location of the deployment package is not required because it is created locally, while the provider default handler method is used for each provider.

Unlike the other three frameworks, M2FaaS does not use any hierarchy to group the code, which means that developers need to specify the same number of LoC for each deployment of each function, regardless of the target provider and region. Due to its flat-based DSL, M2FaaS does not require any LoC per provider, but it needs 6 LoC for each function deployment in one region, which includes name of the function, provider, region, memory, runtime, and timeout. Based on this, Eq. (5) models the complexity of M2FaaS' DSL in terms of LoC.

$$LoC_{M2FaaS} = 6 \cdot p \cdot r \tag{5}$$

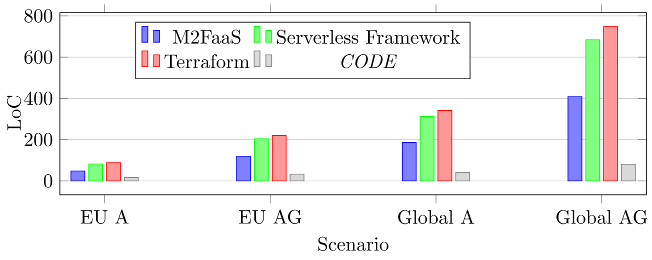

Fig. 21. LoC to be coded in the DSLs of M2FaaS, Serverless Framework, TerraForm, and *CODE* to deploy a function in four different real life scenarios (i) all 8 AWS regions in Europe (EU A), (ii) all 8 AWS and 12 GCP regions in Europe (EU AG), (iii) all 31 AWS regions globally (Global A), and (iv) all 31 AWS and 37 GCP regions globally (Global AG).

4.2.5. Real-case evaluation

We used Eqs. (1), (3), (4), and (5) to evaluate the LoC required for four real case scenarios. These scenarios involve deploying a function across: (i) all 8 AWS regions in Europe (EU A), (ii) all 8 AWS and 12 GCP regions in Europe (EU AG), (iii) all 31 AWS regions globally (Global A), and (iv) all 31 AWS and 37 GCP regions globally (Global AG). For the scenarios EU AG and Global AG, we used the average number of 10 and 34 regions per provider, respectively. Fig. 21 presents the results of this evaluation. We observe that, from all evaluated frameworks, *CODE* requires the lowest number of LoC to deploy a function for each scenario. Terraform requires from 5.18× to 9.23×, Serverless Framework from 5.18× to 8.44×, while M2FaaS requires from 2.82× to 5.04× more LoC than *CODE*, respectively. Note that these values are smaller than the theoretical ones 11×, 9×, and 6×, respectively. The reason is that *CODE* requires five LoC for each function and yet another four LoC for each provider. Because of this fact, all DSLs require similar LoC if a function needs to be deployed in a few regions. For instance, M2FaaS' DSL needs 6 LoC only to deploy a function in a single region, *CODE* and Terraform 9, while Serverless Framework 11 LoC.

4.3. Evaluation of *GoStorage* inside the functions

Benchmark functions. Further on, we developed four implementations of a function `ListBucket`. The first two implementations for AWS Lambda in Go list files in AWS S3 and GCP storage, respectively. The other two implementations were for GCP, which also list files in AWS S3 and GCP storage, respectively. We will refer to these functions as A2 A, A2G, G2 A, and G2G, respectively. Finally, we used the *CODE* programming model to create a single function that uses *GoStorage* and lists files from both storages. Additionally, we deployed the same deployment package in AWS and GCP, to which we refer as *CODE*.

Loc. We first present in Fig. 22 LoC of the four implementations without *GoStorage* and the two implementation with *GoStorage*. We observe that using *GoStorage*, developers need to code 6 or 7 LoC, compared to 14 to 17 LoC for the function implementations without *GoStorage*, or reduction of 57.38 % on average. In reality, this percentage is even higher since developers need to code 2 handler methods only (one for AWS and one for GCP). This leads in total of 13 LoC, or a reduction of 78.7 %.

[10] https://github.com/terraform-aws-modules/terraform-aws-lambda/blob/v5.0.0/examples/multiple-regions/main.tf

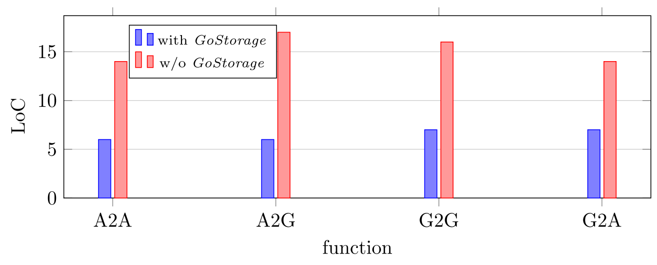

Fig. 22. LoC for various implementations of the function `ListBucket` in Go with and without *GoStorage*.

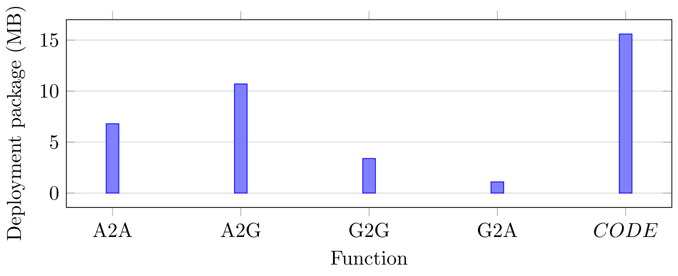

Fig. 23. Deployment package size for the four implementations of the function `ListBucket` without *GoStorage* and the single deployment package with *GoStorage*.

Deployment package size. Fig. 23 shows the size of deployment package for the same implementations of the function `listBucket`. As expected, the deployment packages that use provider's SDK of a single cloud storage have a smaller deployment package than the *CODE* function, since it uses SDKs for both cloud storages of AWS and GCP using *GoStorage*. We also observe that the size of the deployment packages is higher for AWS Lambda functions (6.8 MB and 10.7 MB) compared to respective deployment packages for GCP (1.1 MB and 3.39 MB). The reason for the 3.16× to 6.18× larger deployment packages is that AWS Lambda requires all dependencies to be stored inside the deployment package (the zip file). In contrast, for GCP, the developers need to list all dependencies in the `requirements.txt` file only to be included in the function during the deployment. Another interesting observation is that the deployment package of functions A2G and G2G that use the SDK for GCP storage is larger by 3.9 MB and 2.29 MB than the respective functions A2 A and G2 A that use the AWS S3 SDK, respectively. Finally, although the deployment package of the function that uses *CODE* has a larger size of 18.6 MB, its deployment package is smaller by 29.06% than all four functions together (21.99 MB). This justifies running the function on both clouds and listing files from the cloud storages of both providers, AWS and GCP.

Deployment time. We further evaluate deployment time of the packages to both providers to examine the correlation with the size of the deployment package. Fig. 24 presents the results for deployment times of the packages to regions in North Virginia of both providers. Contrary to expectations, we observed that the deployment to GCP takes significantly longer than to AWS, despite the deployment packages being larger for AWS. The implementations `G2G`, `G2A` require, on average, 62.5 s longer deployment time than their Lambda counterparts. In total, a user would need 253 s to deploy all four functions on both providers, while they would only need 116 s to deploy the single function on both providers, resulting in an overall deployment time reduction of 46 %.

4.4. Evaluation of the deployment of Python functions

Benchmark python functions. To generalize our findings for deploying functions written in different programming languages, we also evaluated the functions of the workflow presented in Fig. 4. All functions are written in Python for both for GCP and AWS implementations. The only exception is the function `detectCelebrities`, which was deployed only on AWS because GCP restricts the use of celebrity detection to industrial partners, preventing us from using the GCP service.

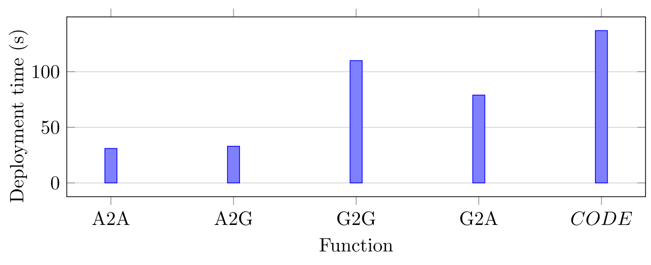

Fig. 24. Deployment time for the four implementations of the function `ListBucket` without *GoStorage* and the single deployment package with *GoStorage*.

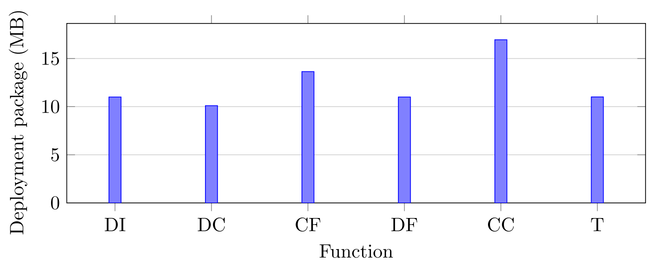

Fig. 25. Size of the deployment package for functions DI (`Distribute Images`), DC (`Detect Celebrities`), CF (`Crop Faces`), DF (`Distribute Faces`), CC (`Create Collage`), *T* (`Translate`) of the `celebrityCollage` serverless workflow implementations in Python for AWS.

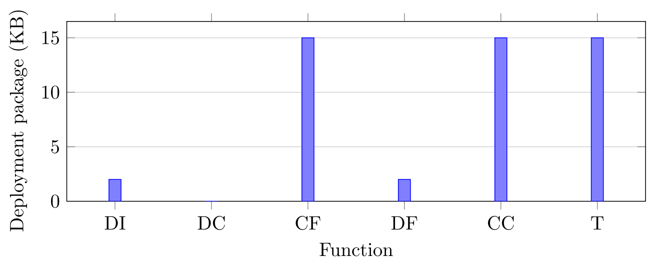

Fig. 26. Size of the deployment package for functions DI (`Distribute Images`), DC (`Detect Celebrities`), CF (`Crop Faces`), DF (`Distribute Faces`), CC (`Create Collage`), *T* (`Translate`) of the `celebrityCollage` serverless workflow implementations in Python, for GCP.

Deployment package size. Fig. 25 presents the size of deployment package for each function of the workflow for AWS. We observe that deployment packages for all functions have a similar size, ranging from 10.1 MB up to 16.96 MB on AWS. However, similar to the functions in Go presented in the previous Section 4.3, Fig. 26 shows that deployment packages for GCP functions are significantly smaller, ranging from 2 kB to 15 kB. This size difference is because AWS deployment packages include all dependencies, while GCP functions only require references in the `requirement.txt` file only, which are included during deployment.

Deployment time. The time needed to deploy the Python functions does not follow the pattern of deployment package size, either. Fig. 27 presents deployment times for all functions across two regions. We observe that it takes 133.7 s and 206.2 s to deploy all functions of the serverless workflow in AWS's North Virginia and Oregon regions, respectively. In contrast, significantly longer times of 649.2 s and 735.7 s

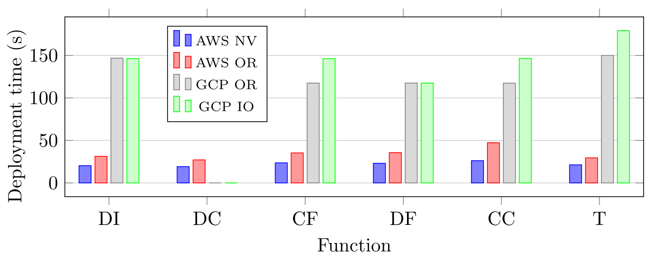

Fig. 27. Deployment time for functions presented in Figs. 25 and 26 in AWS's North Virginia and Oregon and GCP's Oregon and Iowa.

are required for GCP's Oregon and Iowa regions, respectively, which is 3.6× to 4.9× longer. Note that the function `detectCelebrities` is not included in the calculation for GCP. Additionally, we observe a consistently longer deployment time for AWS Oregon compared to the "closer" AWS North Virginia, due to network proximity [24]. For GCP, this is not observed for the `distributeImages` and `distributeFaces` functions since their deployment package are only a few kilobytes, making the impact of network proximity negligible.

5. Discussion

This section presents the related work and explains how *CODE* goes beyond the state-of-the-art approaches for automatic deployment of functions in federated FaaS and federated storage infrastructures. It also highlights the novelty of this paper compared to our previous work [16]. We also discuss *CODE* limitations and threats to validity.

5.1. Related work

We separate the related work in four areas (i) DSLs of IaC frameworks, (ii) automatic deployment of functions with FaaSifiers in Federated FaaS, (iii) unified storage access, and (iv) the novelty compared to the original paper [16].

5.1.1. Automatic deployment with IaC tools

Provider specific iac. AWS introduced their YAML-based DSL CloudFormation. Unfortunately, while the DSL is open source, CloudFormation works for AWS infrastructure only. Therefore, developers need to learn SDKs of all providers to deploy functions across multiple providers in federated FaaS.

Iac frameworks. Most popular open source IaC frameworks, such as Terraform and Serverless Framework, offer their own DSLs to code and automatize deployment of functions. Unfortunately, DSLs of both frameworks are mainly focused on the provider, rather than the function (applications). Such a hierarchy, as our evaluation has shown, requires writing redundant LoC when the same deployment package needs to be deployed across multiple regions of the same or different providers. SEAPORT [45] automatically evaluates the portability of serverless orchestration tools with respect to a chosen target provider or platform. QuickFaaS [46] improves portability of FaaS code by defining a uniform programming model across providers, while also taking care of the deployment steps to different cloud providers.

FaaSifiers. The recent FaaSifier M2FaaS [14] introduces a DSL without hierarchy. M2FaaS DSL requires fewer parameters to be coded for each deployment, since some parameters are predefined. For instance, the archive location is not needed as the archive is created automatically during FaaSification. Still, it needs six parameters (function name, provider, region, timeout, memory, and runtime), which are needed to be filled for each function deployment.

Code novelty. While all above-mentioned approaches automate the deployment of functions in federated FaaS, they require many LoC to be coded multiple times for each region of each provider separately. Developers need to repeatedly specify several parameters, such as the name and deployment package location, even when they are the same. Unlike these approaches, *CODE* puts the function at the top of the hierarchy, allowing developers to simply add provider-specific information and regions at the lower levels. This hierarchy reduces LoC compared to state-of-the-art IaC frameworks.

5.1.2. Automatic packaging and deployment with FaaSification

In order to benefit from FaaS scalability and elasticity, monolithic applications are *faasified*, that is, converted into hybrid applications by exporting parts of its code as equivalent functions [3,8,12,14]. Still, the interface to the user is retained and running the offloaded code is transparent [47,48].

Serverless Application Analytic Framework (SAAF) [49] automatically creates a deployment package by wrapping the local method with the specific handler and deploys it. While SAAF supports deployment on multiple providers, it supports a single region per provider, that is, the default region that is configured in the local credentials. The Node2FaaS FaaSifier [13] extracts methods that contain loops as functions and can deploy them on three providers AWS, GCP, and Azure, but also supports the default regions only. Another weakness of SAAF and Node2FaaS is that they support isolated functions only.

Ristov et al. [12] enhanced FaaSification by automating the packaging of code and its dependencies, and deploying the functions on AWS. The same authors further automated data dependencies and introduced fault tolerance with multi-provider support [14]. IaC frameworks also support packaging of functions during deployment. While IaC frameworks support deployment of functions in different programming languages, FaaSifiers are not language agnostic and support only a single programming language.

5.1.3. Unified storage access

Storage federation systems. Several works have presented systems that support unified global access to storages of different cloud providers, which are complementary to *CODE* as they can speed up data access times. Onedata [50] improves block-based data transfer in federated storage by splitting files and distributing their chunks across different storages. Lithops [11] shares files between multiple functions of a parallel loop. Although both systems allow seamless global data access by keeping multiple copies with reduced access time, they do not provide mechanisms to select the appropriate storage region. The *GoDeploy* module of our *CODE* supports dynamic URIs from AWS S3 or GCP storage, allowing developers to specify any bucket from both providers or a local file path.

Libraries for unified storage access. Several libraries support unified data access to multiple storages. pkgcloud [51] is a Node.js library that abstracts the storage services of different providers behind a common interface. Similarly, Apache Libcloud [52] hides the storages behind common interfaces for Python applications, while Apache jClouds [53] offers similar support for Java applications. Unfortunately, although these libraries provide a common interface for accessing federated storage, developers need to hardcode the provider in the code and the corresponding driver to their storage accordingly. In contrast, *CODE*'s *GoStorage* module offers developers to dynamically select the storage by simply replacing a URI from one provider with another one from other provider. This feature is already utilized by *CODE* for specifying archive locations of deployment packages.

5.1.4. Extensions compared to our previous paper godeploy

This paper significantly extends our previous paper [16]. In particular, this paper introduces:

- A novel library *GoStorage*, which dynamically selects the target storage driver based on the given URI, regardless if it is from AWS or GCP;
- The *GoStorage* library reduces LoC by up to 81.8 % compared to the individual provider SDKs;
- Extended evaluation with Serverless Framework, apart from Terraform and the M2FaaS FaaSifier;
- Generalized model for LoC of DSLs (9.23× reduction);
- Evaluation of deployment package size and deployment time of functions with *CODE*.

5.2. CODE limitations

Support for two providers. Both modules of *CODE* support developer APIs for unified storage access and deployment of functions on two providers AWS and GCP. However, *CODE*'s architecture is designed to be easily extendable to support the deployment of workflow functions on other providers and abstract their storages. Developers simply need to implement the provider specific implementations of the general module `<Provider>_Deploy` presented in Fig. 19 and the `Provider` interface presented in Fig. 8, respectively.

Attaching layers to functions. *CODE* facilitates the deployment of functions with dependencies included in the deployment package or referenced in the `references.txt` file for functions on GCP. However, certain functions may utilize layers, a feature not yet supported by *CODE*, but available in the state-of-the-art IaC frameworks Terraform and Serverless Frameworks. Nevertheless, *CODE* prioritizes enhancing the DSL rather than duplicating features already present in existing IaC frameworks.

5.3. Threats to validity

Loc reduction vs. other programming languages. *CODE* is implemented in Go programming language and utilizes the corresponding Go SDKs of AWS and GCP for copying deployment packages and deploying them in federated FaaS. Our evaluation revealed that *CODE* significantly reduces LoC compared to these SDKs. However, SDKs in other programming languages require fewer LoC than Go's. Nonetheless, *CODE* has already minimized the necessary LoC, potentially narrowing the gap. For instance, AWS' Boto3 library in Python typically demands 10 LoC for file transfer to AWS S3,[11] with an additional 7 LoC for GCP storage transfer with Boto3.[12] This suggests that similar LoC are required for Python SDKs as for Go's.

Loc reduction vs. IaC frameworks. It is essential to note that reducing LoC could conceal some provider-specific functionalities from users. Our evaluation was based on the current version of IaC framework DSLs. Still, to mitigate bias, we only considered the minimum configuration necessary for function deployment.

6. Conclusion and future work

CODE is an IaC framework designed to automate the deployment of functions in federated FaaS environments. Adopting the principle of "code once, deploy everywhere", *CODE* enables developers to create a single deployment package for all supported providers and write minimal deployment scripts to deploy functions across federated FaaS. The deployment package can reside in the local file system or a single storage region, with *CODE* transparently copying it to all target regions across different providers. Currently, *CODE* supports copying deployment packages to AWS S3 and GCP storage, as well as deploying the function.

Novelty. The main novelty of this work is the publicly available *GoStorage* library, integrated within *CODE* but also usable in any Go application, including serverless functions. It enables unified access to AWS S3 and GCP Storage and allows dynamic switching of storage providers during runtime.

Contribution. *CODE*'s DSL reduces the lines of code needed for deployment by up to 9.23× for all AWS and GCP regions due to its innovative three-layer hierarchy, which prioritizes the function above providers and regions. Within the function, the *GoStorage* library offers two main benefits: (i) it reduces lines of code by up to 78.7 % compared to using both provider SDKs, and (ii) unlike state-of-the-art storage access libraries that hard code the storage driver, *GoDeploy* allows for dynamic selection of storage providers during runtime.

Insights. Although *CODE* and other IaC frameworks automate deployment, the deployment time can be considerable, particularly on GCP where dependencies are integrated into the function during deployment. This approach poses a challenge for federated FaaS. For example, AWS Lambda requires up to half a minute to deploy a function across two regions (North Virginia and Oregon), whereas GCP can take up to three minutes for nearby regions — resulting in a deployment time that is six times longer. Notably, GCP reports longer deployment time despite having significantly smaller deployment packages (in kilobytes) compared to AWS functions, which are in the range of megabytes.

Future work. We plan to extend *CODE* in five directions:

1. *Model deployment time:* One insight from this paper is the paradoxical deployment time, which does not correlate with the deployment package size but rather with the network proximity of the region where the function is deployed. We will investigate function deployment in federated FaaS in more detail and develop a mathematical model to estimate deployment time based on various parameters, including provider, region, deployment package size, programming language, layers, and other significant factors;
2. *Benchmarking deployed functions:* We plan to extend *CODE*, including its DSL, to specify benchmarking of various non-functional requirements of deployed functions, such as performance and cost. This will involve using different data inputs to attach different storage with *GoStorage*, concurrency, repetitions, memory, etc. Finally, it will maintain Pareto-optimal deployments across federated FaaS;
3. *Unified interface for other managed cloud services:* Our evaluation showed significant reduction of lines of code when using the *GoStorage* library. We will extend its architecture to support other managed cloud services, such as object recognition, speech2text, text2speech, OCR, etc;
4. We plan to develop *libraries for other programming languages* such as Python, Java, and Node.js, as well as for other providers;
5. We will develop *middleware for the automatic deployment of AFCL workflows* in federated FaaS using *CODE*, based on the decision of a scheduler.

CRediT authorship contribution statement

Sashko Ristov: Writing – original draft, Visualization, Supervision, Methodology, Investigation, Conceptualization. **Simon Brandacher:** Software, Methodology, Investigation. **Mika Hautz:** Validation, Software, Investigation. **Michael Felderer:** Writing – review & editing. **Ruth Breu:** Writing – review & editing.

[11] https://boto3.amazonaws.com/v1/documentation/api/latest/guide/s3-uploading-files.html

[12] https://russell.ballestrini.net/copying-files-between-cloud-object-stores-like-s3-gcp-and-spaces-using-boto3/

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the public repository of both libraries within the paper.

Acknowledgment

This work was funded by Land Tirol, German, funding contract F.35499.

References

[1] M.T. Heath, Scientific Computing: An Introductory Survey, Revised Second Edition, SIAM, Philadelphia, 2018.

[2] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, R. Prodan, A serverless real-time data analytics platform for edge computing, IEEE Internet Comput. 21 (4) (2017) 64–71, http://dx.doi.org/10.1109/MIC.2017.2911430.

[3] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, Occupy the cloud: Distributed computing for the 99%, in: Symposium on Cloud Computing, 2017, pp. 445–451, http://dx.doi.org/10.1145/3127479.3128601.

[4] S. Ristov, S. Pedratscher, T. Fahringer, AFCL: An Abstract Function Choreography Language for serverless workflow specification, Future Gener. Comput. Syst. 114 (2021) 368–382.

[5] A. Arjona, P.G. López, J. Sampé, A. Slominski, L. Villard, Triggerflow: Trigger-based orchestration of serverless workflows, Future Gener. Comput. Syst. 124 (2021) 215–229, http://dx.doi.org/10.1016/j.future.2021.06.004.

[6] M. Malawski, A. Gajek, A. Zima, B. Balis, K. Figiela, Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google cloud functions, Future Gener. Comput. Syst. 110 (2020) 502–514, http://dx.doi.org/10.1016/j.future.2017.10.029.

[7] S. Ristov, S. Pedratscher, T. Fahringer, xAFCL: Run scalable function choreographies across multiple FaaS systems, IEEE Trans. Serv. Comput. 16 (1) (2023) 711–723, http://dx.doi.org/10.1109/TSC.2021.3128137.

[8] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, K. Chard, FuncX: A federated function serving fabric for science, in: International Symposium on High-Performance Parallel and Distributed Computing, HPDC '20, ACM, Stockholm, Sweden, 2020, pp. 65–76, http://dx.doi.org/10.1145/3369583.3392683.

[9] S. Ristov, D. Kimovski, T. Fahringer, FaaScinating resilience for serverless function choreographies in federated clouds, IEEE Trans. Netw. Serv. Manag. 19 (3) (2022) 2440–2452, http://dx.doi.org/10.1109/TNSM.2022.3162036.

[10] A. John, K. Ausmees, K. Muenzen, C. Kuhn, A. Tan, SWEEP: Accelerating scientific research through scalable serverless workflows, in: IEEE/ACM International Conference UCC Companion, ACM, Auckland, New Zealand, 2019, pp. 43–50, http://dx.doi.org/10.1145/3368235.3368839.

[11] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, P. Garcia-Lopez, Outsourcing data processing jobs with lithops, IEEE Trans. Cloud Comput. (2021) 1, http://dx.doi.org/10.1109/TCC.2021.3129000.

[12] S. Ristov, S. Pedratscher, J. Wallnöfer, T. Fahringer, DAF: Dependency-aware FaaSifier for Node.js monolithic applications, IEEE Softw. 38 (1) (2021) 48–53, http://dx.doi.org/10.1109/MS.2020.3018334.

[13] L. Carvalho, A.P.F. de Araújo., Remote procedure call approach using the Node2FaaS framework with terraform for Function as a Service, in: International Conference on Cloud Computing and Services Science - CLOSER, SciTePress, Online, 2020, pp. 312–319, http://dx.doi.org/10.5220/0009381503120319.

[14] S. Pedratscher, S. Ristov, T. Fahringer, M2FaaS: Transparent and fault tolerant FaaSification of Node.js monolith code blocks, Future Gener. Comput. Syst. 135 (2022) 57–71, http://dx.doi.org/10.1016/j.future.2022.04.021.

[15] S. Eismann, J. Scheuner, E.v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C.L. Abad, A. Iosup, The state of serverless applications: Collection, characterization, and community consensus, IEEE Trans. Softw. Eng. 48 (10) (2022) 4152–4166, http://dx.doi.org/10.1109/TSE.2021.3113940.

[16] S. Ristov, S. Brandacher, M. Felderer, R. Breu, GoDeploy: Portable deployment of serverless functions in federated FaaS, in: 2022 IEEE Cloud Summit, 2022, pp. 38–43, http://dx.doi.org/10.1109/CloudSummit54781.2022.00012.

[17] G.B. Berriman, E. Deelman, J.C. Good, J.C. Jacob, D.S. Katz, C. Kesselman, A.C. Laity, T.A. Prince, G. Singh, M.-H. Su, Montage: A grid-enabled engine for delivering custom science-grade mosaics on demand, in: Optimizing Scientific Return for Astronomy Through Information Technologies, vol. 5493, SPIE, 2004, pp. 221–232.

[18] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Gener. Comput. Syst. 29 (3) (2013) 682–692, http://dx.doi.org/10.1016/j.future.2012.08.015.

[19] M. Pawlik, P. Banach, M. Malawski, Adaptation of workflow application scheduling algorithm to serverless infrastructure, in: Euro-Par 2019: Parallel Processing Workshops, GÖTtingen, Germany, August 26–30, 2019, Springer, 2020, pp. 345–356.

[20] A. Al-Haboobi, G. Kecskemeti, Improving existing WMS for reduced makespan of workflows with lambda, in: Euro-Par 2020: Parallel Processing Workshops: Euro-Par 2020 International Workshops, Warsaw, Poland, August 24–25, 2020, Springer, 2021, pp. 261–272.

[21] M. Hautz, S. Ristov, M. Felderer, Characterizing AFCL serverless scientific workflows in federated faas, in: Proceedings of the 9th International Workshop on Serverless Computing, WoSC '23, Association for Computing Machinery, Bologna, Italy, 2023, pp. 24–29, http://dx.doi.org/10.1145/3631295.3631397.

[22] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, Y. Cheng, INFINICACHE: Exploiting ephemeral serverless functions to build a cost-effective memory cache, in: Conference on File and Storage Technologies, FAST '20, USENIX, Santa Clara, CA, USA, 2020, pp. 267–282.

[23] J. Sampé, G. Vernik, M. Sánchez-Artigas, P. García-López, Serverless data analytics in the IBM cloud, Middleware '18, ACM, Rennes, France, 2018, pp. 1–8.

[24] S. Ristov, M. Hautz, C. Hollaus, R. Prodan, SimLess: Simulate serverless workflows and their twins and siblings in federated FaaS, in: ACM Symposium on Cloud Computing, SoCC '22, ACM, San Francisco, CA, USA, 2022, pp. 323–339, http://dx.doi.org/10.1145/3542929.3563478.

[25] S. Ristov, C. Hollaus, M. Hautz, Colder than the warm start and warmer than the cold start! experience the spawn start in faas providers, in: Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems (ApPLIED '22), ACM, Salerno, Italy, 2022, pp. 35–39, http://dx.doi.org/10.1145/3524053.3542751.

[26] C.P. Smith, A. Jindal, M. Chadha, M. Gerndt, S. Benedict, FaDO: Faas functions and data orchestrator for multiple serverless edge-cloud clusters, in: 2022 IEEE 6th International Conference on Fog and Edge Computing, ICFEC, 2022, pp. 17–25, http://dx.doi.org/10.1109/ICFEC54809.2022.00010.

[27] B. Sethi, S.K. Addya, J. Bhutada, S.K. Ghosh, Shipping code towards data in an inter-region serverless environment to leverage latency, J. Supercomput. 79 (10) (2023) 11585–11610, http://dx.doi.org/10.1007/s11227-023-05104-7.

[28] J. Diaz-Montes, M. Diaz-Granados, M. Zou, S. Tao, M. Parashar, Supporting data-intensive workflows in software-defined federated multi-clouds, IEEE Trans. Cloud Comput. 6 (1) (2018) 250–263, http://dx.doi.org/10.1109/TCC.2015.2481410.

[29] J.J. Durillo, R. Prodan, J.G. Barbosa, Pareto tradeoff scheduling of workflows on federated commercial clouds, Simul. Model. Pract. Theory 58 (2015) 95–111, http://dx.doi.org/10.1016/j.simpat.2015.07.001, Special Issue on techniques and applications for sustainable ultrascale computing systems.

[30] I. Stoica, S. Shenker, From cloud computing to sky computing, in: Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '21, Association for Computing Machinery, Ann Arbor, Michigan, 2021, pp. 26–32, http://dx.doi.org/10.1145/3458336.3465301.

[31] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F.S. Luan, G. Mittal, S. Shenker, I. Stoica, SkyPilot: An intercloud broker for sky computing, in: Symposium on Networked Systems Design and Implementation, NSDI 23, USENIX, Boston, MA, 2023, pp. 437–455.

[32] S. Ristov, P. Gritsch, FaaSt: Optimize makespan of serverless workflows in federated commercial FaaS, in: 2022 IEEE International Conference on Cluster Computing, CLUSTER '22, IEEE, Heidelberg, Germany, 2022, pp. 182–194, http://dx.doi.org/10.1109/CLUSTER51413.2022.00032.

[33] A. Aske, X. Zhao, Supporting multi-provider serverless computing on the edge, in: International Conference on Parallel Processing Companion, ICPP '18, ACM, OR, USA, 2018, pp. 1–6, http://dx.doi.org/10.1145/3229710.3229742.

[34] M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Dustdar, O. Scekic, T. Rausch, S. Nastic, S. Ristov, T. Fahringer, A deviceless edge computing approach for streaming IoT applications, IEEE Internet Comput. 23 (1) (2019) 37–45, http://dx.doi.org/10.1109/MIC.2019.2892219.

[35] T. He, H. Khamfroush, S. Wang, T.L. Porta, S. Stein, It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources, in: 2018 IEEE 38th International Conference on Distributed Computing Systems, ICDCS, IEEE Computer Society, Los Alamitos, CA, USA, 2018, pp. 365–375, http://dx.doi.org/10.1109/ICDCS.2018.00044.

[36] B. Wang, A. Ali-Eldin, P. Shenoy, LaSS: Running latency sensitive serverless computations at the edge, in: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '21, ACM, Virtual Event, Sweden, 2021, pp. 239–251, http://dx.doi.org/10.1145/3431379.3460646.

[37] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, Q. Li, LAVEA: Latency-aware video analytics on edge computing platform, in: ACM/IEEE Symposium on Edge Computing, SEC '17, ACM, San Jose, California, 2017, pp. 1–13, http://dx.doi.org/10.1145/3132211.3134459.

[38] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, ACM, Providence, RI, USA, 2019, pp. 3–18, http://dx.doi.org/10.1145/3297858.3304013.
[39] C. Richardson, microservices patterns: with examples in java, Simon and Schuster, 2018.
[40] V. Yussupov, J. Soldani, U. Breitenbücher, F. Leymann, Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA, Softw. - Pract. Exp. 52 (6) (2022) 1454–1495.
[41] okta, Data Transfer Object DTO Definition and Usage, 2023, https://www.okta.com/identity-101/dto. (Accessed: 03 June 2024).
[42] H. Foidl, M. Felderer, Integrating software quality models into risk-based testing, Softw. Qual. J. 26 (2) (2018) 809–847, http://dx.doi.org/10.1007/s11219-016-9345-3.
[43] R. Hartauer, J. Manner, G. Wirtz, Cloud function lifecycle considerations for portability in function as a service, in: CLOSER, 2022, pp. 133–140.
[44] X. Lin, M. Simon, N. Niu, Scientific software testing goes serverless: Creating and invoking metamorphic functions, IEEE Softw. 38 (1) (2021) 61–67, http://dx.doi.org/10.1109/MS.2020.3029468.
[45] V. Yussupov, U. Breitenbücher, A. Kaplan, F. Leymann, SEAPORT: Assessing the Portability of Serverless Applications, in: International Conference on Cloud Computing and Services Science, CLOSER 2020, SciTePress, 2020, pp. 456–467, http://dx.doi.org/10.5220/0009574104560467.
[46] P. Rodrigues, F. Freitas, J. Simão, QuickFaaS: Providing portability and interoperability between FaaS platforms, Future Internet 14 (12) (2022) http://dx.doi.org/10.3390/fi14120360.
[47] L. Baresi, M. Garriga, A. De Renzis, Microservices identification through interface analysis, in: F. De Paoli, S. Schulte, E. Broch Johnsen (Eds.), Service-Oriented and Cloud Computing, Springer International Publishing, Cham, 2017, pp. 19–33, http://dx.doi.org/10.1007/978-3-319-67262-5_2.
[48] L. Baresi, D.F. Mendonça, M. Garriga, S. Guinea, G. Quattrocchi, A unified model for the mobile-edge-cloud continuum, ACM Trans. Internet Technol. 19 (2) (2019) http://dx.doi.org/10.1145/3226644.
[49] R. Cordingly, H. Yu, V. Hoang, Z. Sadeghi, D. Foster, D. Perez, R. Hatchett, W. Lloyd, The serverless application analytics framework: Enabling design trade-off evaluation for serverless software, in: International Workshop on Serverless Computing, WoSC '20, ACM, Delft, Netherlands, 2020, pp. 67–72, http://dx.doi.org/10.1145/3429880.3430103.
[50] M. Orzechowski, M. Wrzeszcz, B. Kryza, Ł. Dutka, R.G. Słota, J. Kitowski, Global access to legacy data-sets in multi-cloud applications with onedata, in: Parallel Processing and Applied Mathematics: 14th International Conference, PPAM 2022, Gdansk, Poland, September 11–14, 2022, Revised Selected Papers, Part I, Springer, 2023, pp. 305–317.
[51] Pkgcloud, 2023, https://www.npmjs.com/package/pkgcloud. (Accessed: 03 June 2024).
[52] Libcloud, 2023, https://libcloud.apache.org/. (Accessed: 03 June 2024).
[53] Jcloud, 2023, https://jclouds.apache.org/. (Accessed: 03 June 2024).

Sashko Ristov, Ph.D., is an Assistant Professor at 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.

Simon Brandacher, BSc., is a master student in Software Engineering at the University of Innsbruck and received his Bachelor in computer science in 2022 from the University of Innsbruck, Austria. His main research interests include serverless computing and cloud federation.

Mika Hautz, BSc., is a master student in Software Engineering at the University of Innsbruck and received his Bachelor in computer science in 2021 from the University of Innsbruck, Austria. His main research interests include serverless computing, simulation, and cloud federation.

Michael Felderer, Prof., is the Director of the Institute for Software Technology at German Aerospace Center (DLR), a Full Professor at the University of Cologne, and an Associate Professor at the University of Innsbruck. His main research interests include Software Quality, Software Validation and Verification, Software Architecture, Security Engineering, and Empirical Software Engineering. Prof. Felderer co-authored more than 150 publications and received 12 best paper awards.

Ruth Breu, Prof., is a Full Professor at the University of Innsbruck. Her main research interests include Quality Engineering, Model Engineering, Security Engineering, Requirements Engineering, Software Development Processes, and Enterprise Architecture Management.