pyStorageLess: Leveraging Von Neumann's Architecture to Abstract Storage Heterogeneity in Serverless Applications

Sashko Ristov[1(✉)], Mika Hautz[1], Philipp Gritsch[1], Isabella Schmut[1], Peter Koll[1], and Michael Felderer[1,2,3]

[1] University of Innsbruck, Innsbruck, Austria
sashko.ristov@uibk.ac.at
[2] Institute of Software Technology, German Aerospace Center (DLR), Cologne, Germany
[3] University of Cologne, Cologne, Germany

Abstract. A novel approach pyStorageLess introduces storage interoperability for serverless functions in federated serverless infrastructures. The serverless functions are deployed only once, and the storage can be dynamically linked to the functions at runtime by the user through control data inputs while invoking the serverless functions. pyStorageLess uses the Von Neumann approach to abstract the function to have computing resources, memory, storage, and input/output data, regardless of the provider that hosts each part. pyStorageLess splits data inputs into value and control, allowing users to dynamically attach the storage at runtime. With such extreme flexibility, users may distribute parts of the function across different providers to improve performance.

Keywords: Federated clouds · FaaS · storage interoperability · serverless

1 Introduction

Function-as-a-Service (FaaS) is a widely used serveless computing model that allows the execution of serverless applications without maintaining the underlying cloud infrastructure. Recent approaches, such as federated Faas [14] and Sky computing [18], offer more cost-effective and faster execution of serverless applications. Various techniques exist to FaaSify a method [1,2,15] or a code block [9] and deploy it as a serverless function across multiple providers. Despite achieving a high level of abstraction in terms of FaaS portability, interoperability remains a challenge. Functions are not pure computing units and are not isolated [6]. The stateless nature of serverless computing necessitates storing state in persistent storage, or functions call other managed cloud services [3]. Migrating such non-isolated code from on-premises to the cloud or between cloud providers

Published by Springer Nature Switzerland AG 2025
C. Pahl et al. (Eds.): ESOCC 2025, LNCS 15547, pp. 171–178, 2025.
https://doi.org/10.1007/978-3-031-84617-5_14

often requires migrating the associated storage due to data locality [17], thereby adapting the SDKs to the target storage.

Unfortunately, the existing approaches for storage interoperability lag significantly behind solutions for function portability. For example, while the libraries, such as Apache Libcloud or jclouds, offer a common interface for storages of different providers, their abstraction is at the low level because developers still need to recode the function to change the provider and redeploy the updated function. Therefore, in this paper, we first investigate the weaknesses of the current storage interoperability methods, which motivate our novel approach for achieving storage interoperability at a high level of abstraction. We introduce a publicly available storage interoperability library pyStorageLess[1], which we used to develop several serverless workflows[2] utilizing our Abstract Function Choreography Language (AFCL) [4,13].

2 Von Neumann's Abstraction of the Serverless Function in PyStorageLess

The state-of-the-art approaches include open-source libraries such as Apache Libcloud or jclouds, or the Lithops storage API [16], which provide a single interface for accessing storages like AWS S3 or GCP Cloud Storage. Developers specify the driver for the respective storage, allowing for interoperable storage access without changing the interface. Figure 1 (left) presents the semi-dynamic, two-layered approach of Apache Libcloud. At the top layer, developers must select the driver (provider), and at the second layer, they use developer APIs for access and provisioning. While this approach allows developers to choose the specific driver, they must recode and redeploy functions if they want to change the driver. pyStorageLess (Fig. 1, right), on the other side, employs a different approach. At the top, pyStorageLess abstracts all providers and their regions, enabling developers to focus on their APIs instead of selecting the appropriate driver first. This approach allows for the dynamic selection of buckets from specific cloud providers at runtime without requiring recoding and redeployment of the functions.

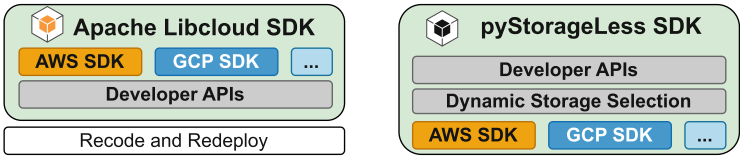

Fig. 1. Apache Libcloud 2-layers vs. pyStorageLess dynamic 3-layers.

[1] https://github.com/FaaSTools/pyStorage.
[2] https://github.com/AFCLWorkflows/.

We leverage Von Neumann's architecture to represent an abstract view of a serverless function that persist the state, i.e., accesses an abstract persistent storage, as depicted in Fig. 2. A serverless function comprises portable computing code that utilizes host memory, CPU, and the file system where it's deployed. During the deployment, the user selects the memory assigned to the function. Cloud providers often scale CPUs accordingly; for instance, AWS Lambda can utilize up to six CPUs when assigned the maximum allowed memory of 10 GB [8]. GCP, on the other side, allows developers to configure CPU resources of a function. The default hard drive is 512 MB, which is configurable during deployment. We denote all these portable and re-configurable computing resources with the gray lambda function in Fig. 2.

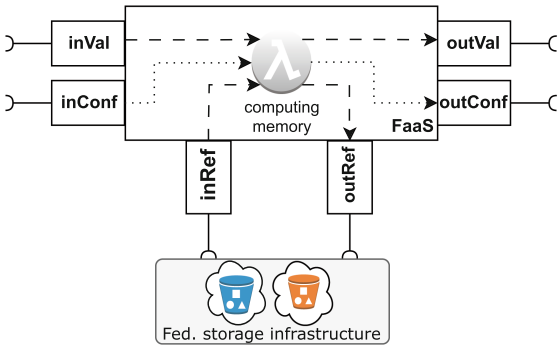

Fig. 2. Abstraction based on the Von Neumann's architecture.

Users invoke serverless functions with a payload, which is a key-value structure in JSON. Typically, functions receive small data explicitly *by value* through the payload when invoked. We denote this data port as `inVal`. Additionally, users pass large data implicitly *by reference* to files, which is provided in the payload. Unfortunately, existing libraries like Apache Libcloud restrict the storage to the provider whose driver is hardcoded in the function code. To address this limitation, we introduce an additional input data port `inConf`. Through `inConf`, users implicitly *configure* the driver that the function uses to access the referenced file. pyStorageLess parses the given URI in the `inConf` and automatically detects the required driver and the location of the target data (e.g., files). It's important to note that `inConf` is not only used for URIs for downloading large data but also to specify the drivers and locations for large data outputs. Based on the extracted drivers and locations, pyStorageLess uses a third data port, `inRef`, to access the given data *by reference*. This setup allows for more flexible and dynamic management of storage resources within serverless functions.

Once all data is placed in the file system or memory of the function, the processing begins. After processing finishes, the small data outputs of the computing are explicitly delivered to the output port `outVal` of the function. On

the other hand, large data outputs are first stored via the `outRef` data port at the location that was received from the `inConf` data port. Once the large output data is stored, references to the stored output data are implicitly delivered to the output port `outPort`. This final step enables other functions within the serverless application (e.g., serverless workflow) to use the data through their `inConf` data port. Cloud providers often restrict direct message passing between functions, and serverless workflows typically exchange large data through cloud object storage to overcome this limitation [7].

3 pyStorageLess Architecture and Implementation

pyStorageLess introduces an interoperable library that supports dynamic selection of storage attachments in serverless functions through control data inputs, as presented in Fig. 3. Developers can send bucket URL parameters as function data inputs and dynamically choose the bucket of the specific FaaS provider at runtime, rather than at the design phase (as in state-of-the-art Apache Libcloud and jclouds approaches). For example, if the function is invoked with source and destination URLs that refer to AWS S3 and GCP storages, respectively, the function will download the input data from AWS S3 and upload the results on GCP storage. Without any change in the function, the function can be now invoked with source and destination URLs that refer to GCP and AWS S3 storages, respectively, to download the input data from GCP storage and store the results in AWS S3.

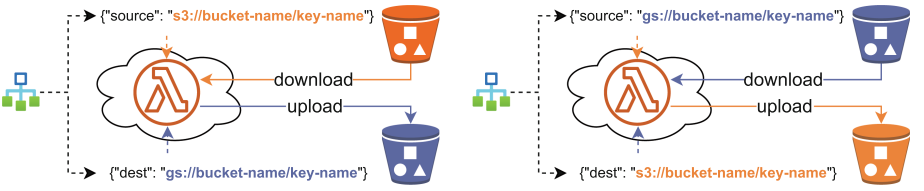

Fig. 3. Dynamic storage selection through the data inputs to the function.

Moreover, developers use a single `copy(source, dest)` method to download files from the bucket `source` to the function's local file system, upload results to the `dest` bucket, or copy from one storage to another, even across different providers. For the latter case, developers need to create two objects of Apache Libcloud, while pyStorageLess handles copies files with a single object.

Figure 4 presents an example of a function written in Python that downloads the file with the reference `source` and uploads it to a set of destinations `targets`. The developer imports the pyStorageLess as a library and codes the function to load the source and destination URLs (lines 6–7). In line 11, developer uses the copy interface to download the file from the source location. Further on, for each given target URL, developers code the function to upload the downloaded

file (line 14), with the same object and copy method. Note that developers simply read values from the event (function's payload) and pass the source and destination to the copy method, which enables dynamic selection of the storages.

```
1  import json
2  from storage.pyStorage import pyStorage
3
4  def lambda_handler(event, context):
5
6      source_url = event['source']
7      targets = event['dest']
8
9      response = []
10     local_filename = source.split('/')[-1]
11     pyStorage.copy(source, local_filename)
12
13     for t in targets:
14        x =  pyStorage.copy(local_filename, t)
15        response.append(x)
16
17     return {
18         'statusCode': 200,
19         'body': json.dumps(response)
20     }
```

Fig. 4. An example of a serverless function written in Python that downloads a file from the `source` and uploads it to multiple destination buckets `dest`. `source` and `dest` are provided at runtime and can span across various cloud providers.

3.1 Overview of Developer APIs

Figure 5 presents the high-level overview of the pyStorageLess architecture. pyStorageLess provides various APIs for dynamically attaching storage to serverless functions. For simplicity, Fig. 5 presents two APIs only. The interface `downloadFile(srcStorage, destLocal)` is used to download a file from the `srcStorage` and stores the file in the local file system of the function in the local file path `destLocal`. Unlike Apache Libcloud, where the developer needs to change the driver and change and redeploy the function, pyStorageLess allows the developer to pass the input data to the `srcStorage` at design time and then simply invoke the function with the URL of another provider at runtime. With this approach, `downloadFile()` attaches one or multiple abstract storage backends and dynamically selects the URIs of the files to be downloaded.

Similarly, the interface `uploadFile(srcLocal, destStorage)` uploads a file from the local file path `srcLocal` to an abstract storage `destStorage` that is attached to the function. The target storage is determined dynamically by the input data that is passed to `destStorage` at runtime, using the same procedure that was already explained for `downloadFile(srcStorage, destLocal)`. pyStorageLess supports three other interfaces for an entire bucket, such as `copy_bucket(source, target)`, `create_bucket(bucket_name, region)`, and `delete_bucket(bucket, delete_if_not_empty)`.

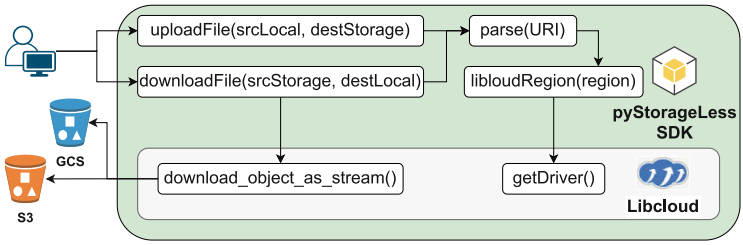

Fig. 5. pyStorageLess overview (partial view).

pyStorageLess extends Apache Libcloud with a mechanism to dynamically switch between different storage backends without impacting function execution. Figure 5 shows the main components and APIs of pyStorageLess. The pyStorageLess interfaces `downloadFile(srcStorage, destLocal)` and `uploadFile(srcLocal, destStorage)` call the `parse(URI)` with the respective parameters `srcStorage` and `destStorage`. Then, `parse(URI)` parses the given URI and determines the provider and its region. Since Apache Libcloud uses a separate driver for each AWS region, while GCP has only one driver for all regions, `get_driver()` is invoked with the translated Apache Libcloud region from the method `libloudRegion(region)`. This method takes an AWS region label as input and returns an Apache Libcloud region label. Finally, for uploading and downloading a file, pyStorageLess uses the Apache Libcloud methods, e.g., the `download_object_as_stream()` for downloading a file.

3.2 pyStorageLess in AFCL Workflows

The input data in the AFCL workflow json file can specify the location of input data (AWS S3 or GCP), that is, by replacing the field `<bucket>` from Fig. 6 with either AWS S3 or GCP cloud storage URI. Additionally, inside the workflow json, one can update the respective locations of the functions that should be invoked, either on AWS or GCP. With this approach, users can dynamically attach storages to workflow functions each time the AFCL workflow is invoked.

```
{
    "key_input": "<bucket>/input/ALL.chr22.80000.vcf.gz",
    "key_columnsfile": "<bucket>/input/columns.txt",
    "output_bucket": "<bucket>",
    "AFR": "<bucket>/input/AFR",
    "ALL": "<bucket>/input/ALL",
    "AMR": "<bucket>/input/AMR",
    "EAS": "<bucket>/input/EAS",
    "EUR": "<bucket>/input/EUR",
    "GBR": "<bucket>/input/GBR",
    "SAS": "<bucket>/input/SAS"
}
```

Fig. 6. Example of the workflow input that is sent to the xAFCL serverless workflow management system to attach the storages dynamically during runtime.

4 Conclusion and Future Work

The contributions of pyStorageLess's approach, enabling users to manage storage dynamically at runtime, are manifold. Firstly, it facilitates the development of portable functions with interoperable storage, enhancing flexibility and reducing vendor lock-in. Secondly, it expands data access capabilities with an interoperable method, `copy(source, dest)`, which simplifies file operations by requiring only a single object, regardless of whether it involves downloading, uploading, or copying files between storage providers. Thirdly, pyStorageLess allows users to deploy identical functions across both AWS and GCP, enabling dynamic selection of storage drivers and data input/output locations. Users can choose between AWS S3 or GCP Cloud Storage for their data inputs and outputs, enhancing flexibility and interoperability across multiple cloud providers. Lastly, community can utilize other serverless workflows (Montage, BWA, Genome1000, celebrityCollage [10], etc.), whose functions are coded with pyStorageLess and allow interoperable storage in federated FaaS.

We will extend our work into several directions. First, we will introduce runtime mechanisms in pyStorageLess to dynamically select the provider of the destination target storage by considering data locality. Second, we will develop a multi-objective scheduler (e.g., as an extension of our FaaSt scheduler [11]) to optimize the conflicting objectives cost and performance. Third, we will consider dynamic bandwidth between functions and storage, which depends on the assigned memory to the functions [5]. Finally, we will extend our SimLess mathematical model [12] to reuse data from not only computing part of a function, but also data access to storages in federated FaaS.

Acknowledgments. This work was supported by Land Tirol under the contract F.35499 (TIM) and KDT JU (grant agreement 101140216, MATISSE).

References

1. Carvalho, L., de Araújo., A.P.F.: 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 (2020)
2. Cordingly, R., et al.: The serverless application analytics framework: enabling design trade-off evaluation for serverless software. In: International Workshop on Serverless Computing (WoSC 2020), pp. 67–72 (2020)
3. Eismann, S., et al.: The state of serverless applications: collection, characterization, and community consensus. IEEE Trans. Softw. Eng. **48**(10), 4152–4166 (2022)
4. Hautz, M., Ristov, S., Felderer, M.: Characterizing AFCL serverless scientific workflows in federated FaaS. In: International Workshop on Serverless Computing, WoSC 2023, pp. 24–29. ACM, Bologna (2023)
5. Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., Kozyrakis, C.: Pocket: elastic ephemeral storage for serverless analytics. In: Symposium on Operating Systems Design and Implementation (OSDI 2018), pp. 427–444. USENIX Association, Carlsbad (2018)

6. Larcher, T., Gritsch, P., Nastic, S., Ristov, S.: BaaSLess: backend-as-a-service (baas)-enabled workflows in federated serverless infrastructures. IEEE Trans. Cloud Comput. 1–15 (2024)
7. Mahgoub, A., Shankar, K., Mitra, S., Klimovic, A., Chaterji, S., Bagchi, S.: SONIC: application-aware data passing for chained serverless applications. In: Annual Technical Conference (ATC 21), pp. 285–301. USENIX Association (2021)
8. Mahgoub, A., Yi, E.B., Shankar, K., Elnikety, S., Chaterji, S., Bagchi, S.: ORION and the three rights: sizing, bundling, and prewarming for serverless DAGs. In: Symposium on Operating Systems Design and Implementation (OSDI 2022), pp. 303–320. USENIX Association, Carlsbad (2022)
9. Pedratscher, S., Ristov, S., Fahringer, T.: M2FaaS: transparent and fault tolerant FaaSification of node.js monolith code blocks. Future Gener. Comput. Syst. **135**, 57–71 (2022)
10. Ristov, S., Brandacher, S., Hautz, M., Felderer, M., Breu, R.: CODE: code once, deploy everywhere serverless functions in federated FaaS. Futur. Gener. Comput. Syst. **160**, 442–456 (2024)
11. Ristov, S., Gritsch, P.: FaaSt: optimize makespan of serverless workflows in federated commercial FaaS. In: International Conference on Cluster Computing, CLUSTER 2022, pp. 182–194. IEEE, Heidelberg (2022)
12. Ristov, S., Hautz, M., Hollaus, C., Prodan, R.: SimLess: simulate serverless workflows and their twins and siblings in federated FaaS. In: Symposium on Cloud Computing, SoCC 2022, pp. 323–339. ACM, San Francisco (2022)
13. Ristov, S., Pedratscher, S., Fahringer, T.: AFCL: an abstract function choreography language for serverless workflow specification. Fut. Gen. Comp. Syst. **114**, 368–382 (2021)
14. Ristov, S., Pedratscher, S., Fahringer, T.: xAFCL: run scalable function choreographies across multiple FaaS systems. IEEE Trans. Serv. Comput. **16**(1), 711–723 (2023)
15. Ristov, S., Pedratscher, S., Wallnoefer, J., Fahringer, T.: DAF: dependency-aware FaaSifier for node.js monolithic applications. IEEE Softw. **38**(1), 48–53 (2021)
16. Sampé, J., Sánchez-Artigas, M., Vernik, G., Yehekzel, I., García-López, P.: Outsourcing data processing jobs with lithops. IEEE Trans. Cloud Comput. **11**(1), 1026–1037 (2023)
17. Sethi, B., Addya, S.K., Bhutada, J., Ghosh, S.K.: Shipping code towards data in an inter-region serverless environment to leverage latency. J. Supercomput. **79**(10), 11585–11610 (2023)
18. Yang, Z., et al.: SkyPilot: an intercloud broker for sky computing. In: Symposium on Networked Systems Design and Implementation (NSDI 23), pp. 437–455. USENIX Association, Boston (2023)