



**CloudButton**



**HORIZON 2020 FRAMEWORK PROGRAMME**

**CloudButton**

(grant agreement No 825184)

**Serverless Data Analytics Platform**

## **D3.3 Serverless Compute Engine Reference Implementation**

Due date of deliverable: 10-06-2022

Actual submission date: 2022

Start date of project: 01-01-2019

Duration: 36 months

## Summary of the document

<b>Document Type</b>	Report
<b>Dissemination level</b>	Public
<b>State</b>	v1.0
<b>Number of pages</b>	54
<b>WP/Task related to this document</b>	WP3 / T3.1, T3.2, T3.3, T3.4
<b>WP/Task responsible</b>	IBM
<b>Leader</b>	Gil Vernik (IBM)
<b>Technical Manager</b>	Peter Pietzuch (Imperial)
<b>Quality Manager</b>	Josep Sampé (URV)
<b>Author(s)</b>	Gil Vernik, Josep Sampe, Pedro García, Rut Palmero, Carlos Segarra, Bo Zhao, Guo Li
<b>Partner(s) Contributing</b>	IBM, URV, ATOS
<b>Document ID</b>	CloudButton_D3.3_Public.pdf
<b>Abstract</b>	This document describes the final architecture design and the final implementation of the CloudButton platform for data intensive computations.
<b>Keywords</b>	FaaS, serverless, Kubernetes, hybrid cloud, workflow orchestration, data intensive computations, SLA monitoring.

## History of changes

Version	Date	Author	Summary of changes
0.1	17-05-2022	Gil Vernik (IBM)	Table of Contents
0.2	19-05-2022	Roi Sucasas (Atos)	SLA and monitoring
0.3	22-05-2022	Gil Vernik (IBM)	Final design and implementation
0.4	23-05-2022	Carlos Segarra (IMP), Simon Shillaker (IMP)	C++, WebAssembly, and Faasm
0.5	30-05-2022	Carlos Segarra (IMP), Guo Li (IMP), Bo Zhao (IMP)	Reivew Faasm text and update Faasm+Lithops integration
0.6	31-05-2022	Aitor Arjona (URV), Pedro García (URV)	Add Triggerflow and Lithops for Apache Airflow sections
0.7	01-07-2022	Gil Vernik (IBM)	Updates and corrections
0.8	02-07-2022	Pierre Sutra (IMT)	Updates.
1.0	03-07-2022	Gil Vernik (IBM)	Final version.

## Table of Contents

<b>1</b>	<b>Executive summary</b>	<b>2</b>
<b>2</b>	<b>Motivation and Background</b>	<b>4</b>
2.1	Serverless Computing Overview . . . . .	4
2.2	Beyond the Current Serverless Development Experience . . . . .	5
2.3	Hybrid Cloud . . . . .	6
2.4	Performance Acceleration . . . . .	7
2.5	Overall Objectives . . . . .	7
<b>3</b>	<b>State of the Art</b>	<b>8</b>
3.1	Workflow orchestrates . . . . .	8
3.2	Serverless beyond Function as a Service . . . . .	9
<b>4</b>	<b>Final design and Implementation of the Serverless Compute Engine for Big Data</b>	<b>10</b>
4.1	Lithops general architecture and design . . . . .	10
4.2	No vendor lock-in and multi cloud portability . . . . .	12
4.2.1	Multiple APIs . . . . .	12
4.2.2	Futures API . . . . .	12
4.2.3	Multiprocessing API . . . . .	12
4.2.4	Storage API . . . . .	14
4.2.5	Storage OS API . . . . .	14
4.3	Multiple Storage backends and Big Data processing . . . . .	14
4.3.1	Data discovery and data partitioner . . . . .	15
4.4	Serverless without limits . . . . .	16
4.4.1	Localhost execution mode . . . . .	16
4.4.2	Serverless execution mode . . . . .	16
4.4.3	Standalone execution mode . . . . .	17
4.5	Serverless without constraints and hybrid workloads . . . . .	17
4.6	LithopsCloud CLI tool . . . . .	19
4.7	Temporary data . . . . .	19
4.8	CloudObject to share results and maintain state . . . . .	19
4.9	Cost effective serverless model for Big Data analytics . . . . .	20
<b>5</b>	<b>Lithops in the broad scope of CloudButton</b>	<b>21</b>
5.1	Crucial . . . . .	21
5.1.1	Overview . . . . .	21
5.1.2	Integration with Lithops . . . . .	22
5.2	WebAssembly . . . . .	23
5.2.1	Format . . . . .	23
5.2.2	Linear memory . . . . .	24
5.2.3	Toolchains and runtimes . . . . .	24
5.2.4	WASI: the WebAssembly system interface . . . . .	25
5.2.5	Future WebAssembly development . . . . .	25
5.3	C++ - Faasm . . . . .	25
5.3.1	Faasm integration with Lithops . . . . .	25
5.3.2	Integration with RedHat and Infinispan . . . . .	26
<b>6</b>	<b>Serverless Workflows</b>	<b>27</b>
6.1	Triggerflow . . . . .	27
6.2	Apache Airflow . . . . .	30

<b>7</b>	<b>SLA Monitoring and Management</b>	<b>33</b>
7.1	Rational behind the CloudButtonSLA component . . . . .	33
7.2	Lithops and CloudButtonSLA integration . . . . .	35
7.2.1	Lithops metrics . . . . .	35
7.2.2	PromSQL queries . . . . .	36
7.2.3	CloudButton agreements and Swagger API . . . . .	38
7.2.4	Integration with Lithops through Rabbit queue . . . . .	40
7.2.5	Prometheus Pushgateway Violation notification . . . . .	41
7.2.6	A full sample of context . . . . .	42
7.2.7	CloudButtonSLA cost control panel in Grafana . . . . .	46
7.3	Predicted metrics with Prometheus holt_winters and predictor_linear . . . . .	47
<b>8</b>	<b>Summary, conclusions and the next steps</b>	<b>48</b>
8.1	Applications . . . . .	48
8.2	Adoption of the platform by 3rd party developers . . . . .	49
8.3	Next steps . . . . .	49

## List of Abbreviations and Acronyms

<b>ADF</b>	Azure Durable Functions
<b>API</b>	Application programming interface
<b>ASF</b>	Amazon Step Functions
<b>CD</b>	Continued Development
<b>CLI</b>	Command-line interface
<b>CNCF</b>	Cloud Native Computing Foundation
<b>COS</b>	Cloud Object Storage
<b>CPU</b>	Central Processing Unit
<b>CRC</b>	Custom Resource Controller
<b>CRD</b>	Custom Resource Definition
<b>DAG</b>	Directed Acyclic Graph
<b>DSL</b>	Domain Specific Lanaguage
<b>ETL</b>	Extract, Transform, Load
<b>FaaS</b>	Function as a Service
<b>FDR</b>	False Discovery Rate
<b>GPU</b>	Graphics Processor Unit
<b>GUI</b>	Graphical User Interface
<b>HPSC</b>	High Performance Serverless Compute Engine
<b>HTTP</b>	Hypertext Transfer Protocol
<b>ICT</b>	Information and Communication Technology
<b>JSON</b>	JavaScript Object Notation
<b>K8s</b>	Kubernetes
<b>PaaS</b>	Platform as a Service
<b>QoS</b>	Quality of Service
<b>REST</b>	Representational State Transfer
<b>SDK</b>	Software Development Kit
<b>SLA</b>	Service Layer Agreement
<b>SLO</b>	Service Layer Objective
<b>SOTA</b>	State of the art
<b>UI</b>	User interface
<b>VM</b>	Virtual Machine
<b>YAML</b>	YAML Ain't Markup Language

## 1 Executive summary

Cloud-native transformation is happening in the field of data intensive computations. At the core of this transformation, there is a microservices architecture with container (e.g., Docker) and container orchestrating (e.g., Kubernetes) technologies powering up the microservices approach. One of the most important recent developments in the cloud-native movement is "serverless" (also known as Function-as-a-Service (FaaS)) computing. FaaS holds two main promises for data intensive computations: (a) massive just in time parallelism at a fraction of the cost of an always-on sequential processing and (b) lower barriers for developers who need to focus only on their code and not on the details of the code deployment.

To fully leverage FaaS potential for the data intensive computations, a simplified consumption model is required, so that a data scientist, who is not familiar with the cloud computing details in general and FaaS in particular, could seamlessly leverage FaaS from her program written in a high level programming language, such as Python.

Nowadays data is not located in one physical place in an enterprise. Rather, data is distributed over a number of clusters in a private cloud with some data and other resources being in the public cloud(s). This gives the rise to the architectural approach known as *hybrid cloud*. In this approach data and computational resources are federated over a number of clusters/clouds, so that logically they can be accessed in a uniform and cost-efficient way. The peculiarities of the hybrid cloud should be transparent to the data scientist who works at the higher level of abstraction, treating the federation as something that can be accessed and used as a "whole".

Modern intensive data computations take form of complex *workflows*. Usually, these workflows are not limited to serverless computations, but include multiple parallel and sequential steps across the hybrid cloud, where the flow of control is driven through events of different nature. Serverless computations are ephemeral by nature, but flows require state and complement serverless computations in this respect. It is paramount that the flows designed by the data scientists allow to glue together serverless and "serverfull" functionalities. We refer to this model as "*servermix*".

To support the servermix model cost-efficiently in the hybrid cloud, a number of challenges pertaining to portability, flexibility, and agility of a servermix computational engine should be solved.

This document describes our progress with the architecture design and the final implementation of the Serverless Compute Engine for Big Data platform for data intensive computations. The Serverless Compute Engine platform comprises five main parts:

- **Developer Toolkit and APIs:** a developer/data scientist facing component (a client environment) that executes functionality expressed in a high level programming language, such as Python, transparently leveraging parallelism of serverless as part of the data intensive computational workflows through submitting jobs to the Serverless Compute Engine;
- **Serverless Compute Engine for Big Data:** A suite for high performance Compute Engine optimized for running massively parallel data intensive computations and orchestrating them as a part of the stateful data science related workflows. This component includes Lithops core components, implements scheduling logic, multitenancy, workflow orchestrations, and SLA management;
- **Backend serverless (i.e., FaaS) Framework:** this is a pluggable component that can have many implementations (e.g., public cloud vendor serverless offering, Kubernetes serverless framework, a standalone serverless solution, etc.)
- **Persistent Storage Service:** this is a pluggable component that is provided independently from the Serverless Compute Engine platform (e.g., Cloud Object Storage (COS))
- **Caching Service:** this is another independently deployed component providing caching services for the Serverless Compute Engine (e.g., Infinispan [1]).

In this specification, we focus on the first two functional components. We start from a brief introduction of the main concepts and programming paradigms involved in our solution in Section 2. Next we briefly describe SOTA in Section 3 and proceed to laying out the architecture of the Serverless Compute Engine platform that is based on the Lithops open source framework [2] and also contains SLA with Monitoring. The architecture follows a cloud-native microservices based approach.

In Section 4 we describe our final design and implementation of the serverless compute engine. We demonstrate how the public cloud services are leveraged in our architecture. The Lithops framework we developed is an open source project that we released for broad usage and adoption. Lithops is the main component of Serverless Compute Engine for Big Data and all tasks in WP3 (T3.4, T3.1, T3.2, T3.3) are using Lithops, like High Performance Serverless Compute Engine that also contains advanced SLA and monitoring component (will be explained in section 7).

In Section 5 we explain how tools developed in this work package are being widely used by all partners in the CloudButton project. In particular we explain how Lithops being used in context of Crucial and Java programming, Shell scripts, WebAssembly and C++ Faasm.

Finally we will explain adoption of the platform by 3rd party developers and provide numerous applications where Lithops fits with some of the next steps.



## 2 Motivation and Background

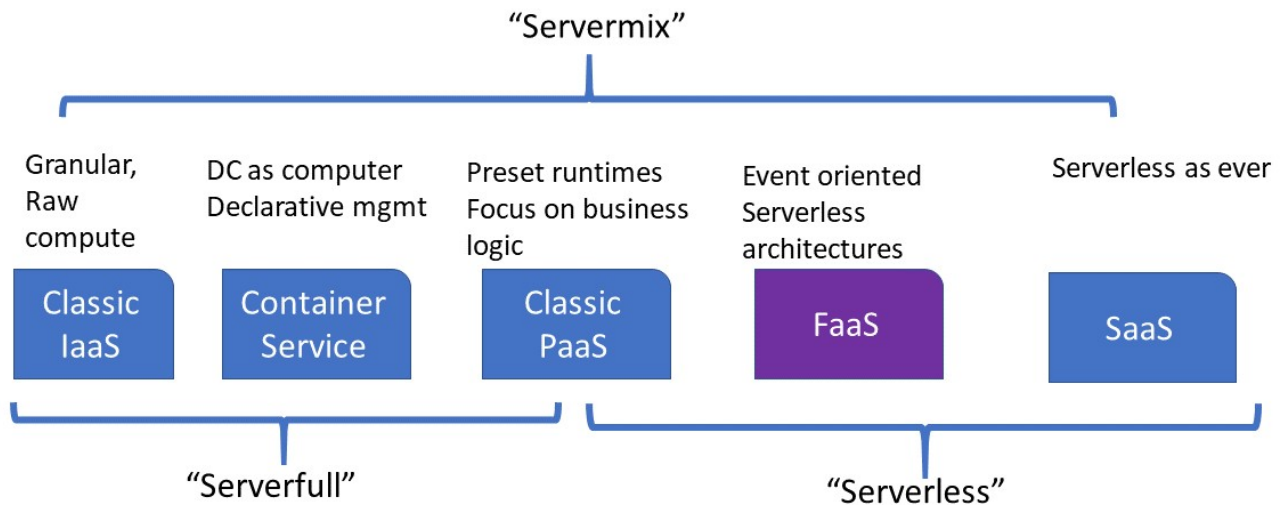


Figure 1: Serverless Taxonomy

### 2.1 Serverless Computing Overview

The serverless programming model, also known as Function-as-a-Service (FaaS<sup>1</sup>) has gained considerable momentum since its introduction in 2014 [3]. The term serverless is somewhat misleading. The term does not imply that no servers are involved in running an application. Rather it hints at a level of abstraction, which allows to ignore deployment details (e.g., servers configuration and maintenance) and focus exclusively on the application code. Figure 1 positions FaaS on the spectrum of programming models. FaaS can be viewed as a specialized Platform-as-a-Service (PaaS) taking care of all deployment and run-time issues and relieving the developer from any concerns related to server provisioning and maintenance.

There are several main principles pertaining to FaaS, which are universally applied across a variety of implementations:

- A unit of execution is a function written in a high-level programming language;
- A function is executed in response to an event (which also can be an HTTP call);
- Rules and triggers can be defined to bind functions and events together, so FaaS is an intrinsically event driven programming model;
- A customer is charged only for the resources used during the function execution (at a very fine granularity: typically, being on the order of 100 ms);
- Functions are transparently auto-scaled horizontally to be instantaneously elastic: i.e., the load balancer is built into a platform and new functions are started in response to events as needed. Some system limits, such as maximum number of simultaneous invocations per user and invocations/sec per user are usually enforced in FaaS implementations;
- Functions are ephemeral (i.e., stateless)<sup>2</sup>

<sup>1</sup>It can be argued that FaaS pertains to delivering serverless computations as a metered and billed cloud service to the customers. In this document we will use the terms serverless and FaaS interchangeably unless this results in a confusion.

<sup>2</sup>FaaS extensions, such as AWS Step Functions [4] and Azure Durable Functions [5] allow to maintain state in the serverless computations. The mechanisms used in these solutions are fairly different. The former implements long running state machines and the latter uses event scoping.

- Functions can be chained with the output of one action being the input of another;
- Functions can be orchestrated to execute in complex application topologies<sup>3</sup>
- There is no server administration or provisioning;
- Users typically have to select a function *flavor* (i.e., amount of memory and CPU – unless allocated proportionally to memory by default) upon the function deployment;
- Functions can execute both synchronously and asynchronously.

Serverless computing is a very attractive choice for big data computations, where data parallelism exists since it offers tremendous potential for ease-of-use, instantaneous scalability and cost effectiveness providing low cost access to hundreds and thousands of CPUs, on demand, with little or no setup.

To illustrate this point, consider a geospatial use case of CloudButton. A satellite image object can be partitioned into sub-images and a separate function can be assigned to process each sub-image (e.g., apply object classification model on a sub-image). These functions can be run in parallel as a single map step. For the details of the CloudButton use cases and how they render themselves to serverless computing see deliverable D2.1 and D3.1.

## 2.2 Beyond the Current Serverless Development Experience

A basic FaaS development cycle is as follows. A developer writes a function using her favorite text editor. Then she *creates* the function (i.e., register it in the platform's data base) using a CLI or a Web based GUI. Upon creation the function receives a name, by which it can be bound to event triggers and rules that cause function invocation in response to the events represented by the triggers.

The reader is referred to IBM Cloud Functions tutorial [11] and Apache OpenWhisk community resources [12] for a detailed step by step examples of serverless programming with Apache OpenWhisk, as a typical example of the serverless programming experience

In their inspirational paper [13], the authors observed that even this apparently simple development cycle is too complicated for most scientists who prefer focusing on their domain rather than on mastering a new programming paradigm. This complexity prevents the scientists from leveraging the advantages of the serverless computing.

Data scientists need a flexible environment where they can run their simulations while not worrying about resources that the simulations may require. While serverless is the right solution to make e.g., AI flows more efficient — many potential users are unsure of what is involved and required to make this happen in their scientific applications. Simplifying development experience by provide data scientists with the "push to the cloud" functionality is the primary goal of the CloudButton project. To this end, we focus on how to connect an existing code and frameworks to serverless without the painful process of starting from scratch, redesigning applications or learning new skills. Since serverless computing provides great benefit for HPC workloads (e.g., embarrassingly parallel Monte Carlo simulations), Big Data analytics and AI frameworks, it is important to make sure that users can easily integrate serverless with the frameworks and programming languages of their choice.

Furthermore, in the real world big data applications, the applications are rarely a single computational step, which can be reduced to a serverless function call or a number of calls performed in a loop (parallelism). Rather than that, typical big data analytics involves multiple steps that should be coordinated and orchestrated seamlessly. Consuming serverless computations from a cloud (either centralized or hybrid) is ultimately an exercise in distributed computing. And distributed computing is notoriously hard. In most cases, it is totally out of the data scientist skills to develop an efficient and robust code for orchestrating distributed serverless computation.

---

<sup>3</sup>The orchestrators are typically external to the FaaS frameworks. Apache Composer [6] is an exception, since it allows to execute a function composition as it was a function in Apache OpenWhisk. Important examples of the orchestrating technology include Airflow [7], KubeFlow [8], Argo Flows [9], Fission Workflows [10]. We performed evaluation of some of these technologies towards their possible use in the CloudButton platform and will discuss some of them later on in this document.

As a simple example, consider face alignment in facial recognition workloads. The process of aligning an image is fairly straightforward and can be done using the Dlib library [14] and its face landmark predictor. Only a few lines of Python code are required to apply the face landmark predictor to preprocess a single image. However, processing millions of images stored in the cloud (e.g., in the Cloud Object Storage (COS), a massively used cost-efficient storage solution both for structured and unstructured data), is far from trivial.

To start with, a lot of boilerplate code is required to deal with locating the images inside COS and accessing them for read and write. Next, there should be code dealing with data partitioning, function invocations, collection of the results, restarting of failed functions, traffic shaping (i.e., adhering to the system limits, such as functions/sec rate), and informing the next computational stage in a pipeline about the results of the current one when it finishes. In addition, some external services might be required to complete different computational tasks and pass information.

This brings the notions of the *servermix* workflows, workflow orchestration, and their integration with serverless to the forefront, making them of critical importance for data intensive pipelines<sup>4</sup>.

In the system we envision, a data scientist develops her code in a high level language such as Python (as it was before), the code is automatically translated into a DAG of tasks and these tasks are being executed on a backend FaaS system with all the boilerplate functionality pertaining to the workflow orchestration executing transparently. The data scientist will be able to provide scheduling hints to the system specifying the target FaaS service of her choice and SLO pertaining parameters.

### 2.3 Hybrid Cloud

As some recent studies show [15], enterprises have unique requirements to cloud computing, which prevents many of the enterprise workloads to be seamlessly moved to the public cloud. As we go to press, it is estimated that on average only 20% of the enterprise workloads are currently in the cloud, with 80% still being on premises. For the enterprises the cloud does not mean a traditional centralized cloud anymore. To start with, even a traditional "centralized" cloud is actually a distributed one with multiple geographically disparate regions and availability zones and enterprise data scattered among them. Moreover, nowadays, most enterprises use multi-cloud strategy for their ICT [16] with each cloud being distributed. On the private cloud side, even a medium size enterprise has more than one compute cluster today and more than one storage location and the computations should be pertained in a distributed manner across these clusters and data silos. The public and private cloud usage trends culminate in enterprises engaging in the hybrid cloud deployments with multiple multi-regional public clouds and multi-cluster private cloud federated together in some form allowing concerted workload execution.

With the hybrid cloud model on the rise, enterprises face a number of non-trivial challenges with arguably the most compelling one being portability. To allow for portability applications have to be developed in a certain way known as cloud-native [17], which make them ready for cloud deployment in the first place (among other features cloud-nativeness implies containerization of the application components). Another necessary condition is cloud agnosticism in the control plane related to supporting the DevOps cycle of the application. To this end, a number of container orchestrator have been tried by the cloud software development community over the last few years [18, 19, 20] with CNCF's Kubernetes (K8s) [20] being a market leader today.

K8s provides PaaS for declarative management of containers. Containerized cloud-native applications are seamlessly portable across K8s clusters that can also be federated. Thus, K8s becomes a de-facto standard for the enterprise hybrid PaaS.

In the K8s environment, serverless functions are essentially pods (a unit of scheduling in K8s) executing containers with potentially multiple containers per pod, where the higher level of abstraction, which is a "function", is provided by some developer facing shim to insulate the developers from the low level K8s APIs.

A number of serverless platforms and building blocks as well as K8s native workflow manage-

---

<sup>4</sup>We discuss servermix model at length in Deliverable D2.1 in the context of the CloudButton use cases and overall platform architecture.

ment frameworks have appeared recently. We will briefly review the more important of them in the next section. In addition, K8s provides mature frameworks for service meshes, monitoring, networking, and federation. An important feature of K8s is its extensibility. Through the Custom Resource Definition (CRD) and Custom Resource Controller (CRC) mechanisms, K8s can be extended with additional resources (originally non-K8s) that are added to the control plane and managed by the K8s API. This mechanism is used by "K8s native" workflow management tools, such as Argo [9] and KubeFlow [8] to manage complex workflows in a federated K8s environment.

As an intermediate summary, the servermix workflows in the K8s based hybrid cloud boils down to orchestrating pods. K8s is an event-driven management system and its scheduling mechanism includes hooks for extension. This is helpful in our approach to developing the CloudButton platform, because it allows to add smartness to e.g., scheduling decisions taken by K8s w.r.t. pods comprising a workflow.

## 2.4 Performance Acceleration

Originally, serverless use cases were focusing on event driven processing. For example, an image is uploaded to the object storage, an event is generated as a result that automatically invokes serverless function which generates a thumbnail. As serverless computing become mainstream, more use cases start benefiting from the serverless programming paradigm. However, current serverless models are all stateless and do not have any innate caching capabilities to cache frequently access data. In the CloudButton project, we will explore the benefit of a caching layer and how it can improve serverless workflows. The data shipping model permeates serverless architectures in public, private, and hybrid clouds alike and gravely affecting their performance.

Consider a user function that takes input data and applies an ML model, stored in the object storage. Executing this function at a massive scale as a serverless computation against various data sets will require each invocation to use the same exact ML model. However, if there no caching used, each function will have to read the model from the remote storage each time it runs, which is both expensive and slow. Having a cache layer will enable to store ML model in the cache, as opposite to each invocation try to get the same model from some shared storage, like object storage. Metabolomics use case has various level of caching, where they store molecular databases. Likewise, in the Geospatial pipelines, there are multiple opportunities for improving performance through caching.

In CloudButton, we plan to explore the tradeoffs between the local per-node cache, such as Plasma Object Storage (from Apache Arrow) [21] and cluster based caching, such as Infinispan [1], and develop an architecture that would be able to accommodate the two approaches and balance between them for cost-efficiency and performance improvements.

## 2.5 Overall Objectives

From examining the hybrid cloud features, it is easy to see that in order to be able to cater for the multiple deployment options, and therefore aim at maximum traction with the community, the CloudButton platform should be cloud-native itself, because this approach is highly modular and extensible and allows to gradually build an ecosystem around CloudButton. Indeed, as we explain in Section 4, we follow the cloud-native microservices based approach to the CloudButton architecture.

In general, we simultaneously target two different approaches predicated on the level of control of the backend FaaS framework used to execute serverless workloads. In case of the public cloud, this control is limited, which reduces CloudButton scheduling to relatively simple algorithms (mostly focusing on pooling capacity across clouds and/or cloud regions) with no ability to guarantee SLO/SLA for the workloads, but rather resorting to general improvements, such as caching to improve overall performance of the platform. In case of the K8s hybrid cloud based deployment, the level of control is much higher and the CloudButton components related to scheduling and SLA/SLO enforcement can be much more sophisticated. To capture these different approaches within the same architecture, we define APIs for the services that will implement them and provide different "plugins" suitable for different deployment constellations, thus also opening a door for third party FaaS

backend plugins, schedulers, orchestrators, runtime systems, user facing clients, etc.

From the exploitation perspective, we aim at creating at least two levels for the project: a light weight "community edition" with only minimal functionality that would be suitable to run relatively small workloads by a single user and an "enterprise edition" aiming at multi-tenant, production-worthy deployments.

### 3 State of the Art

#### 3.1 Workflow orchestrates

FaaS is based on the event-driven programming model. In fact, many event-driven abstractions like triggers, Event Condition Action (ECA) and even composite event detection were already inspired by the veteran Active Database Systems [22].

Event-based triggering has also been extensively employed in the past to provide reactive coordination of distributed systems [23, 24]. Event-based mechanisms and triggers have also been extensively used [25, 26, 27, 28] in the past to build workflows and orchestration systems. The ECA model including trigger and rules fits nicely to define the transitions of finite state machines representing workflows. In [29], they propose to use synchronous aggregation triggers to coordinate massively parallel data processing jobs.

An interesting related work is [28]. They leverage composite subscriptions in content-based publish/subscribe systems to provide decentralized Event-based Workflow Management. Their PADRES system supports parallelization, alternation, sequence, and repetition compositions thanks to content-based subscriptions in a Composite Subscription Language.

More recently, a relevant article [30] has surveyed the intersections of the Complex Event Processing (CEP) and Business Process Management (BPM) communities. They clearly present the existing challenges to combine both models and describe recent efforts in this area. We outline that our paper is in line with their challenge "Executing business processes via CEP rules", and our novelty here is our serverless reactive and extensible architecture.

In serverless settings, the more relevant related work aiming to provide reactive orchestration of serverless functions is the Serverless trilemma [31] from IBM. In their paper, the authors advocate for reactive run-time support for function orchestration, and present a solution for sequential compositions on top of Apache OpenWhisk.

A plethora of academic works are proposing different so-called serverless orchestration systems like [32, 33, 34, 35, 36, 37]. However, most of them rely on non-serverless services like VMs or dedicated resources, or they use functions calling functions patterns which complicate their architectures and fault tolerance. None of them offer extensible trigger abstractions to build different schedulers.

All Cloud providers are now offering cloud orchestration and function composition services like IBM Composer, Amazon Step Functions, Azure Durable Functions, or Google Cloud Composer.

IBM Composer service is in principle designed for short-running synchronous composition of serverless functions. IBM Composer generates a state machine representation of the workflow to be executed with IBM Cloud Functions. It can represent sequences, conditional branching, loops, parallel, and map tasks. However, fork/join synchronization (map, parallel) blocks on an external user-provided Redis service, limiting their applicability to short running tasks.

Amazon offers two main services: Amazon Step Functions (ASF) and Amazon Step Functions Express Workflows (ASFE). The Amazon States Language (based on JSON) permits to model task transitions, choices, waits, parallel, and maps in a standard way. ASF is a fault-tolerant managed service designed to support long-running workflows and ASFE is designed for short-running (less than five minutes) highly intensive workloads with relaxed fault-tolerance.

Microsoft's Azure Durable Functions (ADF) represents workflows as code using C# or Javascript, leveraging async/await constructs and using event sourcing to replay workflows that have been suspended. ADF does not support map jobs explicitly, and only includes a `Task.whenAll` abstraction enabling fork/join patterns for a group of asynchronous tasks.

Google offers Google Cloud Composer service leveraging a managed Apache Airflow cluster.

Airflow represents workflows in a DAG (Directed Acyclic Graph) coded in Python, so that it cannot support cycles. It is not ideally suited for parallel jobs or high-volume workflows, and it is not designed for orchestrating serverless functions.

Two previous papers [38, 39] have compared public FaaS orchestration services for coordinating massively parallel workloads. In those studies, IBM Composer offered the fastest performance and reduced overheads to execute map jobs whereas ASF or ADF imposed considerable overheads. We will also show in this paper how ASFE obtains good performance for parallel workloads.

None of the existing cloud orchestration services is offering an open and extensible trigger-based API enabling the creation of custom workflow engines. Our work on Triggerflow tries to fill this gap, offering a tool to implement existing models like ASF or Airflow DAGs with reactive schedulers leveraging Knative standard technologies.

### **3.2 Serverless beyond Function as a Service**

Initially FaaS and serverless shared the same definition and the same goal to enable users to execute their code or flows without manually setup VMs with exact resources required. Initially FaaS also considered as a "glue" to tie multiple services, so that various events could trigger invocation of the user's functions. For example, user may define an event that invokes a function in case new row added to the database. However today serverless has grown beyond traditional FaaS and now considered as a broad user experience of getting right resources for their workloads, where user focus on the business logic and deploy their workload to the cloud while cloud provide the right resources without user explicitly specify them. As example, consider a user who need to run ML workload that need large amounts of GPUs. Traditionally, with a serverfull approach, user will get to his cloud account, perhaps using UI and provision right set of VMs with GPUs. Then deploy the ML workload, monitor VMs, collect the results when job completed and then shut down VMs or perhaps keep them idle and deploy another workload. As opposite, with serverless approach, we believe user should use "push to the cloud approach" where he deploy his ML workload only while all resources provisioned on demand in run-time, results collected, transferred back the user and then all resources automatically dismantled. New evolving solutions, like Ray [40] are cluster approach to provide user a "serverless" experience over Ray cluster which is designed to scale ML workloads.

## 4 Final design and Implementation of the Serverless Compute Engine for Big Data

We now describe the Lithops [2] framework and how it being used for the High Performance Serverless Compute Engine (HPSCE), Operations Support and Big Data Serverless Execution Engine. Lithops is the main part of the Serverless Compute Engine for Big Data that is developed in WP3 that also contains SLA and Monitoring. We start by describing the Lithops module. As you will see, not only Lithops is a main module in the HPSCE and all other tasks of WP3, but also serves as a "glue" to tie various components and modules that were developed during the course of CloudButton project.

### 4.1 Lithops general architecture and design

Lithops is a multicloud framework that enables the transparent execution of unmodified, regular Python code against disaggregated cloud resources. With the Lithops, there is no new API to learn. It provides the same API as Python's standard multiprocessing [41] and concurrent.futures [42] and PyWren-IBM [43] libraries. Any program built on top of these libraries can be run on any of the major serverless computing offerings in today's market. This minimizes the learning curve for knowledgeable Python developers, keeps interfaces simple and consistent, and provides access transparency to disaggregated storage and memory in the cloud. Further, its multicloud-agnostic architecture ensures portability and overcomes vendor lock-in. Altogether, this represents a significant step forward in the programmability of the cloud. Lithops enables transparent access for users to virtually unbounded multicloud resources as nothing more than writing a program with a familiar language.

The initial prototype of Lithops was designed to work with IBM Cloud Functions [11] FaaS platform to run MapReduce tasks as serverless functions, compositions of functions, etc., and IBM Cloud Object Storage (IBM COS) to store all internal data required to make cloudbutton working. IBM Cloud Functions, which is based on Apache OpenWhisk, is a public cloud FaaS platform for running functions in response to events while IBM COS is IBM's public cloud offering for unstructured data storage service designed for durability, resiliency and security.

Currently, Lithops is completely refactored, and it now integrates what we called **Compute** and **Storage** abstractions. These two abstractions allow to integrate in our architecture any compute and storage service beyond IBM Cloud Functions, hiding its underlying vendor-specific implementations with common high-level methods. Thus, Lithops can be now identified as extensible and multicloud. For example, as of today, it already supports all the compute and storage backends listed in Table 1.

Cloud	Compute backend	Storage backend
IBM	IBM Cloud Functions	IBM COS
Amazon	AWS Lambda AWS Fargate	AWS S3
Google	Cloud Functions Cloud Run	Google Storage
Microsoft	Azure Functions	Azure Blob Storage
Alibaba	Function Compute	Alibaba OSS
Generic	Knative, KEDA	Swift, Ceph, Redis

Table 1: Lithops backends

The high-level architecture is depicted in Figure 4. Internally, the Lithops engine exploits the Python's dynamism to transparently capture the user's function and dependencies, package them, and upload them to the cloud. It is worth to note that the user's functions are not directly deployed

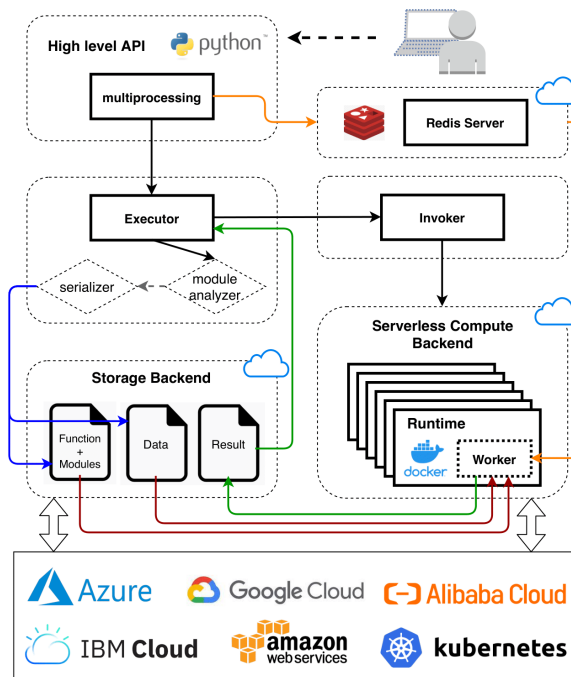


Figure 2: High level representation of Lithops

in the serverless compute backend. Instead, they are stored in the storage backend. Then, Lithops deploys a generic function called **Agent** responsible to lookup the packaged code and dependencies and run them. The usage of a function's Agent removes the overhead for function registration, favors the reuse of the single registered function in order to mitigate cold starts, and allows to run user-defined functions. At the same time, it eliminates the majority of hindering barriers about deployment, packaging and task execution that inhibit most users from painlessly entering the cloud.

In our effort to create a transparent and extensible framework, Lithops is built following a plugin-oriented model. To do so, we created the compute and storage abstractions in our architecture. These abstractions hide the underlying complexities of each compute and storage service, at the same time that they allow any new serverless compute and storage service to be easily integrable in our framework at later stage.

Another key component of the architecture is the **runtime**. The runtime is the place where the functions are executed. It can take different forms depending of the Serverless compute backend, for example, from Docker containers to python virtual environments packaged into a zip file. In any case, it contains the Cloudbutton Agent as the main entry point. Thus, the runtime with the Agent is deployed as a single generic function in the serverless compute backend. In this way, during the execution of a multiprocessing application, the Cloudbutton engine orchestrates the serverless compute backend to invoke, at large scale, the necessary Agent functions, each one representing one parallel process. Then, each function receives a json payload indicating where the user function's code and dependencies are stored, the data to be processed, and the place to store the final results.

Lithops offers much more flexibility in contrast of other tools. It allows to configure function memory in runtime, configure the desired number of workers for each application, and much more. Moreover, it includes a fault-tolerant mechanism that allows to run a job until completion, even if a function failed the execution of a single task. To summarize, Lithops can be viewed as a multicloud, large scale, executor engine, capable of running any local multiprocessing-based Python code in any Cloud. It is currently open-sourced in github [44].

One core principle behind CloudButton is programming simplicity. For this reason, we have devoted extra efforts to integrate Lithops with other tools (e.g., Python notebooks such as Jupyter), which are very popular environments for the scientific community. Python notebooks are interactive computational environments, in which one can combine code execution, rich text, mathematics, plots



and rich media.

## 4.2 No vendor lock-in and multi cloud portability

Vendor lock-in is usually one of the major concerns preventing wide adoption of the cloud and in particular of the serverless platforms. Up today, numerous serverless platforms offers different APIs, semantics and provides different user experience. With such diversity, it makes very complicated for the developer to move application from one serverless platform to another. As example, if user adapted his software to deploy workloads to the IBM Cloud Functions and now need to deploy same workload to the Kubernetes cluster, it will require him to learn K8s API and semantics and write dozens of the boiler plate code to support additional K8s as a compute backend. To address the vendor lock-in issue and to easy the user experience of using various cloud providers, we have designed Lithops to expose standard Python APIs while internally Lithops supports variety of serverless platforms. Figure 3 shows how same Lithops API supports both Amazon Lambda and IBM Cloud Functions

```
from lithops import FunctionExecutor

def double(i):
    return i * 2

fexec_aws = FunctionExecutor(backend='aws_lambda')
futures_aws = fexec_aws.map(double, [1, 2, 3, 4])

fexec_ibm = FunctionExecutor(backend='ibm_cf')
futures_ibm = fexec_ibm.map(double, [5, 6, 7, 8])

print(fexec_ibm.get_result(futures_aws+futures_ibm))
```

Figure 3: Multi-cloud experience with Lithops

### 4.2.1 Multiple APIs

At high-level, Lithops is shipped with two different Compute APIs that allow to interact with the underlying compute platforms, abstracting away the complexities of managing and using these services. These APIs are the Futures API and Multiprocessing API which also standard APIs that are popular among Python applications. In addition, Lithops provides a Storage API that allows to interact with a storage backend for managing data (upload/download/move/delete) in a simple way.

### 4.2.2 Futures API

Lithops implements a similar API to the built-in python concurrent.futures library.

This API is based on objects called Futures, created when Lithops spawns a function. With this Future object, it is possible to access the results and some statistics about the execution. For example, we can use the `call_async()` API method to spawn only one function and get the result by managing the generated Future 6

### 4.2.3 Multiprocessing API

Lithops implements most of the built-in python multiprocessing library methods and abstractions to run tasks in the Cloud with a familiar API. For example, we can create a `pool()` and use the `map()` method to spawn one function for each entry in an iterable list. Lithops now supports most of the

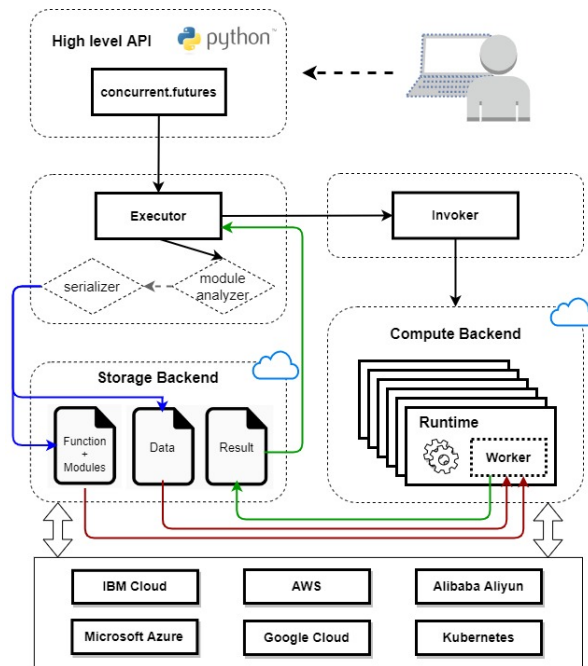


Figure 4: Futures API of Lithops

```
from lithops import FunctionExecutor

def hello(name):
    return 'Hello {}'.format(name)

with FunctionExecutor() as fexec:
    future = fexec.call_async(hello, 'World')
    print(future.result())
```

Figure 5: Future API of Lithops

Python multiprocessing abstractions, such as the Process, Pool, Queue, Pipe, Lock, Semaphore, Event, Barrier, and also remote memory in Manager objects.

```
from lithops.multiprocessing import Pool

def double(i):
    return i * 2

with Pool() as pool:
    result = pool.map(double, [1, 2, 3, 4, 5])
    print(result)
```

Figure 6: Multiprocessing API of Lithops

#### 4.2.4 Storage API

The storage API makes it easy to operate the storage backend with simple API methods similar to the python boto3 library. The Storage API in conjunction with a compute API provides unprecedented flexibility to run tasks to the Cloud like big data analytics or other kind of applications that involves data analysis or management. For example, we can use the storage API to upload a file from our computer to a cloud storage service, and then get this file from a function spawned with the *call\_async()* as shown in Figure 7

```
from lithops import FunctionExecutor, Storage

BUCKET = 'my-bucket'

def get_file(key, storage):
    return storage.get_object(bucket=BUCKET,
                              key=key))

if __name__ == "__main__":
    storage = Storage()
    storage.put_object(bucket=BUCKET,
                      key='test.txt',
                      body='Hello World')

    with FunctionExecutor() as fexec:
        fut = fexec.call_async(get_file, 'test.txt')
        print(fut.result())
```

Figure 7: Storage API of Lithops

#### 4.2.5 Storage OS API

Lithops provides a transparent way to interact with the storage backend. The module *lithops.storage.cloud\_proxy* mimics the os and the built-in function open to access Cloud Storage as if it were a local file system. By default, the configuration is loaded from the lithops config file, which makes it very easy to access cloud storage via OS API (See 8

```
from lithops.storage.cloud_proxy import os, open

with open('dir/file.txt', 'r') as f:
    content = f.read()
```

Figure 8: Storage OS API of Lithops

### 4.3 Multiple Storage backends and Big Data processing

Big data processing is crucial for many workloads, like data pre-processing for ML workloads, ETL, Monte-Carlo simulations and so on. To enable user application to easily access data while working with Lithops framework we have designed Lithops to expose a single Storage API that can be easily configured to use most popular storage solutions. Lithops currently supports major storage platforms, like

- IBM Cloud Object Storage

- Amazon S3
- Google Cloud Storage
- Azure Blob Storage
- Aliyn Object Storage Service
- Infinispan
- Ceph
- MinIO
- Redis
- OpenStack Swift

If developer need to develop additional storage connector, he can simply write it and plug-it into Lithops framework. All is required to implement Storage API interface of Lithops.

#### 4.3.1 Data discovery and data partitioner

To enable Lithops and effective way to access Big Data, we have implemented an advanced data Partitioner, which supports the following input data types

- Arrays of any type, e.g., numbers, lists of URLs, nested arrays, etc. A default partitioning logic is to partition the work allocating each entry in the array as input to a separate serverless function. As a consequence there will number of serverless tasks as the length of the input array, where each invocation process a single entry from the input array.
- A list of the data objects;
- A list of URLs, each pointing to a data object;
- A list of object storage bucket names;
- A bucket name with a prefix to filter out only the relevant objects from the bucket.

*Data discovery* is process that is automatically started when the bucket names are specified as an input. The data discovery process consists of a HEAD request over each bucket to obtain the necessary information to create the required data for the execution. The naive approach is to partition a bucket at the granularity of the data objects, resulting in the mapping of a single data object per serverless function.

This approach may lead to suboptimal performance and resource depletion. Consider an object storage bucket with two CSV files, one being 10GB and another one being 10MB in size. If CloudButton would partition the bucket by objects, this will lead to one serverless action processing 10GB and another one processing 10MB, which is obviously not optimized and may lead to running out of resource available to the invocation.

To overcome this problem, our Partitioner also is designed to partition data objects by sharding them into smaller chunks. Thus, if the chunk size is 64MB, then Partitioner will generate the number of partitions which is equal to the object size divided by the chunk size.

Upon completing the data discovery process, Partitioner assigns each partition to a function executor, which applies the map function to the data partition, and finally writes the output to the IBM COS service. Partitioner then executes the reduce function. The reduce function will wait for all the partial results before processing them.

It's not a trivial to partition a CSV file by byte range, since most likely it will split a single line. For example, consider a CSV file of 65MB that contains 10000 rows. If we use 64MB chunk size then most

likely the line at 64MB will be divided between two chunks, causing first chunk to have first part of the line and second chunk the last part of the line. Thus counting number of lines in both chunks will usually lead to the wrong number of 10,001 lines. To address this problem, we implemented an algorithm that prevents a single line to be split into 2 during partition phases.

#### 4.4 Serverless without limits

We designed Lithops to provide serverless user experience over various execution backends. Not only Lithops supports major FaaS backends, but we also implemented support for standalone backends, like virtual machines. By using Standalone backends Lithops enables to deploy workloads that not fits into FaaS paradigm. This could be long running job that requires large amount of GPUs or CPUs that executes some legacy code. Lithops is shipped with 3 different modes of execution. The execution mode allows you to decide where and how the functions are executed. Different execution modes of Lithops allow developer to write any Python code, execute it on his laptop or let Lithops to scale it against any serverless backend by a single configuration change

##### 4.4.1 Localhost execution mode

This mode allows you to run functions in your local machine, by using processes. This is the default mode of execution if no configuration is provided. It is mainly designed for testing purposes as it does not need any Cloud to be configured to make it running. By default, the local executor uses a local storage interface, which is the faster option. However, it can also use any public storage backend such as the IBM Cloud Object Storage service.

##### 4.4.2 Serverless execution mode

This mode allows you to run functions by using publicly accessible Serverless compute services, such as IBM Cloud Functions, Amazon Lambda or Google Cloud Functions, among others. In this mode of execution, each function invocation equals to a parallel task running in the cloud in an isolated environment. Lithops supports most FaaS platforms as follows

- **IBM Cloud Code Engine** allows to run applications, job or container on a managed serverless platform. Auto-scale workloads and only pay for the resources you consume. IBM Code Engine exposes both Knative and Kubernetes Job Descriptor API. Lithops supports both of them.
- **Kubernetes Jobs** allows to submit Lithops workloads as a Kubernetes jobs by hiding all complexity of K8s Job descriptor API
- **AWS Lambda** is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers. ([45])
- **AWS Batch** enables developers, scientists, and engineers to easily and efficiently run hundreds of thousands of batch computing jobs on AWS. AWS Batch dynamically provisions the optimal quantity and type of compute resources (e.g., CPU or memory optimized instances) based on the volume and specific resource requirements of the batch jobs submitted. ([46])
- **Google Cloud Functions** Run your code in the cloud with no servers or containers to manage. Cloud Functions is a scalable, pay-as-you-go functions as a service (FaaS) product to help you build and connect event driven services with simple, single purpose code.([47])
- **Google Cloud Run** Develop and deploy highly scalable containerized applications using your favorite language (Go, Python, Java, Node.js, .NET, and more) on a fully managed serverless platform. ([48])
- **Azure Functions** Azure Functions is a cloud service available on-demand that provides all the continually updated infrastructure and resources needed to run your applications. You focus on the pieces of code that matter most to you, and Functions handles the rest. Functions provides serverless compute for Azure. You can use Functions to build web APIs, respond to database changes, process IoT streams, manage message queues, and more. ([49])

- **Apache OpenWhisk executor** is an open source and serverless cloud platform that performs functions in response to events. The platform uses a function as a service (FaaS) model to manage infrastructure and servers for cloud-based applications and servers. OpenWhisk removes concerns about management of infrastructure and scaling by using Docker containers. In Lithops, the Apache OpenWhisk executor allows to execute functions in any vanilla OpenWhisk installation by communicating with its endpoint API. At the same time, the executor provides all the necessary methods to build custom runtimes, abstracting and hiding its complexities to the users.
- **IBM Cloud Function executor** is an extension of the Apache OpenWhisk executor and allow to execute functions in the IBM Cloud Functions service by using IBM-specific authentication methods.
- **Knative executor** is a serverless framework that is based on Kubernetes. One important goal of Knative is to establish a cloud-native and cross-platform orchestration standard. Knative implements this serverless standard through integrating the creation of container or function, workload management and auto scaling, and event models. In Lithops, the Knative executor allows to run functions in any knative deployment. It also allows to build and deploy docker runtimes to execute the functions.

#### 4.4.3 Standalone execution mode

This mode allows to run functions by using one or multiple Virtual machines (VM), either in a private cluster or in the cloud. In each VM, functions run using parallel processes like in the Localhost mode. Lithops can directly execute code in the operational system of VM or execute code inside Docker image in the VM. Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

Lithops allows to run functions using processes within a single docker container. In this way, a user can create a specific runtime with all the requirements libraries and packages instead of having to install all of them in the local machine. As stated before, this container is portable and can be executed in any other machine. In this sense, The docker executor can be used to run functions both locally or in an other remote server/VM. Currently, the docker executor supports both knative and Openwhisk/IBM Cloud functions runtimes. this means that a user can re-use these runtimes to run function in the locally machine without having to install any package.

The benefit of Standalone mode, is that some workloads can't be easily paralyzed and requires longer execution time with large amount of resources, CPU and GPU. Thus Lithops can automatically start any number of VMs with many resources and deploy user workload to those VMs, monitor executions, collect results and dismantle VMs, once analytic job is finished. Lithops supports

- Remote Virtual Machine
- IBM Virtual Private Cloud
- AWS Elastic Compute Cloud (EC2)

#### 4.5 Serverless without constraints and hybrid workloads

For load-intensive or more complex workloads, there is still some unsatisfied demand that is not fully addressed by today's serverless offerings, such as being too constrained with respect to available CPU, memory and disk capacity. The foundational thought is that in established serverless compute

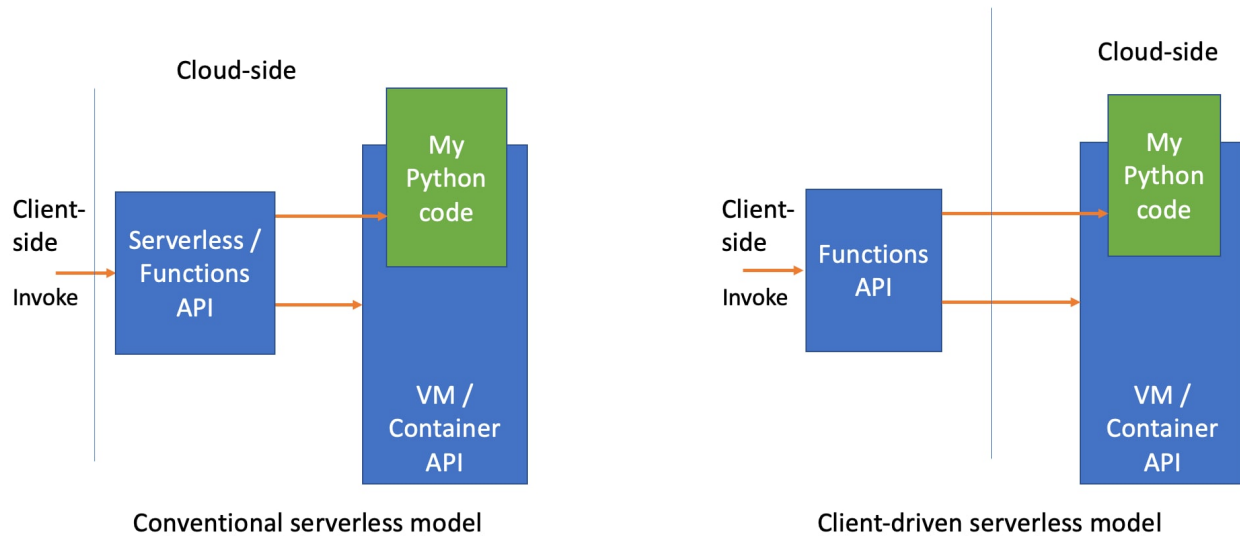


Figure 9: Client-driven serverless model

offerings, code gets isolated by running in a container, (micro-)VM, etc. As part of that, there is always a dedicated (but relatively limited) amount of CPU, memory, etc. available per invocation.

As shown in the figure 9, the user-facing serverless experience would remain the same as what is available today. However, the user would get the option to select the resources available for execution from the full spectrum of VM sizes available, not just a more constrained subset of CPU, mem, disk and network capacities. An additional benefit of this model is that it would be available at the price point of regular VMs while expose serverless user experience. Certain workloads (e.g., in the ML/AI or data processing space) can benefit significantly from running as much as possible on the same machine for low-latency, in-memory data sharing. To be concrete, this means up to 128 vCPUs, 1TB of memory, many TBs of local SSD disk, etc. are available per invocation and, potentially, hours of execution time if needed.

Also, the approach described in this post allows for the acquisition of infrastructure at the regular VM cost-level. While VMs are provisioned per invocation at the lowest level, the actual code still runs within one container on top of it. This means the underlying container base image can continue to be customized by the user by preinstalling libs and taking other configuration steps.

You'd like all infrastructure you're using to reside in your account, in your VPC, with support for security groups, public gateways, control over the subnets being used, the ability to use various storage options, large fast local SSDs and many 100's of GBs of memory.

We enabled this capability with this abstractions in the Lithops project. This extension to Lithops offers a very simple serverless map/reduce interface, where capacity management is entirely transparent to the user and where all features mentioned above are delivered.

In particular, Lithops supports the following modes:

- Custom images
- GPUs
- Auto dismantle mode
- Warm machines
- Serverless user experience
- LithopsCloud tool
- Master-Worker approach

Hybrid workloads are crucial for variety of use cases and scenarios. By hybrid we not necessarily mean multiple clouds, and it may also cover hybrid calculations between private clusters deployed on premise and public clouds or even hybrid calculations between multiple data centers of the same cloud provider. As example, consider a pharmacy company that has a very sensitive data that need to be processed on premise. But once data became less sensitive, it can be further processed in the public cloud, to benefit from large scale and resources of the cloud provider. We developed Lithops to support all kind of hybrid workloads, between public clouds, private and public clouds and between various data centers of the same cloud provider.

#### 4.6 LithopsCloud CLI tool

To easy adoption of CloudButton tools we have developed a LithopsCloud [50] which is a novel command line interface (CLI) tool that enables iterative way to setup Lithops for the new users. Without this tool new users need to follow Lithops documentation and setup Lithops configuration file for specific backend and storage they wish to use. To easy this process, new users can use LithopsCloud CLI tool and they only required to provide their cloud *apikey*. Then LithopsCloud will use iterative way to create Lithops configuration file that is configured for the compute and storage backends. LithopsCloud supports IBM Cloud Functions, IBM Cloud Code Engine, IBM Gen2 and IBM Cloud Object Storage. The modular design of LithopsCloud tool allows developers to easily add additional backends

#### 4.7 Temporary data

Temporary data is a special kind of data that only required between different stages during the execution process while at certain point this data need to be cleaned. The cleaning process may be automatically invoked or can manually triggered manually by user actions. In CloudButton we address two types of temporary data

**System generated** is type of data that generated internally by Lithops. The process usually not visible to the end user and he is not aware of this type of data. As example Lithops generates a single JSON file 10 per each invocation. This file includes invocation status, completion timestamp, and other metadata that is needed by Lithops. Once all invocation completed, Lithops reads all generated files, obtain statuses of each invocation and decides what response return to the user.

```
{"exception": false, "host_submit_tstamp": 1592715321.0335932, "start_tstamp": 1592715331.4192479, "python_version": "3.6.9", "call_id": "00000", "job_id": "M000", "executor_id": "0d76ac/0", "activation_id": "0b566331cc2100000", "type": "__end__", "function_download_time": 0.01882219, "data_download_time": 0.10679126, "function_start_tstamp": 1592715332.0058982, "function_end_tstamp": 1592715338.2056653, "function_exec_time": 6.19976711, "result": true, "output_upload_time": 0.02636218, "end_tstamp": 1592715338.2378411}
```

Figure 10: Status JSON file

**User generated** This type of temporary data is usually implicitly generated by the user. As example, analytic job may consists of various stages, where output of one stage is used as in input to the consequence stage. In this scenario, application needs implicitly persist intermediate data between stages. It's then user responsibility to invoke cleanup of the temporal data. To enable the capability we designed and implemented CloudObject which is special type of data that can be used to share data between stages.

#### 4.8 CloudObject to share results and maintain state

We noticed that data sharing in variety of use cases is essential because each serverless job is memory limited and also run in an isolated environment. Dealing this challenge naively would be manually define unique storage keys for each object, but we found that this approach can be confusing and even quite messy when applying parallel jobs - it requires to configure keys prefixes for each task and also



developing dedicated indexing logic for each one. Trying to make the code clearer and cleaner, we defined a generic user-friendly class called “CloudObject” which contains all required details to reach the storage object that it represents. For example, when processing several serverless jobs in parallel, we can simply store CloudObjects as follows: By this approach, the CloudObject can be considered as

```
def my_function(data, storage):  
    processed_data = ...  
    cloud_object = storage.put_cobject(processed_data)  
    return cloud_object
```

And we can also load CloudObjects by:

```
def my_function(cloud_object, storage):  
    data = storage.get_cobject(cloud_object)  
    ...
```

Figure 11: CloudObject

a pointer to the stored data, without the necessity of managing its storage path exactly, and Lithops can encapsulate code that manages the storage service instance. We use CloudObjects as part of an external storage system that was developed to store intermediate pipeline results and load each result when needed. By this approach, the user can automatically load these intermediate results if they have already been calculated before and avoid recomputation. After finishing all pipeline analytics, the user can clean its cache easily.

CloudObjects also used to share results between multi-stages jobs and also enables to persist state to support statefull invocations.

#### 4.9 Cost effective serverless model for Big Data analytics

Serverless platforms are often provides very attractive cost model, where consumers pay for the actual resources being consumed. This is different to previous cost models, where billing is done per virtual machines, no matter how much it consumed. Billing in serverless platforms usually calculated per seconds and depends on the memory consumed, CPU requested and execution time. While this is very attractive, improper usage may trigger very high costs. There are various reasons that may trigger high costs

- Requesting too much resources. For example, user need to extract colors from binary images. He submitted as number of invocations as number of images and requested 4GB per invocation. While there is nothing wrong from technical point of view, it might be that 4GB is too much and user could use 2GB of memory, reducing costs by 2.
- Stalled invocations. This may happen because invocations access 3rd party resources that are slow in responding, bug in the user code, etc. If invocation get stuck, user will pay for it until it halted. This obvious may trigger high costs for the straggler invocations

To address the issues above, we implemented different approaches in Lithops which are tuned for standalone and serverless backends. For standalone backends, Lithops has automatic dismantle counter that is running inside standalone backend. If there is no activity, than after configurable amount of time, the virtual machine will be auto dismantled. This prevents use cases of stalled invocations for standalone backends. We have also implemented mechanism that will shut down standalone backend within specific time limit, even if there is activity. This prevents situations where user code entered endless loop and keep standalone baclend active all the time. For FaaS backends

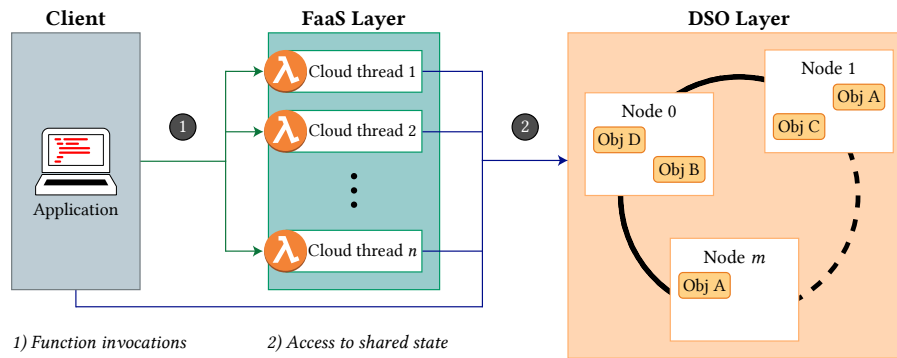


Figure 12: CRUCIAL's overall architecture. A client application runs a set of cloud threads in the FaaS layer. The cloud threads and the client have access to the shared state stored in the DSO layer.

Lithops can submit "batches" of invocations, where each batch uses different memory and CPU footprint. Thus for example if user's code uses Lithops to process  $N$  files of different sizes, it possible to submit  $X$  invocations each requesting 2GB of memory,  $Y$  invocations each requesting 3GB of memory and the rest  $N-X-Y$  with 1GB of memory.

Lithops also supports cost effective model, by enabling multi-stages jobs, where some may be deployed over VMs and some over FaaS. As explained in D2.3 (EMBL use case) this approach has many benefit, reduce costs and is being used by METASPACE of EMBL in their production workloads.

## 5 Lithops in the broad scope of CloudButton

We now explain how Lithops is used in the broad scope of the CloudButton use cases and partners.

### 5.1 Crucial

CRUCIAL is a framework to program highly-parallel stateful serverless applications in Java. It is built upon the key insight that FaaS resembles to concurrent programming at the scale of a datacenter. Accordingly, a distributed shared memory layer is the natural answer to the needs for fine-grained state management and synchronization. The framework allows to port effortlessly a multi-threaded Java code base to serverless, where it can benefit from the scalability and pay-per-use model of FaaS platforms. CRUCIAL is fully detailed in Deliverable D4.3 [51]. Below, we briefly describe CRUCIAL then explain how it is usable within the context of Lithops. In particular, we focus on how to call the distributed shared object (DSO) layer of CRUCIAL from Lithops.

#### 5.1.1 Overview

Figure 12 presents the overall architecture of CRUCIAL. The framework encompasses three main components (from left to right in Figure 12): (1) the client application; (2) the FaaS computing layer that runs the serverless functions, aka., *cloud threads* in Crucial parlance; and (3) the DSO layer that stores the shared objects. A client application differs from a regular JVM process in two aspects: threads are executed as serverless functions, and they access shared data using the DSO layer. Moreover, CRUCIAL applications may also rely on external cloud services, such as object storage to fetch input data (not modeled in Figure 12).

Each object in the DSO layer is uniquely identified by a reference. Fine-grained updates to the shared state are implemented as methods of these objects. Given an object of type  $T$ , the reference to this object is  $(T, k)$ , where  $k$  is the name of the annotated object field. When a cloud thread accesses an object, it uses its reference to invoke remotely the appropriate method.

CRUCIAL constructs the DSO layer using consistent hashing [52], similarly to Cassandra [53]. Each storage node knows the full storage layer membership and thus the mapping from data to node. The location of a shared object  $o$  is determined by hashing the reference  $(T, k)$  of  $o$ . This offers the following usual benefits: (1) no broadcast is necessary to locate an object; (2) disjoint-access parallelism can be exploited; and (3) service interruption is minimal in the event of server addition and removal.

```
1 dso=os.environ.get('DSO')
2
3 def my_function(x):
4     client = Client(dso)
5     d = client.getAtomicCounter("cnt")
6     return d.increment(x)
7
8 if __name__ == '__main__':
9     fexec = lithops.FunctionExecutor(runtime='0track/lithops-dso:1.1')
10    fexec.call_async(my_function, 3)
11    client = Client(dso)
12    c = client.getAtomicCounter("cnt")
13    print("counter: "+str(c.tally()))
14    print(fexec.get_result())
15    print("counter: "+str(c.tally()))
```

Listing 1: Using CRUCIAL from Lithops.

Stateful applications (e.g., the training phase of a ML algorithm) often aggregate and combine small granules of data. Unfortunately, serverless functions are not network-addressable and run separate from data. As a consequence, these applications are routinely left with no other choice but to “ship data to code”. This is known as one of the biggest downsides of FaaS platforms [54].

To illustrate this point, consider an AllReduce operation where  $N$  cloud functions need to aggregate their results by applying some commutative and associative operator  $f$  (e.g., a sum). To achieve this, each function first writes its local result in the storage layer. Then, the functions await that their peers do the same, fetch the  $N$  results, and apply  $f$  sequentially. This algorithm is expensive and entails a communication cost of  $N^2$  messages with the storage layer.

CRUCIAL fully resolves this anti-pattern with minimal efforts from the programmer. Complex computations are implemented as object methods in DSO and called by the cloud functions where appropriate. Going back to the above example, each function simply calls  $f(r)$  on the shared object, where  $r$  is its local result. With this approach, communication complexity is reduced to  $\mathcal{O}(N)$  messages with the storage layer. We exploit this key feature of CRUCIAL in our serverless implementation of several ML algorithms (e.g.,  $k$ -means, linear regression, random forest). Its performance benefits are detailed in D43 [51].

### 5.1.2 Integration with Lithops

CRUCIAL is callable from Lithops serverless functions. In particular, Lithops may invoke the distributed shared object (DSO) layer. This provides access to all of the coordination and mutable shared abstractions in CRUCIAL from a Lithops context. In what follows, we detail how this integration is made and present a base example.

CRUCIAL is callable from Python thanks to a JPytype connector [55]. This connector allows to call any CRUCIAL objects from a Python program. Under the hood, JPytype runs a JVM separately from the Python VM and communicate with it using shared memory. The connector covers a few hundreds lines of code. It is part of the CRUCIAL framework and as such open sourced under an Apache License [56].

Listing 1 gives an integration example of Lithops with Crucial. In this figure, a shared counter named `cnt` is created by the client application (line 12). Function `my_function` retrieves a reference to this counter (line 5) and returns its value after an increment (line 6). Notice that the client application invokes 3 instances of `my_function`. As a consequence, `cnt` is incremented 3 three times. Since `cnt` is linearizable, the response values of the functions is exactly  $\{1, 2, 3\}$ . As a consequence, the client application outputs 3 on the console as its final result (line 15).

<pre> 1 int myFun(int x) { 2     return x + 2; 3 } </pre>	<pre> 1 (module 2   (table 0 anyfunc) 3   (memory \$0 1) 4   (export 'memory' (memory \$0)) 5   (export 'myFun' (func \$myFun)) 6   (func \$myFun (; 0 ;) \ 7     (param \$0 i32) (result i32) 8     (i32.add 9       (get_local \$0) 10      (i32.const 2) 11    ) 12  ) 13 ) </pre>
---	---

Figure 13: A C source function and its representation in WASM text format, WAST.

## 5.2 WebAssembly

WASM [57] is a portable binary format that is the successor to Native Client [58] and asm.js [59]. WASM was originally created to provide software-fault-isolation (SFI) when executing untrusted code in a browser. Since then, its potential as a general-purpose SFI mechanism has been exploited in serverless [60], IoT [61], embedded devices [62], edge computing [63], and as a replacement for container-based isolation in Kubernetes [64].

WASM offers strong memory safety guarantees by constraining memory access to a single linear byte array, referenced with offsets from zero. This enables efficient bounds checking at both compile time and runtime, with runtime checks backed by traps. These traps, and others for referencing invalid functions, are implemented as part of WASM runtimes [65]. The security guarantees of WASM are well established in existing literature, which covers formal verification [66], taint tracking [67], and dynamic analysis [68]. WASM offers mature support for languages with an LLVM front-end such as C, C++, C#, Go and Rust [69], while toolchains exist for Typescript [70] and Swift [71]. Java bytecode can also be transpiled [72], and further language support is possible by compiling language runtimes to WASM, *e.g.* Python, JavaScript and Ruby. WASM is under active development, with an extensive roadmap [73] including 64-bit WASM (currently WASM is limited to a 32-bit address space), shared memory and modules.

### 5.2.1 Format

Figure 13 shows two listings, one of a simple C function, and the other of the text representation of its corresponding WASM, using the WebAssembly Text Representation (WAST) [65]. WAST is a human-readable format that supports two-way translation with WASM binaries via the WebAssembly Binary Toolkit [74].

In the WAST listing we see a top-level `module`, which forms the basis of all WASM binaries. Interaction between WASM modules is not yet officially supported, although multi-module WASM is under development [73], and there is also partial support for dynamic linking [75]. Each WASM module defines a function table, which is used to support indirect function calls via the `call_indirect` instruction. Indirect calls are used to support function pointers in source languages that use them, and when dynamically linking two WASM modules. The `memory` keyword shows the definition of the WASM linear memory for this module, which can also include limits on the size of the memory. If no limit is specified, this limit will be 4 GB, corresponding to the maximum value that can be represented using a 32-bit integer. The `export` command specifies which parts of this module are accessible to the WASM runtime and other WASM modules.

WASM uses a stack machine, with each function defining a set of local variables that can be moved onto and off the stack and manipulated in place. WASM functions can also load and manipulate values held in the global linear memory array using integer offsets. WASM enforces a structured control flow, so does not support arbitrary jumps, hence cannot compile certain language features,

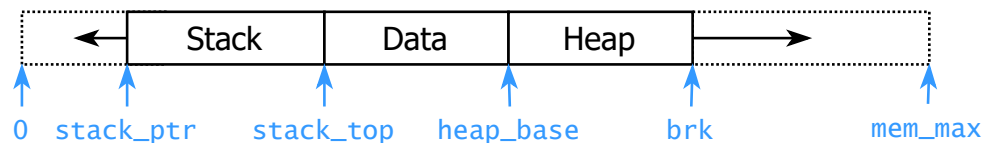


Figure 14: WASM linear memory layout. All addresses are expressed as offsets from zero. Stack memory grows downwards towards zero, while heap memory grows upwards towards its maximum offset at `mem_max`.

such as `goto` statements or virtual method tables in C/C++ [76].

### 5.2.2 Linear memory

Memory in WASM is handled via zero-indexed linear byte arrays. All memory accesses are expressed as offsets from zero, so can be bounds checked by checking they are greater than zero, and less than the maximum memory size. WASM code has access to several memory-related operations including `grow-memory` and `shrink-memory`, but cannot allocate disjoint address ranges. The WASM runtime is responsible for maintaining the memory that backs this linear address space and converting the offsets into pointers to it. The runtime must halt execution if an invalid memory access is made. The conformance of a runtime to these rules can be tested using the WASM specification tests [65].

Figure 14 shows the layout of the WASM linear memory, which contains three regions, a stack, static data, and a heap. The WASM stack grows downwards towards zero, hence has a fixed maximum size, which is specifiable by the WASM module using the `stack_top` value. The static data associated with the module, such as string literals and constants, is held in the data region, which does not change size and is created automatically by the runtime. The heap grows upwards towards the maximum memory size, which can be set by the module, or left to the default maximum which is roughly four gigabytes in 32-bit WASM.

WASM linear memory is most commonly implemented using a linear byte array placed at the bottom of a 4 GB virtual memory region. 4 GB is the largest possible linear memory array for a 32-bit WASM module, and by pre-allocating, the runtime can rely on the OS's memory manager to cause a page fault if the application makes memory accesses outside of this region. This avoids the runtime needing to add its own bounds checks on each memory access, which would otherwise severely affect performance. A standard x86 machine can support a virtual memory address space of  $2^{48}$  bytes, so it has room for more than four billion such mappings. However, this technique will not be possible with 64-bit WASM, whose linear address space will exceed the available virtual memory address space of the machine.

### 5.2.3 Toolchains and runtimes

Although any compiler toolchain may choose to implement a back-end for WASM, the most developed and widely used is that offered by LLVM [77]. It provides front-ends for a range of languages including C and C++ through Clang [78], Rust [79], and Go [80]. To execute an application compiled to WASM, all library dependencies must also be compiled to WASM. Standard libraries for all supported languages such as `libc` and `libc++` are provided by open-source projects [81].

There are now many WASM runtimes, each focusing on different execution environments, performance goals, and breadth of features. The most significant difference between WASM runtimes is whether they interpret the WASM at runtime, or compile it ahead of time (AOT) into machine code. AOT runtimes offer better performance as the code generation step can introduce architecture-specific optimisations, but require running this code generation once for each target architecture. Interpreters are slower, but not platform-specific, so are more suitable for execution environments where the architecture is not known ahead of time, such as web browsers. The most popular AOT WASM runtimes are currently: WAVM [82], a general-purpose WASM runtime written in C/C++ with support for all the most recent WASM features; WAMR [83], also written in C/C++ but tar-

getting a smaller resource footprint, suitable for IoT, edge and trusted execution environments; and wasmtime [84], another general-purpose WASM runtime written in Rust with support for all current WASM features. The most popular interpreter runtimes are those implemented in the four major browsers; V8 in Chrome/Chromium [85], SpiderMonkey in Firefox [86], Chakra in Microsoft Edge [87] and JavascriptCore/SquirrelFish in Webkit [88].

#### 5.2.4 WASI: the WebAssembly system interface

WASM itself does not define a foreign function interface or set of supported syscalls. Such an API would be platform-specific, and hence at odds with the goals of the project to provide platform-independent isolation. Platform-specific APIs for interaction with the execution environment and underlying host will vary between WASM runtimes, but the Bytecode Alliance [89] has created the WebAssembly System Interface (WASI) [81], which standardises a range of POSIX-like system calls. WASI uses a capability-based security model [90] across a suite of POSIX-like system calls [81]. These calls include file I/O, networking, timing and error handling.

#### 5.2.5 Future WebAssembly development

WASM is still evolving, with an extensive roadmap [73]. It is currently limited to a 32-bit address space, but 64-bit WASM is under active development. The WASM specification does not yet include mechanisms for sharing memory; there is a proposal to add a form of synchronised shared memory to WASM [91], but it is not well suited to sharing serverless state dynamically due to the required compile-time knowledge of all shared regions. It also lacks an associated programming model and provides only local memory synchronisation.

### 5.3 C++ - Faasm

FAASM [60] is a serverless runtime that uses *Faaslets* to execute distributed stateful serverless applications across a cluster. *Faaslets* are a new isolation mechanism introduced in FAASM that uses WASM for inter-function isolation (together with additional Linux tooling). FAASM is designed to integrate with existing serverless orchestration platforms, which provide the underlying infrastructure, auto-scaling functionality and user-facing front-ends. FAASM handles the scheduling, execution and state management of *Faaslets*. The design of FAASM follows a distributed architecture: multiple FAASM runtime instances execute on a set of servers, and each instance manages a pool of *Faaslets*. Details on the FAASM runtime, *Faaslets*, and their evaluation are included in Deliverable 5.3 [92].

FAASM is implemented in C++20 and released as open-source code on Github [93]. FAASM is compiled using clang-13. FAASM applications and all transitive dependencies, *e.g.* *libc*, are compiled to WASM using clang-10 [78], as part of the Faasm CPP toolchain [94]. *Faaslets* support executing WASM code in two different WebAssembly runtimes: WAVM [82] and WAMR [83]. FAASM integrates with Knative v1.1 [95], a state-of-the-art serverless framework built using Kubernetes [96], and can therefore be seamlessly deployed on a Kubernetes cluster.

#### 5.3.1 Faasm integration with Lithops

FAASM can integrate with Lithops as a new *Compute Backend* of the serverless kind. Given that FAASM exposes all its endpoints via a REST API, it is straightforward to orchestrate a workflow with Lithops, and invoke the functions in Faasm. In particular, FAASM already implements a CLI tool, *faasmcli* to invoke functions in FAASM from the command line using Python; the integration with Lithops is then very similar to *faasmcli*. Figure 15 depicts the architecture in more detail. In particular, all the calls that Lithops expects compute backends to implement like *invoke*, *deploy*, or *clear*, are translated to HTTP requests that the FAASM runtime understands. The abstraction behind the REST API means that FAASM's deployment model is transparent to Lithops users. We observe that Lithops native Job object, contains a superset of all the information FAASM ever needs, requiring no changes to the user-facing API.

FAASM supports executing functions written in compiled languages like C or C++ (with experimental support for Rust), or interpreted languages like Python. Lithops executes functions written in Python. By integrating FAASM as a Lithops compute backend, we open the possibility of running

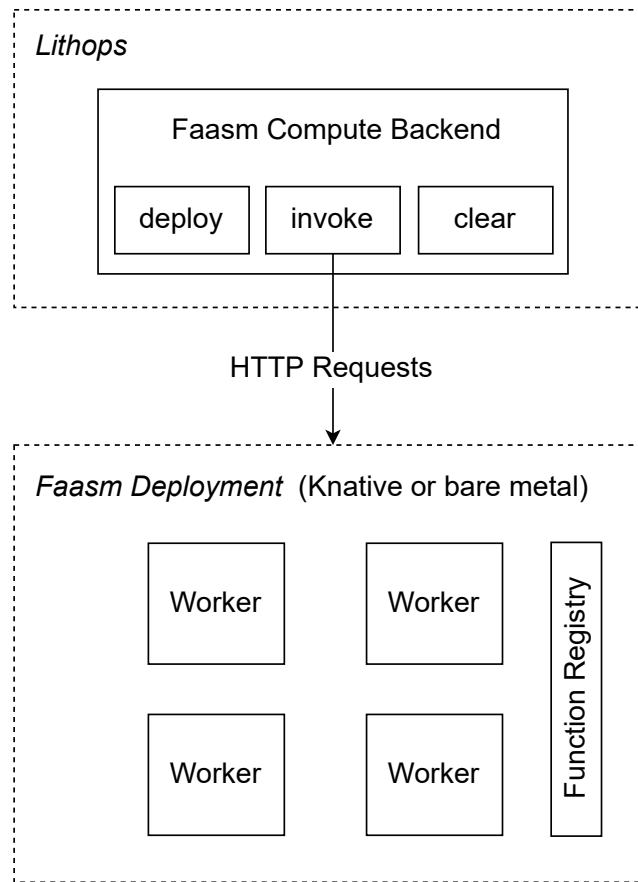


Figure 15: FAASM integration with Lithops as a serverless compute backend.

functions in compiled programming languages like C or C++ with Lithops, in addition to functions written in Python. To support such a model, we add a flag to the user-facing `call_async` Lithops method specifying that the function has an `external=True` definition. The integration is available on FAASM’s fork of Lithops [97].

Functions need to be registered in FAASM before requests for them can be served. Functions written in compiled programming languages are (i) cross-compiled to WebAssembly, (ii) uploaded to FAASM, and upon a succesful upload (iii) machine-code is generated to execute them. To execute functions written in Python, FAASM cross-compile the Python runtime [98]. To indicate the function that runs in the cross-compiled runtime, users must declare it with the signature `def faasm_main()` [99].

Figure 16 summarises the steps to invoke a function in Lithops with FAASM as compute backend. First, before functions are invoked, Lithops deploys the backend compute runtime. For FAASM, this means initialising the workers and the cross-compiled Python runtime. Then, when a function is invoked for the first time (*i.e.* a cold start), we either cross-compile it to WebAssembly using the FAASM toolchain for C/C++ functions [100], or transpile the pickled Python function to meet the required structure for Python functions [98]. After a succesful upload, the function can be invoked through an HTTP request to FAASM. Successive invocations of the same function do not need to repeat the uploading step. Lastly, given that FAASM and Lithops share the same asynchronous semantics, the runtime will await until execution has finished, and return the result.

### 5.3.2 Integration with RedHat and Infinispan

Red Hat Data Grid is an in-memory, distributed, elastic NoSQL key-value datastore. Data Grid is built from the Infinispan open-source software project and is available to deploy as an embedded library, as a standalone server, or as a containerized application on Red Hat OpenShift Container Platform. Based on the configuration, Infinispan may persist data in persistent storage or keep it in the



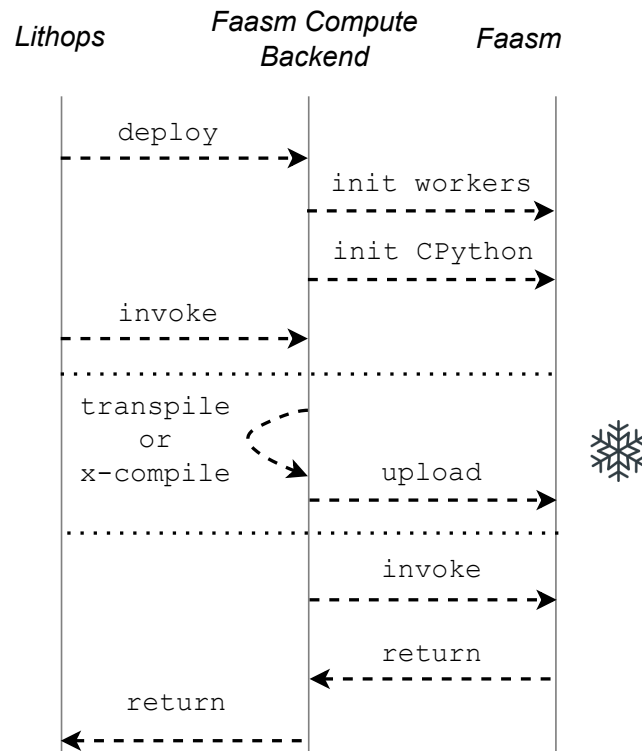


Figure 16: Diagram of the invocation of functions in FAASM through Lithops.

memory only. This makes Infinispan as a perfect candidate for storing temporary data. We designed Lithops so that all internal accesses to storage pass via internal-storage interface. This modular approach, allows us to use different storage connectors to access different storage backends. To extend Lithops to access additional storage backend, all what is required is to implement internal-storage interface and configure Lithops to use new storage. We implemented Infinispan storage connector prototype that internally implements Infinispan RESTfull API to enable access to the remote Infinispan cluster. A simple configuration is required to leverage Infinispan as internal storage of Lithops

```
#infinispan:
#username      : <USER_NAME>
#password      : <PASSWORD>
#endpoint      : <INFINISPAN_SERVER_URL:PORT>
#cache_manager : <CACHE_MANAGER> # Optional. 'default' in default value
```

This enabled to use Infinispan as in memory storage to store temporary metadata generated by Lithops and avoids to use cloud object storage to store temporary data. There are numerous benefits of this approach in particular reducing overall costs, achieving low latency to Inifnispan which greatly improves overall execution times. As next steps we plan to benchmark architecture of using Infinispan and better understand the cost efficiency of this approach.

## 6 Serverless Workflows

### 6.1 Triggerflow

In the context of the CloudButton project, we have created Triggerflow [101], a novel building block for composing event-based services.

Triggerflow aims to leverage existing event routing technology (Knative Eventing) to enable extensible trigger-based orchestration of serverless workflows. Triggerflow includes advanced abstractions not present in Knative Eventing like dynamic triggers, trigger interception, custom filters, ter-



mination events, and a shared context among others. Some of these novel services may be adopted in the future by event routing services to make it easier to compose, stream, and orchestrate tasks.

We can see in Figure 17 an overall diagram of the Triggerflow Architecture. The Trigger service follows an extensible Event-Condition-Action architecture. The service can receive events from different Event Sources in the Cloud (Kafka, RabbitMQ, Object Storage, timers). It can execute different types of Actions (containers, Functions, VMs). And it can also enable the creation of custom filters or Conditions from third-parties. The Trigger service also provides a shared persistent context repository providing durability and fault tolerance. Figure 17 also shows the basic API exposed by TriggerFlow: `createWorkflow` initializes the context for a given workflow, `addTrigger` creates a new trigger (including event, conditions, actions, and context), `addEventSource` permits the creation of new event sources, and `getState` obtains the current state associated to a given trigger or workflow.

Different applications and schedulers can benefit from serverless awakening and rich triggering by using this API to build different orchestration services like Airflow-like DAGs, ASF state machines or Workflow as Code clients like PyWren.

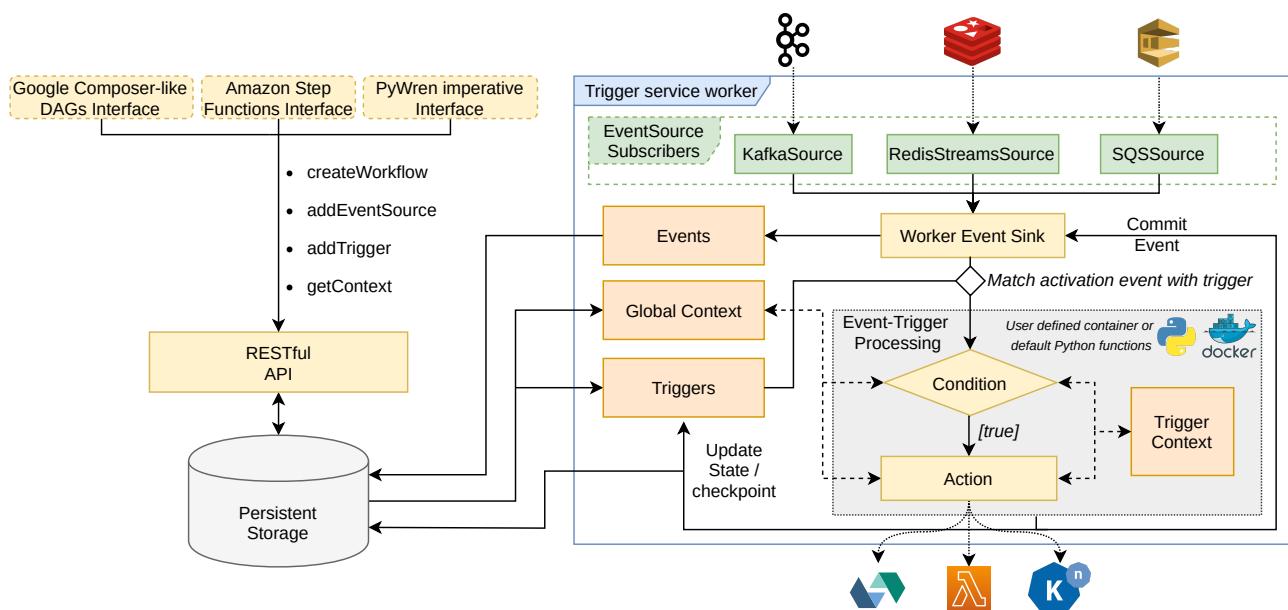


Figure 17: Triggerflow Architecture

This proposed architecture must support a number of design goals:

1. Support for Heterogeneous Workflows: The main idea is to build a generic building block for different types of schedulers. The system should support enterprise workflows based on Finite State Machines, Directed Acyclic Graphs, and Workflow as Code systems.
2. Extensibility and Computational Reflection: The system must be extensible enough to support the creation of novel workflow systems with special requirements like specialized scientific workflows. The system must support introspection and interception mechanisms enabling the monitoring and optimization of existing workflows.
3. Serverless design: The system must be reactive, and only execute logic in response to events, like state transitions. Serverless design also entails pay per use, flexible scaling, and dependability.
4. Performance: The system should support high-volume workloads like data analytics pipelines with numerous parallel tasks. The system should exhibit low overheads for both short-running and long-running workflows.

Our proposal is to design a purely event-driven and reactive architecture for workflow orchestration. Like previous works [25, 26, 27], we also propose to handle state transitions using event-based triggering mechanisms. The novelty of Triggerflow approach precisely relies on the aforementioned design goals: support for heterogeneous workflows, extensibility, serverless design, and performance for high volume workloads.

We follow an **Event Condition Action** architecture in which triggers (active rules) define which action must be launched in response to Events or to Conditions evaluated over one or more Events. The system must be extensible at all levels: Events, Conditions, and Actions.

We have developed two different implementations of Triggerflow: one over Knative, which follows a push-based mechanism to pass the events from the event source to the appropriate worker, and another one using Kubernetes Event-driven Autoscaling (KEDA), where the worker follows a pull-based mechanism to retrieve the events directly from the event source. We created the prototypes on top of the IBM Cloud infrastructure, leveraging the services in its catalog to deploy the different components of our architecture. These components are the following:

- A Front-end RESTful API, where a user connects to interact with Triggerflow.
- A Database, responsible for storing workflow information, such as triggers, context, etc.
- A Controller, responsible for creating the workflow workers in Kubernetes.
- The workflow workers (TF-Worker hereafter), responsible for processing the events by checking the triggers' conditions, and applying the actions.

In our implementation, each workflow has its own TF-Worker. In other words, the scalability of the system is provided at workflow-level and not at TF-Worker level. In our system, the events are logically grouped in what we call *workflows*. The *workflow* abstraction is useful, for example, to differentiate and isolate the events from multiple workflows, allowing to share a common context among the (related) events.

To demonstrate the flexibility that can be achieved using triggers with programmable conditions and actions, we have implemented three different workflow models that use Triggerflow as the underlying serverless and scalable workflow orchestrator: based on State Machines (Amazon Step Functions), Directed Acyclic Graphs (Airflow), and Workflow as Code (Lithops).

We showcase here the Workflow as Code use case. The trigger service is also useful to reactively invoke an external scheduler because of state changes caused by some condition. For example, Workflow as Code systems like Lithops or Azure Durable Functions represent state transitions as asynchronous function calls (`async/await`) inside code written in Python or C#. Asynchronous invocations and futures in Lithops or `async/await` calls in Azure Durable Functions simplify code so developers can write synchronous-like code that suspends and continues when events arrive.

The model supported by Azure Durable Functions is reactive and event-based, and it relies on event sourcing to restart the function to its current state. We can use dynamic triggers to support external schedulers like Durable Functions that suspend their execution until the next event arrives.

In Lithops API, the functions `call_async` and `map` are used to invoke one or many functions. Lithops code is executed normally in a notebook in the client, which is usually adequate for short running workflows. But what if we want to execute a long-running workflow with Lithops in a reactive way? The solution is to run this Lithops code in Triggerflow reacting to events. Here, prior to perform any invocation, Lithops can register the appropriate triggers, for example a function termination trigger in `call_async` function and an aggregate trigger for all functions in a `map` invocation.

After trigger registration for each function, the function can be invoked and the orchestrator function could decide to suspend itself. It will be later activated when the trigger fires.

To ensure that the Lithops code can be restarted and continue from the last point, we use *event sourcing*. When the orchestrator code is launched, an event sourcing action will re-run the code acquiring the results of functions from termination events. It will then be able to continue from the last point.

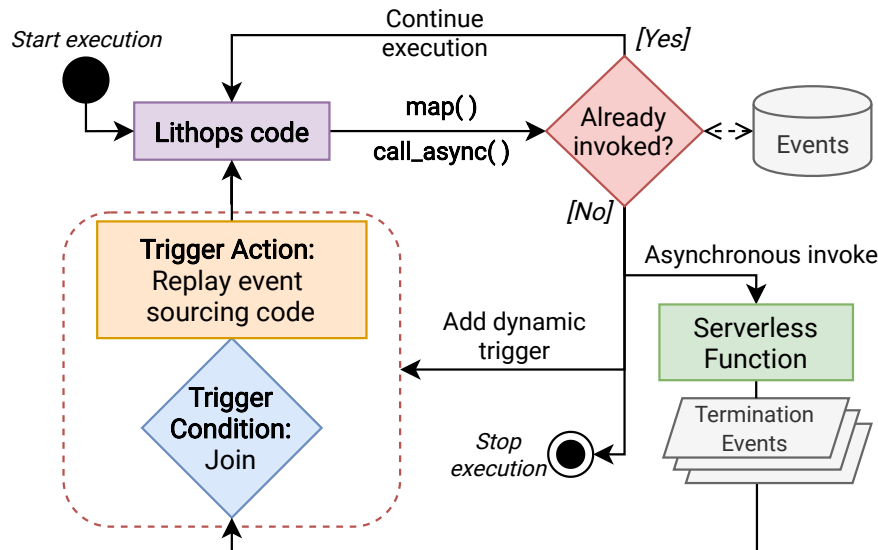


Figure 18: Life cycle of an event sourcing-enabled workflow as code with Lithops as external scheduler

In our system prototype, the event sourcing is implemented in two different ways: native and external scheduler.

In the *native scheduler*, the orchestration code is executed inside a Triggerflow Action. Our Triggerflow system enables then to upload the entire orchestration code as an action that interacts with triggers in the system. When Triggerflow detects events that match a trigger, it awakens the native action. This code then relies on event sourcing to catch up with the correct state before continuing the execution. In the native scheduler, the events can be retrieved efficiently from the context and thus accelerate the replay process. If no events are received in a period, the action will be scaled to zero. This guarantees reactive execution of event sourced code.

In the *external scheduler*, we use Lithops [2], where the orchestration code is run in an external system, like a Cloud Function. Then, thanks to our Triggerflow service, the function can stop its execution each time it invokes for example a `map()`, recovering their state (event sourcing) when it is awoken by our TF-Worker once all `map()` function activations finished their execution. Moreover, to use our event sourcing version of Lithops, it is not required any change in the user's code. This means that the code is completely portable between the local-machine and the Cloud, so users can decide where to run their Lithops workflows without requiring any modification. The life cycle of a workflow using an external scheduler can be seen in Figure 18.

## 6.2 Apache Airflow

Apache Airflow [7] is an open source platform that provides authoring, scheduling and monitoring of workflows represented as directed acyclic graphs (DAGs). Every node of the DAG represents a task, and the edges represent dependencies between tasks, and thus, the order of execution.

Apache Airflow's main objective was to be a scheduling platform for ETL workflows with a focus on the integration of different services from potentially different Cloud providers. In Airflow, DAG nodes are made up of two components: on the one hand we have the operator, which is the entity that describes how or where a task should be executed; on the other hand, the task defines the business logic of the job in question. This design enables Airflow to be extensible, since new operators can be implemented and embedded into Airflow in order to integrate different services.

In addition, Airflow allows to benefit from great workflow fault tolerance, logging, and observability features. One key concept of Airflow is its "configuration as code" paradigm used to compose DAGs as a Python script, which is very convenient to design, maintain and reuse complex workflows. Also, workflows can be triggered from many sources, like periodic cron triggers so that executions

are scheduled automatically. Finally, Airflow GUI makes it comfortable to schedule and monitor DAG runs.

The Airflow architecture, in brief, can be separated into two components. The scheduler is the component that is responsible for monitoring workflows and tasks and dispatching the corresponding tasks according to their dependencies. On the other hand, we have the workers, who are in charge of executing the logic of the assigned tasks. Airflow can scale, however, it is not designed to support all the workload on the servers themselves. Rather, the workload is usually delegated to other external services. Therefore, running many tasks in parallel, such as a Map-Reduce workflow, can significantly overwhelm the load on the worker nodes and can saturate the entire system.

On the other hand, serverless functions, and in particular the Lithops framework, provide a good opportunity to offload all this massively parallel work off the Apache Airflow cluster.

The advantages of running a serverless workflow with Airflow are clear. On the one hand, one has the instant scalability of serverless functions available. In addition, FaaS services do not charge for idle time, so resources are used more efficiently. On the other hand, Airflow provides a robust, fault-tolerant system for orchestrating complex, long-running workflows, with the ability to repeat failed tasks without the need to rerun the entire workflow from the beginning.

Below we describe the newly developed operators that allow serverless functions to be executed as part of an Airflow workflow using the Lithops framework:

- **LithopsCallAsyncOperator**: It invokes a single function – Utilizes `FunctionExecutor.cal_async` call from Lithops framework

```
def add(x, y):
    return x + y

from my_functions import add
my_task = LithopsCallAsyncOperator(
    task_id='add_task',
    func=add,
    data={'x' : 1, 'y' : 3},
    dag=dag,
)

# Returns:
4

from my_functions import add
basic_task = LithopsCallAsyncOperator(
    task_id='add_task_2',
    func=add,
    data={'x' : 4},
    data_from_task={'y' : 'add_task_1'},
    dag=dag,
)

# Returns:
8
```

Figure 19: Lithops Plugin for Apache Airflow - Call Async Operator

- **LithopsMapOperator**: It invokes multiple functions as a parallel Map – Utilizes `FunctionExecutor.map` call from Lithops framework

```
def add(x, y):  
    return x + y  
  
from my_functions import add  
map_task = LithopsMapOperator(  
    task_id='map_task',  
    map_function=add,  
    map_iterdata=[1, 2, 3],  
    extra_params={'y' : 1},  
    dag=dag,  
)  
  
# Returns:  
[2, 3, 4]
```

Figure 20: Lithops Plugin for Apache Airflow - Call Async Operator

- **LithopsMapReduceOperator:** It invokes a full Map-Reduce job as mutiple parallel functions – Utilizes `FunctionExecutor.map_reduce` call from Lithops framework

```
def add(x, y):  
    return x + y  
  
def mult_array(results):  
    result = 1  
    for n in results:  
        result *= 2  
    return result  
  
from my_functions import add, mult  
mapreduce_task = LithopsMapReduceOperator(  
    task_id='mapreduce_task',  
    map_function=add,  
    reduce_funtion=mul,  
    map_iterdata=[1, 2, 3],  
    extra_params={'y' : 1},  
    dag=dag,  
)  
  
# Returns:  
18
```

Figure 21: Lithops Plugin for Apache Airflow - Map Reduce Operator

These new operators have been developed using the Apache Airflow plugin API, which provides developers with a generic interface that allows new third-party operators to be developed in parallel to the base project. For this case, a Lithops Plugin has been developed for Apache Airlfow that incorporates the operators described above for use in the required DAGs. The Lithops Plugin for Apache Airflow is open source and available on GitHub<sup>5</sup>.

**Use case: Geospatial workflows** As a demonstration of a real scenario where mixing serverful and serverless computation is required, we have implemented a scientific geospatial data analysis workflow from the project's geospatial use case. The workflow is about the computation of the Normalized Difference Vegetation Index (NDVI) of a set of terrestrial plots using geospatial data from the ESA produced by the SENTINEL satellite.

The first phase of the workflow requires to download and transform (pre-process) satellite image data. This process is slow (around 30 minutes), needs more memory than current serverless function services can provide, uses “blackbox” legacy software from the ESA and parallelization is not possible. However, atmospheric and geometric correction is necessary and recommended to minimize distortions in the image that may be caused by clouds or alterations in satellite movement, sensor

<sup>5</sup><https://github.com/lithops-cloud/airflow-plugin>

failure, etc. After the pre-processing and optimization of the data, the transformed satellite data can then be processed in parallel using serverless functions.

Using different task operators in Airflow, we can now seamlessly mix serverful and serverless tasks in the same workflow. In this case, we are using the AWS Batch operator to run the task that requires running in a server instance instead of on a serverless function. Thanks to Apache Airflow's operator design, we can combine different services in the same workflow, as well as monitor and orchestrate using the same environment. Figure 22 shows the DAG representation of the NDVI workflow in Apache Airflow web console, where we can see how AWS Batch and Lithops operators are used.

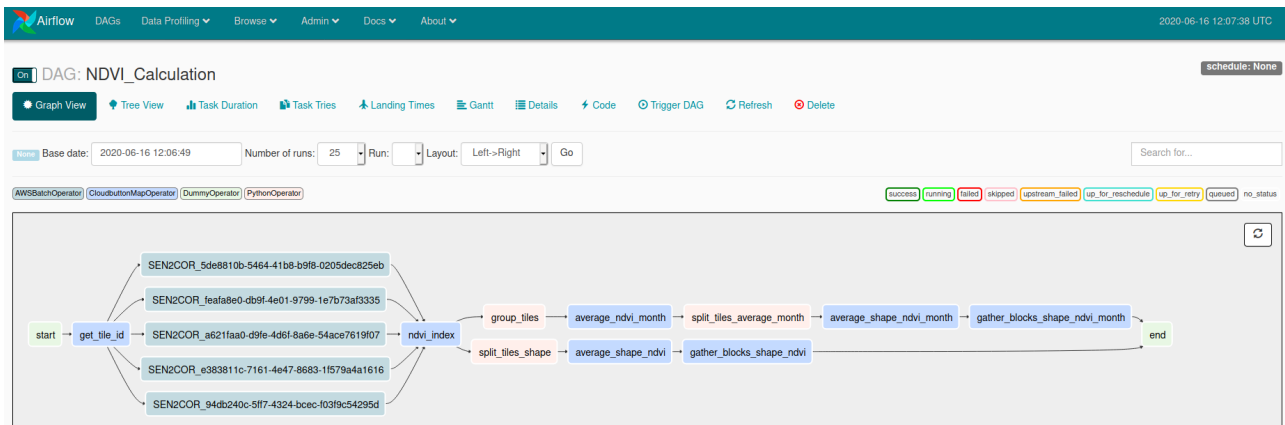


Figure 22: Lithops Plugin for Apache Airflow - NDVI workflow

## 7 SLA Monitoring and Management

### 7.1 Rational behind the CloudButtonSLA component

The architecture of the SLA Management component was explained in D3.1. It offers a solution that handles the complete lifecycle of an SLA management strategy considering two well-differentiated phases: the definition of the contract between the providers and consumers of services, and monitoring its fulfillment in real time.

The development of CloudButtonSLA has taken place making use of the CloudButton testbed, as explained in D2.5. CloudButtonSLA is offered as open source, and it can be installed following the instructions on the readme file at <https://github.com/cloudbutton/sla-management>.

We will describe how the SLA Manager has been adapted to operate as the central point of monitoring and QoS assessment of the execution of Serverless Data Analytics workflows running with Lithops.

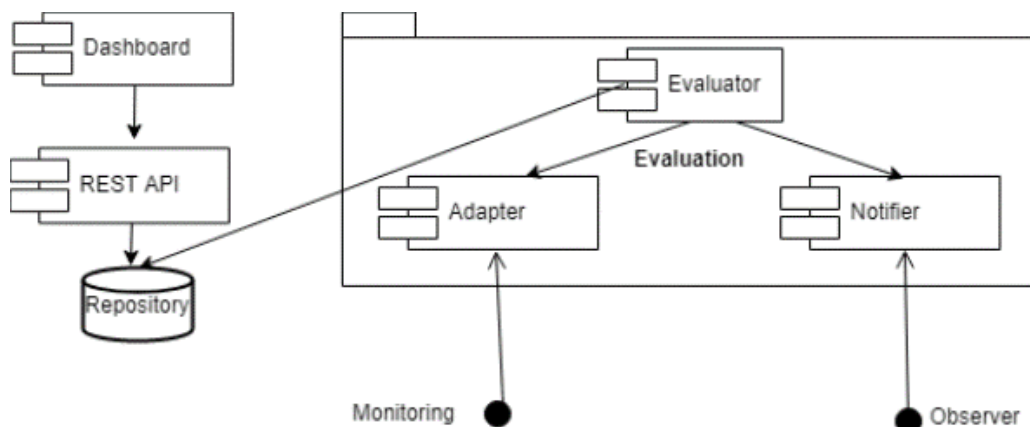


Figure 23: Architecture of SLA Manager.

During the first period, in the context of CloudButton, some of the main components of the SLA Manager were enlarged to serve the CloudButtonSLA needs (Prometheus monitoring adapter, Rabbit and Prometheus Pushgateway observer notifier, etc see D3.2). The objective of the CloudButtonSLA focused on enabling the continuous monitoring of Lithops functions execution by using the metrics provided by Knative and Kubernetes at runtime. In the second period we identified that this scenario has limitations when supervising the execution of Serverless Data analytics workflows:

- The mentioned metrics were based on the performance of the system only, and they could be influenced by side-effects.
- This monitoring will not give information about Lithops using other runtime backends, as IBM Cloud Functions, for example.

Thus, we have worked on the integration of the CloudButtonSLA with Lithops itself, to get a more detailed information about the execution of Serverless Workflows with Lithops.

As explained in <sup>6</sup> Lithops, each map or reduce computation is executed as a separate compute job. FunctionExecutor class is responsible for orchestrating the computation in Lithops. During its initialization it sets up the workers (depending on the specific compute backend), such as constructing docker images, defining IBM Cloud Functions, etc. **All workers of a job are assigned the same amount of memory and are expected to have a similar execution time.** Once job creation is done and the job\_description record for the new job is returned to the FunctionExecutor object, it proceeds to execute the job as a set of independent calls (invocations). This means call invocation is concurrent from the start. FunctionExecutor completes when all calls have notified completion, or a pre-configured timeout has expired.

The rationale of the CloudButtonSLA component in the High Performance Serverless Compute Engine architecture is to describe contexts that affects the QoS of the workflows running with Lithops and inform about anomalous situations. We have identified two contexts and three anomalous situations and defined the corresponding SLA constraints. CloudButtonSLA can react to the anomalous situations with the help of Lithops, or it can just inform and record the violation as a new metric.

#### **Context 1: Execution time of Serverless Data Analytics Workflow.**

In this context we identify the situations that makes a Serverless workflow being run with Lithops to not end in the expected time. We can describe this context by means of two anomalous situations:

- **notStarted:** During initialization, some of the independent calls (functions) that made up a job do not start properly.
- **tooLong:** During execution of the job, some of the independent calls (functions) are taking more than other calls in the same job to end.

#### **Context 2: Total Cost of Serverless Data Analytics Workflow running on IBM Cloud**

In this context, we supervise the cost (in US \$) of the execution of a Serverless Data Analytics Workflow in IBM Cloud Functions from the information provided by IBM <sup>7</sup>. Actually, it will be straightforward to supervise the cost in other context (Google Cloud, AWS, etc) just by applying the corresponding formula. We can describe this cost by means of one anomaly:

- **tooCostly:** The added cost of the execution of all the functions that made up the jobs performed in a single Lithops instance, obtained from the IBM formula, exceeds the total desired cost.

Now that the anomalies that we need to identify are being defined, we can use CloudButtonSLA to:

---

<sup>6</sup>Lithops Architecture Design: <https://lithops-cloud.github.io/docs/source/design.html>

<sup>7</sup>IBM Cloud Functions Cost Calculator <https://cloud.ibm.com/functions/learn/pricing>



- Describe the anomaly in terms of constraints expressed with the metrics obtained by the CloudButtonSLA monitoring.
- Identify a violation of the constraint and take actions to minimise the impact of the anomaly.

As described in D3.2, CloudButtonSLA uses a JSON file to state the SLA agreements that express requirements in terms of QoS. The QoS parameters will be obtained from the monitoring of the Backend FaaS. CloudButtonSLA performs a continuous monitoring and metrics assessment of the fulfillment of the constraints and notifies any breach so that actions can be taken to enforce the agreements.

In the next subsection we will describe how the three anomalies (tooLong, notStarted and tooCostly) are treated with CloudButtonSLA with the help of the information provided by Lithops, and how Lithops uses the notifications of the CloudButtonSLA to recover from an anomaly.

## 7.2 Lithops and CloudButtonSLA integration

### 7.2.1 Lithops metrics

CloudButtonSLA makes use of the Observability and Monitoring tools of the CloudButton Testbed Monitoring Setup, described in D2.5. Lithops has been enlarged to produce the metrics needed to describe contexts that affects the QoS of the workflows running with Lithops.

Lithops sends metrics using the Prometheus Pushgateway everytime a job is created, and a call (function) is started or ends, independently of the backend runtime being used. The metrics are accessible from Prometheus at <http://77.231.202.2:30990/graph> and consumed by CloudButtonSLA, that takes this IP as a configurable parameter for the monitoring adapter.

Here follows the list of all the metrics stored in Prometheus with information about Serverless Data Analytics that run with Lithops. The value and all parameters are set by Lithops from the information of the execution of the instances:

**function\_start** and **function\_end**: The value of this metric corresponds to the UNIX epoch time when the function with the corresponding call\_id has started (ended). There is such a metric for each of the functions that will run concurrently to perform the job identified by job\_id

**Example from function\_end (function\_start is analogous):**

```
1 "metric": {
2   "__name__": "function_end",
3   "call_id": "58d244-0-M001-00000",
4   "exported_instance": "58d244",
5   "exported_job": "lithops",
6   "function_name": "map_interpolation",
7   "instance": "192.168.19.39:9091",
8   "job": "pushgateway",
9   "job_id": "58d244-0-M001",
10  "namespace": "knative-monitoring",
11  "pod": "pushgateway-prometheus-pushgateway-8598ddbfb6-qs6gw"
12 },
13 "value": [
14   1639482191.321,
15   "1639482044.8831794"
16 ]
```

**job\_total\_calls**: The value of this metric is the number of independent calls (functions) that will run concurrently to perform the job identified by job\_id

**Example:**

```
1 "metric": {
2   "__name__": "job_total_calls",
3   "exported_instance": "58d244",
```



```
4   "exported_job": "lithops",
5   "function_name": "map_interpolation",
6   "instance": "192.168.19.39:9091",
7   "job": "pushgateway",
8   "job_id": "58d244-0-M001",
9   "namespace": "knative-monitoring",
10  "pod": "pushgateway-prometheus-pushgateway-8598ddbfb6-qs6gw"
11 },
12 "value": [
13     1639482715.694,
14     "18"
15 ]
```

**job\_runtime\_memory:** The value of this metric is the amount of memory in megabytes assigned to each of the calls (functions) that will run in concurrently to perform the job identified by job\_id

```
1 "metric": {
2   "__name__": "job_runtime_memory",
3   "exported_instance": "58d244",
4   "exported_job": "lithops",
5   "function_name": "map_interpolation",
6   "instance": "192.168.19.39:9091",
7   "job": "pushgateway",
8   "job_id": "58d244-0-M001",
9   "namespace": "knative-monitoring",
10  "pod": "pushgateway-prometheus-pushgateway-8598ddbfb6-qs6gw"
11 },
12 "value": [
13     1639483114.426,
14     "2048"
15 ]
```

Now that we have explained the metrics that we obtain from Lithops during the execution of Serverless Data Analytics workflows, we'll see how they are used to describe the three anomalies in the two different contexts that we mean to prevent.

### 7.2.2 PromSQL queries

Prometheus provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time series data in real time. It is a complex language, based on regular expressions, different operators, etc. to extract information from the metrics that are stored in Prometheus.

We will use PromQL on the Lithops metrics to describe the notStarted, tooLong and tooCostly constraints, and have CloudButtonSLA to define them as agreements and assess the fulfilment of the constraints. We will look for the expressions that can be queried directly in Prometheus to obtain the information we need to assess the constraints that define each anomaly. Let see how the different constraints can be expressed in PromQL:

#### notStarted:

We define the variable **totalNotStartedFunctions** as the total number of functions not started of a job. It can be obtained from the difference between the value of the job\_total\_calls metric for a specific job\_id and the count of the function\_start metrics for this job.

$$totalNotStartedFunctions = sum(job\_total\_calls)by(job\_id) - count(function\_start)by(job\_id)$$

If any of the concurrent functions of a job didn't start, this value will be greater than zero, so we can define the notStarted anomaly with the constraint:

$$totalNotStartedFunctions == 0$$

#### notEnded:

We define the variable **FunctionEndMax** as the maximum expected duration for a function in a job, and it is expressed as the different between the function\_start time (for the functions that didn't end yet) and the average duration time of the functions that have already finished, plus and extra time of a certain percentage of this duration (plus 30% in the example). For example:

```
FunctionEndMax = (function_start unless on(call_id)function_end) - on (job_id)
group_left() (1.3 *avg(function_end-function_start) by (job_id))
```

This query is controlled by a parameter, TFACTOR (time factor), that represent the percentage over the average time of execution that we consider as a limit.

```
FunctionEndMax = (function_start unless on (call_id) function_end) - on(job_id)
group_left() ({{.TFACTOR}} * avg(function_end-function_start) by (job_id))
```

If the actual time (function time(), in Prometheus) minus the FunctionEndMax for any of the running functions of a job is greater than zero, we can conclude that they are taking to finish longer than expected.

```
time() - FunctionEndMax < 0
```

### tooCostly:

We will split the total cost of an experiment, made up by all the jobs belonging to the same instance, (exported\_instance in Prometheus), into the cost of the functions that have already finished plus the cost of the still running functions. Thus, we will define two variables **costNotEndedFunctions** and **costEndedFunctions** and apply the IBM formula to calculate the total cost of an instance.

The cost of the ended functions of an instance in IBM cloud is obtained by adding the duration in milliseconds of all the functions in a job, multiplied by the memory allocated to the functions in this job (always the same), in megabytes, and again adding together the result for all the jobs in the instance.

```
costEndedFunctions = sum(sum(function_end-function_start) by (job_id, exported_instance)
* sum (job_runtime_memory) by (job_id, exported_instance)) by (exported_instance)
```

For the not ended functions, we just consider the time elapsed between the time when the function started and the actual time, and we do the analogous calculation.

```
costNotEndedFunctions = sum(sum(time() - (function_start unless on(call_id)function_end))
by (job_id, exported_instance) * sum(job_runtime_memory) by (job_id, exported_instance))
by (exported_instance)
```

To make sure that the cost of ended and not ended functions for the same instance are added together, we use the following join expression:

```
costFunctions = (sum(sum(function_end-function_start) by (job_id, exported_instance) *
sum(job_runtime_memory) by (job_id, exported_instance)) by (exported_instance)) +
(sum (sum(time() - (function_start unless on(call_id)function_end)) by
(job_id,exported_instance) * sum(job_runtime_memory) by (job_id, exported_instance))
by (exported_instance))
```

The basic IBM rate (as from today) is \$0,000017 GB-s, but in our metrics, we have the memory expressed in megabytes, and the time expressed in milliseconds, so the rate should be scaled to: 0,000000000017 \$ per ms of execution, per Mb of memory assigned. We will also use a parameter to express this IBM rate, as it can change in the future, called UNITCOST.

We will use a parameter called MAXCOST to express the limit in the cost, so that the constraint will be:

$$\{ \{ .UNITCOST \} \} * (costNotEndedFunctions + costEndedFunctions) < \{ \{ .MAXCOST \} \}$$

### 7.2.3 CloudButton agreements and Swagger API

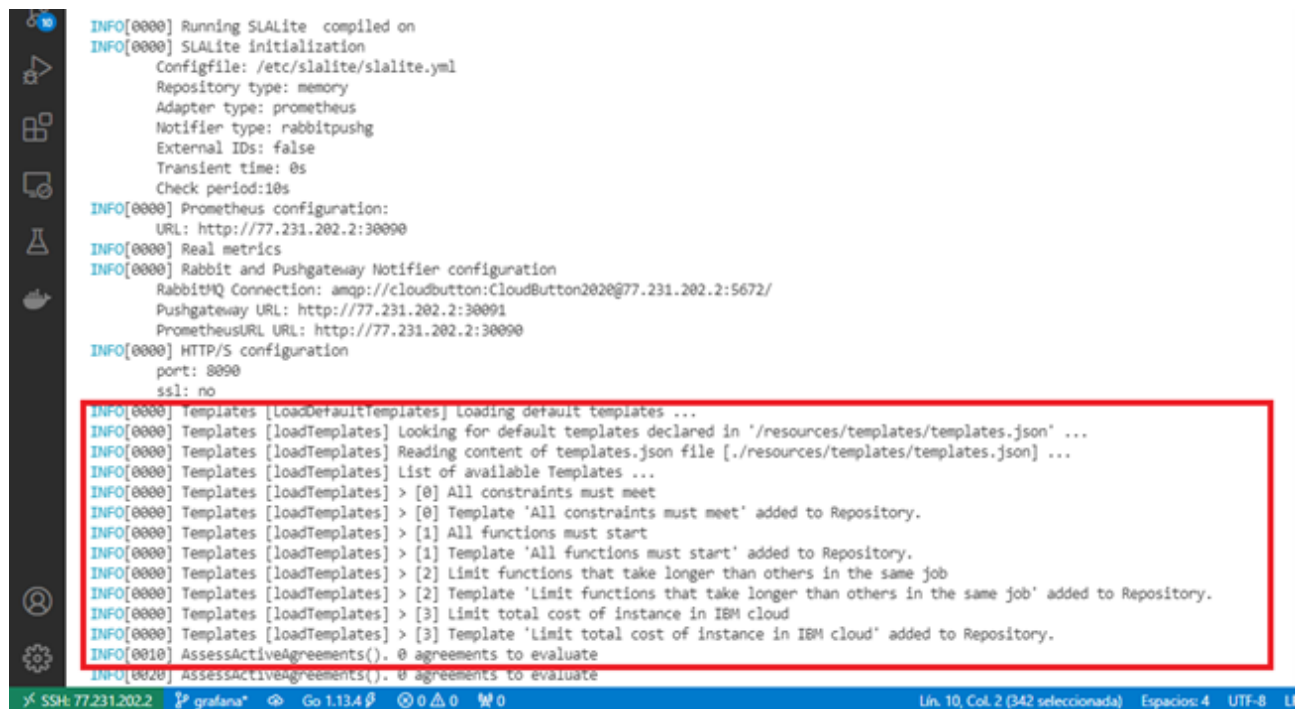
Now that we have a way to express the constraints over the anomalies that we need to supervise in the context of CloudButton execution of Serverless Data Analytics workflows, it is straightforward to define a JSON file to serve as CloudButtonSLA agreement, as described in D3.2.

The Service Level Agreement will be up by several **Agreements** that are assessed by the CloudButtonSLA and notify violations on an independent manner. A **Guarantee** is defined as a complex combination of different metric to define the **Constraints**. For example, the following agreement will help the CloudButtonSLA to identify the notStarted anomaly and submit a violation everytime a function doesn't start. The other agreements (tooLong and tooCostly) will be defined on a similar way from the already defined variables. You can find the agreements in <https://github.com/cloudbutton/sla-management> in folder "/resources".

```
1 {
2   "id": "notStarted",
3   "name": "All functions must start",
4   "state": "started",
5   "details": {
6     "id": "notStarted",
7     "type": "agreement",
8     "name": "All functions must start",
9     "provider": { "id": "a-provider", "name": "A provider" },
10    "client": { "id": "a-client", "name": "A client" },
11    "creation": "2020-01-01T17:09:45Z",
12    "expiration": "2023-01-01T17:09:45Z",
13    "variables": [
14      {
15        "name": "totalNotStartedFunctions",
16        "metric": "sum(job_total_calls)%20by%20(job_id,function_name)-count(
17          function_start)%20by%20(job_id,function_name)"
18      }
19    ],
20    "guarantees": [
21      {
22        "name": "notStarted",
23        "constraint": "totalNotStartedFunctions == 0"
24      }
25    ]
26  }
```

As explained in D3.2, the CloudButtonSLA has a REST API to control the life cycle of SLA agreements (create, activate, modify, etc). In the context of the CloudButton project, the QoS is clearly defined with the three guarantees that control execution time of the workflows running on the testbed (notStarted, tooLong) and the cost (tooCostly) of the experiments running of IBM Cloud.

To simplify the SLA Management, CloudButtonSLA will create the three agreements (notStarted, tooLong and tooCostly) at start up, using SLA Templates. The SLA Templates admit parameters that can modify the constraints in the final agreement.



```
INFO[0000] Running SLALite compiled on
INFO[0000] SLALite initialization
Configfile: /etc/slalite/slalite.yml
Repository type: memory
Adapter type: prometheus
Notifier type: rabbitpushg
External IDs: false
Transient time: 0s
Check period:10s
INFO[0000] Prometheus configuration:
URL: http://77.231.202.2:30090
INFO[0000] Real metrics
INFO[0000] Rabbit and Pushgateway Notifier configuration
RabbitMQ Connection: amqp://cloudbutton:CloudButton2020@77.231.202.2:5672/
Pushgateway URL: http://77.231.202.2:30091
PrometheusURL URL: http://77.231.202.2:30090
INFO[0000] HTTP/S configuration
port: 8090
ssl: no
INFO[0000] Templates [loadDefaultTemplates] Loading default templates ...
INFO[0000] Templates [loadTemplates] Looking for default templates declared in '/resources/templates/templates.json' ...
INFO[0000] Templates [loadTemplates] Reading content of templates.json file ['./resources/templates/templates.json'] ...
INFO[0000] Templates [loadTemplates] List of available Templates ...
INFO[0000] Templates [loadTemplates] > [0] All constraints must meet
INFO[0000] Templates [loadTemplates] > [0] Template 'All constraints must meet' added to Repository.
INFO[0000] Templates [loadTemplates] > [1] All functions must start
INFO[0000] Templates [loadTemplates] > [1] Template 'All functions must start' added to Repository.
INFO[0000] Templates [loadTemplates] > [2] Limit functions that take longer than others in the same job
INFO[0000] Templates [loadTemplates] > [2] Template 'Limit functions that take longer than others in the same job' added to Repository.
INFO[0000] Templates [loadTemplates] > [3] Limit total cost of instance in IBM cloud
INFO[0000] Templates [loadTemplates] > [3] Template 'Limit total cost of instance in IBM cloud' added to Repository.
INFO[0010] AssessActiveAgreements(). 0 agreements to evaluate
INFO[0020] AssessActiveAgreements(). 0 agreements to evaluate
```

Figure 24: SLA logs - initial load of templates

The CloudButton user, instead of submitting an agreement, will only have to submit the value of the parameters (TFACTOR for context 1 or UNITCOST and MAXCOST for context 2 ) in order to create and activate the corresponding agreement. For example, if we have the JSON file called `params_cost.json` with the content:

```
1 {
2   "template_id": "CloudButton_tooCostly",
3   "parameters": {
4     "agreementname": "Total cost of the instance in IBM cloud must be limited",
5     "provider": { "id": "a-provider", "name": "A provider" },
6     "client": { "id": "a-client", "name": "A client" },
7     "UNITCOST": 0.000000000017,
8     "MAXCOST": 1.5
9   }
10 }
```

We can create and start an agreement for context 2 limiting the cost to \$1,5 using:

```
curl -k -X POST -d @ params_cost.json http://localhost:8090/create-agreement
```

Recently, a GUI based in Swagger <sup>8</sup> has been added to simplify the management of agreements and templates. It is accessible from <http://77.231.202.2:8090/swaggerui>

<sup>8</sup>Swagger is an open source set of rules, specifications and tools for developing and describing RESTful APIs: <https://swagger.io/>

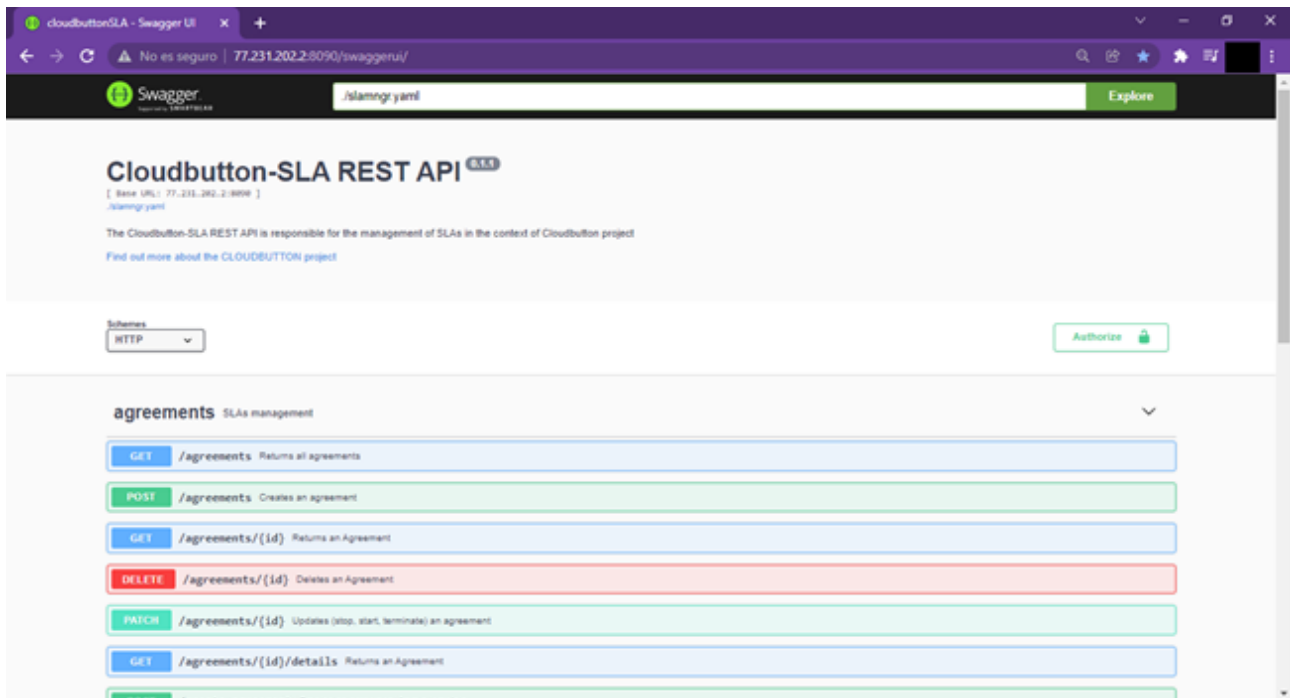


Figure 25: SLA Swagger User Interface

#### 7.2.4 Integration with Lithops through Rabbit queue

In D3.2, we explained how CloudButtonSLA assesses the fulfillment of the constraints in all active agreements and creates a violation message into a Rabbit queue to inform about a breach. This is an example of the content of a Rabbit message for a violation of the tooLong guarantee.

```
1 {
2   "Fields": {
3     "AgreementId": "tooLong",
4     "Guarantee": "tooLong",
5     "ViolationTime": "2021-12-15T13:27:04.307Z"
6   },
7   "Message": [{
8     "key": "2962f6-0-M000-00003",
9     "value": 1639574814.5585732,
10    "datetime": "2021-12-15T13:27:04.307Z",
11    "resource": "summ"
12  }]
13 }
```

Lithops has been enhanced to consume the messages on the CloudButton queue and remediate it. The message shows a **tooLong violation**, and the key '2962f6-0-M000-00003' corresponds to the function that didn't finish in the expected time. Lithops will relaunch the function identified in the key field, and mark the violation as treated. The function will end after relaunch.

Similarly, a **notStarted violation** message will have in the key field the id of the function that didn't, and Lithops will relaunch it. Finally, for the **tooCostly violation**, the key field will have the id of the instance that is overspending, and Lithops, after consuming the message from the queue, will stop the instance and release the resources from IBM Cloud.

The configuration file of lithops has also being enhanced with some new parameters to integrate with CloudButtonSLA:

telemetry -> Indicates that Lithops will send metrics about functions and jobs into Prometheus Pushgateway. Also, the Pushgateway URL and the connection to the Rabbit queue are configurable from the file.

```

-rw-r--r-- 1 root root 246927 May 18 2021 lithops_k8s.zip
-rw-r--r-- 1 1001 1001 754958809 Apr 29 2021 shapefile.zip
(base) root@b311bb4d0eeb:~/work# more .lithops_config
lithops:
  #log_level: None
  storage: ceph
  telemetry: true
  storage_bucket: lithops-cloudbutton

ibm_cf:
  endpoint : https://eu-gb.functions.cloud.ibm.com
  namespace : ruth.palmero@atos.net_dev
  api_key : 1a8abd81-c4c4-468e-9129-5e10651

ibm_cos:
  storage_bucket: lithops-cloudbutton
  region : eu-gb
  api_key : 0W0XWENCZ

ceph:
  storage_bucket: lithops-cloudbutton
  endpoint: http://10.24.17.56:7480
  access_key_id : 1D5M
  secret_access_key : 090n1

k8s:
  runtime: tfmurv/lithops-k8s-geospatial:latest
  runtime_memory: 1024
  runtime_cpu: 0.5
  runtime_timeout: 3600

prometheus:
  apigateway: http://77.231.202.2:30091

rabbitmq:
  amqp_url: amqp://cloudbutton:10.24.17.55:5672

```

Figure 26: Lithops configuration

### 7.2.5 Prometheus Pushgateway Violation notification

As described in D3.2, we added a new notifier to the SLA Manager to send the information of an identified violation to a Rabbit queue, to be consumed by the observer. In the second period, we have enhanced the notifier with new features:

- A violation not only produces a message in a rabbit queue, but it also produces a metric in the Prometheus Pushgateway, called CloudButton\_sla\_QoS\_violation that stores information about the time in which the violation happened, the guarantee that produces it, and depending on the guarantee type, it will store information about the function\_id or the instance affected by the violation. The main use of this is to create Grafana control panels to display information about the execution of an experiment, but many other uses, like maybe predictions on future violations, can be defined in the future.

```

1 {
2   "metric": {
3     "__name__": "CloudButton_sla_QoS_violation",
4     "agreement": "cost",
5     "exported_instance": "935997",
6     "exported_job": "sla",
7     "function_name": "NA",
8     "guarantee": "tooCostly",
9     "instance": "192.168.19.39:9091",
10    "job": "pushgateway",
11    "namespace": "knative-monitoring",
12    "pod": "pushgateway-prometheus-pushgateway-8598ddbfb6-qs6gw",
13    "violation_time": "2021-12-16+15:50:24.037++0000+WET"

```

```

14 },
15 "value": [
16     1639671578.196,
17     "1639669824.0500088"
18 ]
19 }

```

- Each guarantee is treated separately. By default, the SLA Manager assesses and generates violations without considering any difference between them. We have added a module that identifies the constraint that generates the violation and adds some extra logic. For example, from the assessment we obtain the number of functions from a job that didn't start, and the additional logic identifies the function\_id of all the non-started functions, so that the information submitted when the violation is recorded is enriched.

## 7.2.6 A full sample of context

Let's now use a sample notebook that creates a compute job in Lithops composed by four serverless functions that perform a simple task in parallel. We will use a "sleep()" to make one of the functions to take much longer to perform than the others, so that the CloudButtonSLA will generate a tooLong violation. Lithops will react to the notification of the violation by relaunching the function.

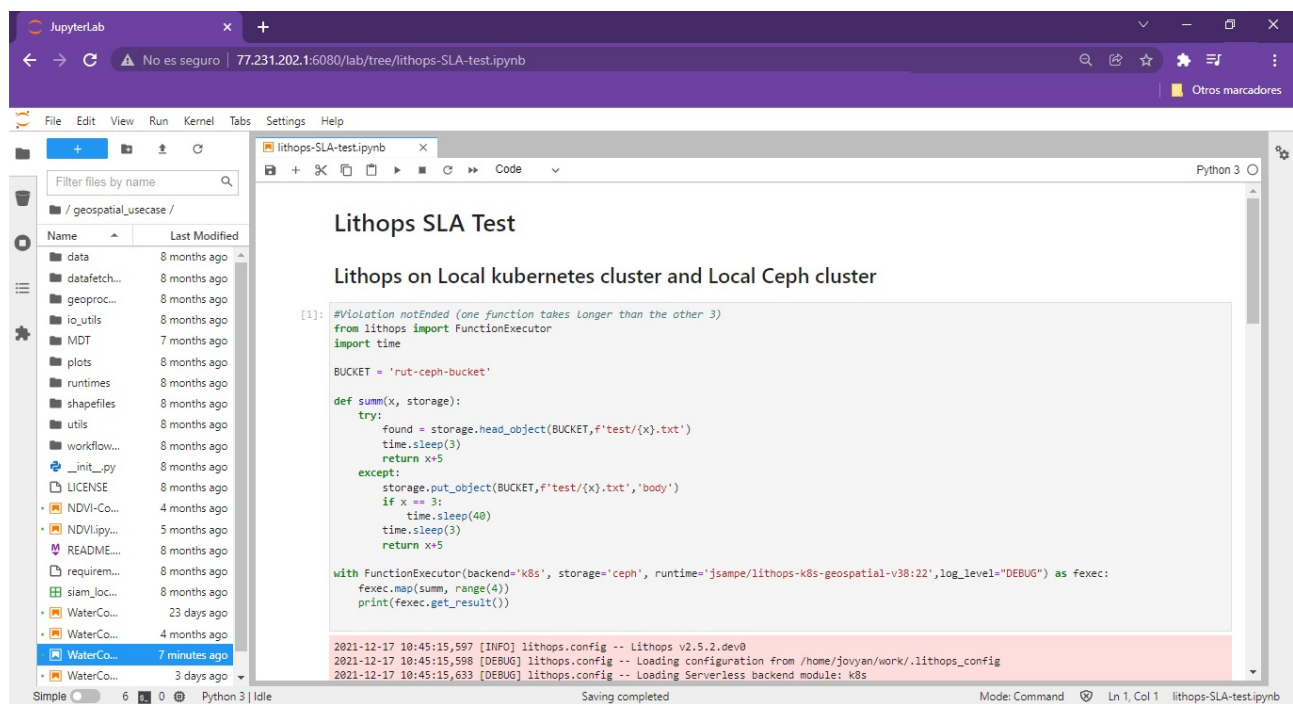


Figure 27: Notebook to run. The task consists of adding 5 to a number. To avoid entering in a loop of violations, we use the creation of a file in the Object Storage as a flag. The function with id=3 will sleep for 40s, and will generate a violation of the CloudButtonSLA



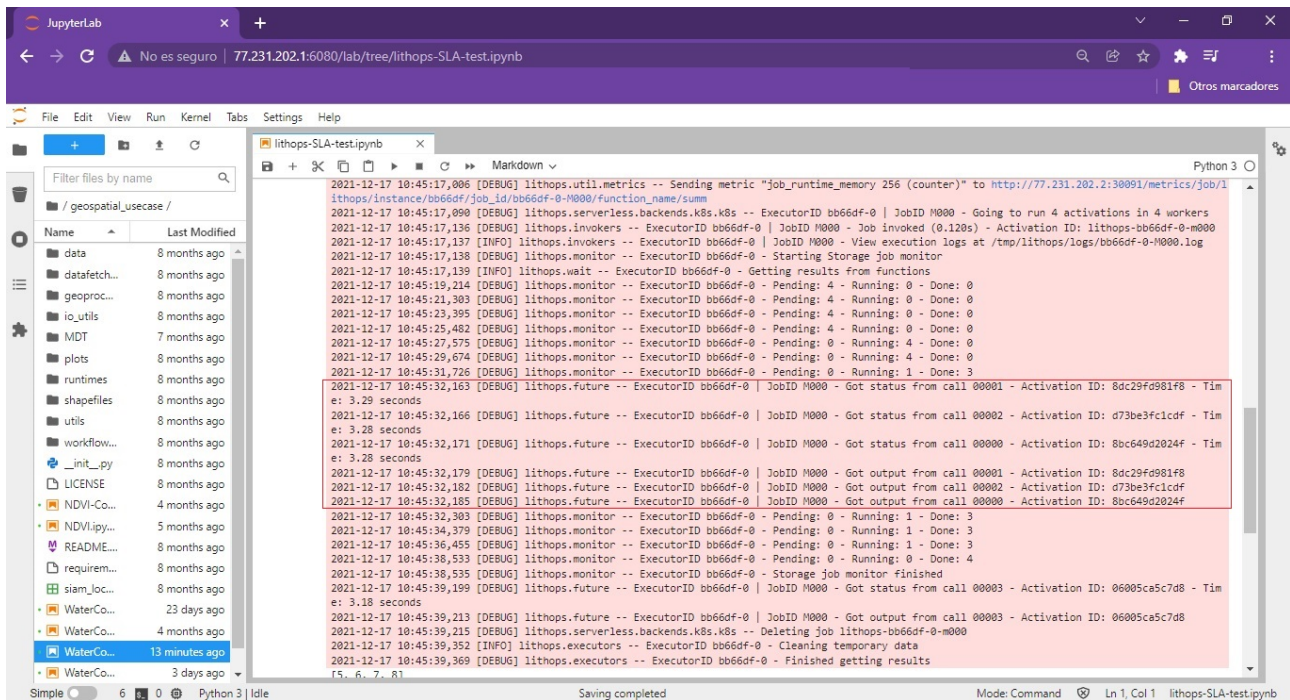


Figure 28: From the logs in the notebook, we see that the functions will call id 0, 1 and 2, ends after around 3ms (their sleep time). The function with call id 3, is still “sleeping”.

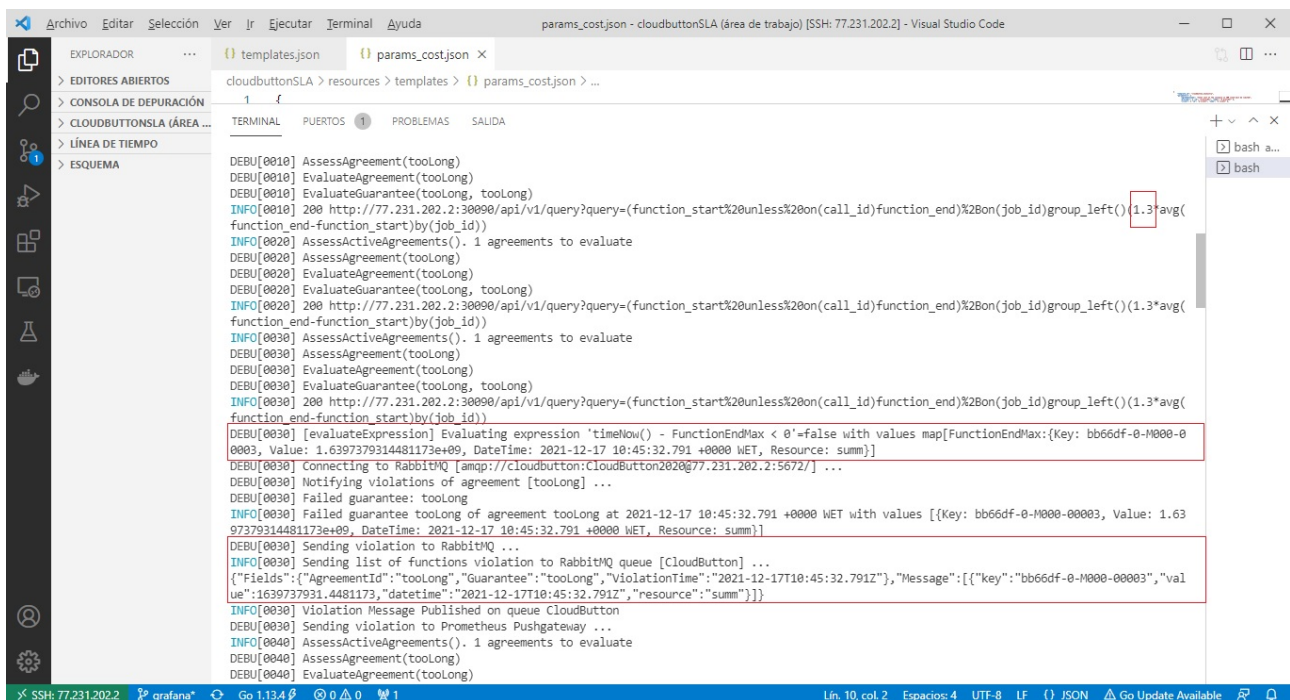


Figure 29: The CloudButton- SLA checks that no function is taking more than 30% over the average time of the other functions in the job to finish. So after around 1,3\*3 ms, it generates a violation and send the message to the rabbit queue.



```
2021-12-17 10:45:36,455 [DEBUG] lithops.monitor -- ExecutorID bb66df-0 - Pending: 0 - Running: 1 - Done: 3
2021-12-17 10:45:38,533 [DEBUG] lithops.monitor -- ExecutorID bb66df-0 - Pending: 0 - Running: 0 - Done: 4
2021-12-17 10:45:38,535 [DEBUG] lithops.monitor -- ExecutorID bb66df-0 - Storage job monitor finished
2021-12-17 10:45:39,199 [DEBUG] lithops.future -- ExecutorID bb66df-0 | JobID M000 - Got status from call 00003 - Activation ID: 06005ca5c7d8 - Time: 3.18 seconds
2021-12-17 10:45:39,213 [DEBUG] lithops.future -- ExecutorID bb66df-0 | JobID M000 - Got output from call 00003 - Activation ID: 06005ca5c7d8
2021-12-17 10:45:39,215 [DEBUG] lithops.serverless.backends.k8s.k8s -- Deleting job lithops-bb66df-0-m000
2021-12-17 10:45:39,352 [INFO] lithops.executors -- ExecutorID bb66df-0 - Cleaning temporary data
2021-12-17 10:45:39,369 [DEBUG] lithops.executors -- ExecutorID bb66df-0 - Finished getting results
[5, 6, 7, 8]
```

Figure 30: From the notebook logs we see that the call 3 ends in around 3ms, as the other functions in the job after relaunching. The job is deleted after execution and the result is displayed.

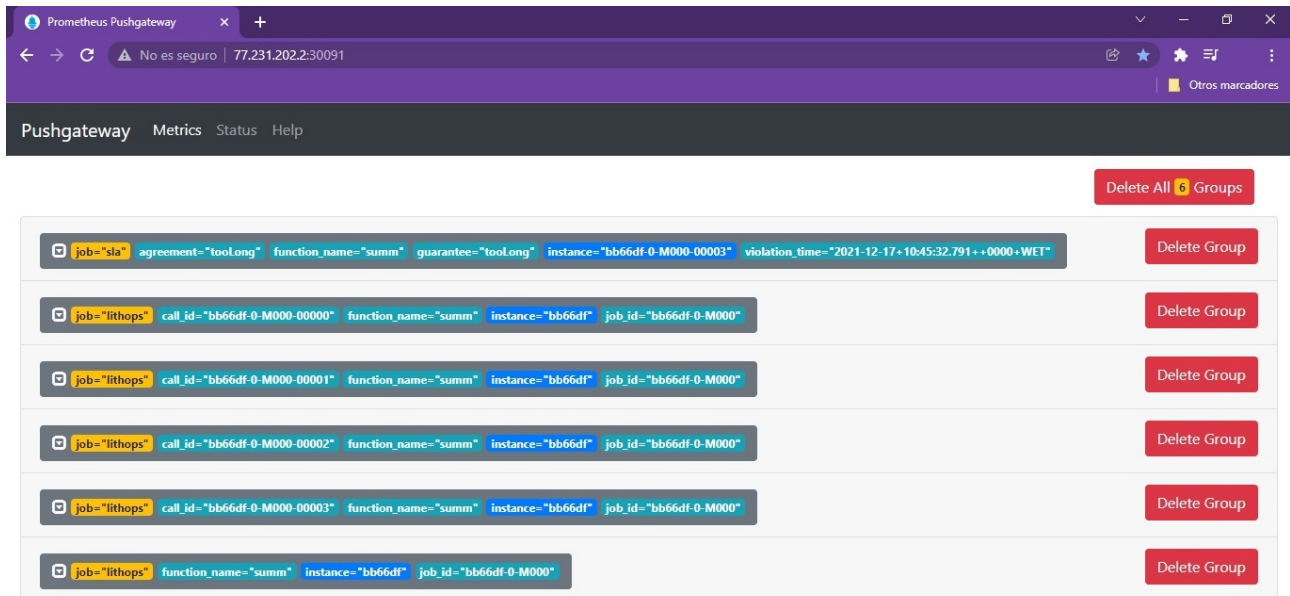


Figure 31: Prometheus pushgateway stores the metrics generated by Lithops (for the job and for each of the functions) and by the CloudButtonSLA (for the violation).

Prometheus Pushgateway

No es seguro | 77.231.202.2:30091

Pushgateway Metrics Status Help

Delete All 6 Groups

job="sla" agreement="tooLong" function\_name="summ" guarantee="tooLong" instance="bb66df-0-M000-00003" violation\_time="2021-12-17+10:45:32.791++0000+WET" Delete Group

CloudButton\_sla\_QoS\_violation Date of last violation. GAUGE last pushed: 2021-12-17T10:45:32Z

Labels	Value
agreement="tooLong" function_name="summ" guarantee="tooLong" instance="bb66df-0-M000-00003" job="sla" violation_time="2021-12-17+10:45:32.791++0000+WET"	1639737932.7997067

push\_failure\_time\_seconds Last Unix time when changing this group in the Pushgateway failed. GAUGE last pushed: 2021-12-17T10:45:32Z

push\_time\_seconds Last Unix time when changing this group in the Pushgateway succeeded. GAUGE last pushed: 2021-12-17T10:45:32Z

job="lithops" call\_id="bb66df-0-M000-00000" function\_name="summ" instance="bb66df" job\_id="bb66df-0-M000" Delete Group

Prometheus Pushgateway

No es seguro | 77.231.202.2:30091

Pushgateway Metrics Status Help

job="lithops" call\_id="bb66df-0-M000-00003" function\_name="summ" instance="bb66df" job\_id="bb66df-0-M000" Delete Group

job="lithops" function\_name="summ" instance="bb66df" job\_id="bb66df-0-M000" Delete Group

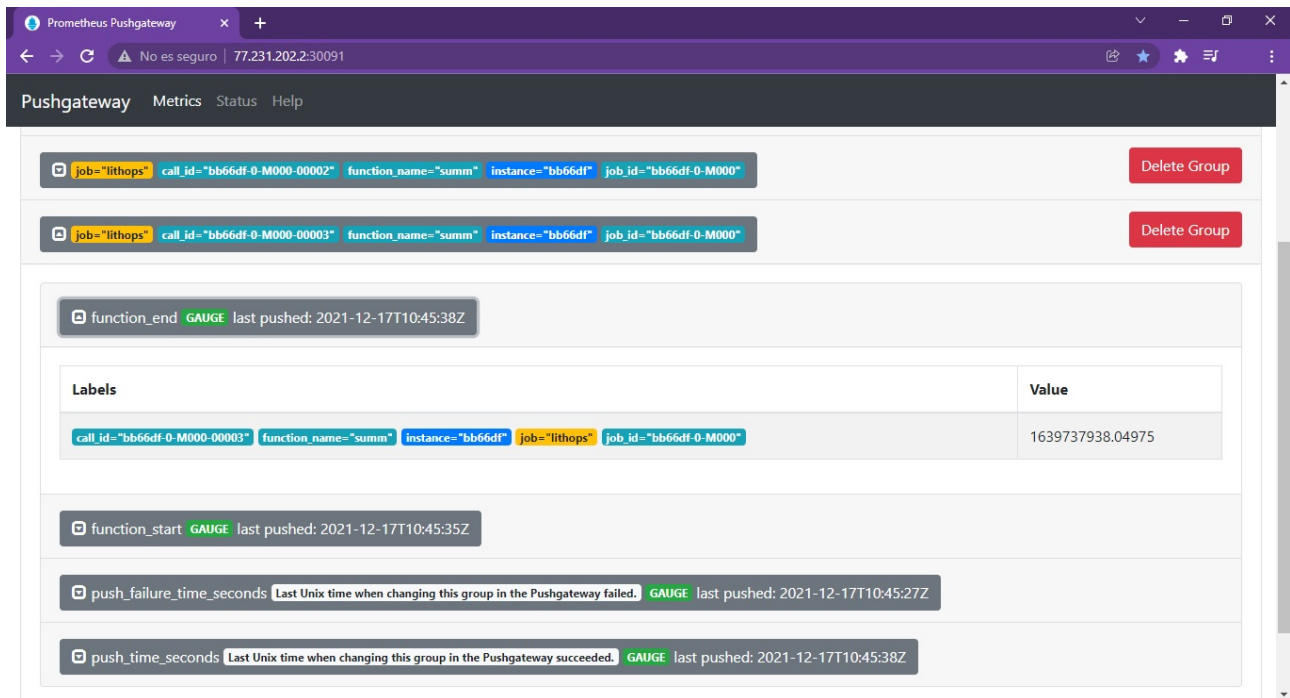
job\_runtime\_memory COUNTER last pushed: 2021-12-17T10:45:17Z

job\_total\_calls COUNTER last pushed: 2021-12-17T10:45:17Z

Labels	Value
function_name="summ" instance="bb66df" job="lithops" job_id="bb66df-0-M000"	4

push\_failure\_time\_seconds Last Unix time when changing this group in the Pushgateway failed. GAUGE last pushed: 2021-12-17T10:45:17Z

push\_time\_seconds Last Unix time when changing this group in the Pushgateway succeeded. GAUGE last pushed: 2021-12-17T10:45:17Z



A full demonstration video can be found in the following URL:

[https://drive.google.com/file/d/1orlMe1KWtjNGu30\\_vcn9MNFLcSultmkkg/view](https://drive.google.com/file/d/1orlMe1KWtjNGu30_vcn9MNFLcSultmkkg/view)

## 7.2.7 CloudButtonSLA cost control panel in Grafana

We have used Grafana to create a Cost control panel like the following:

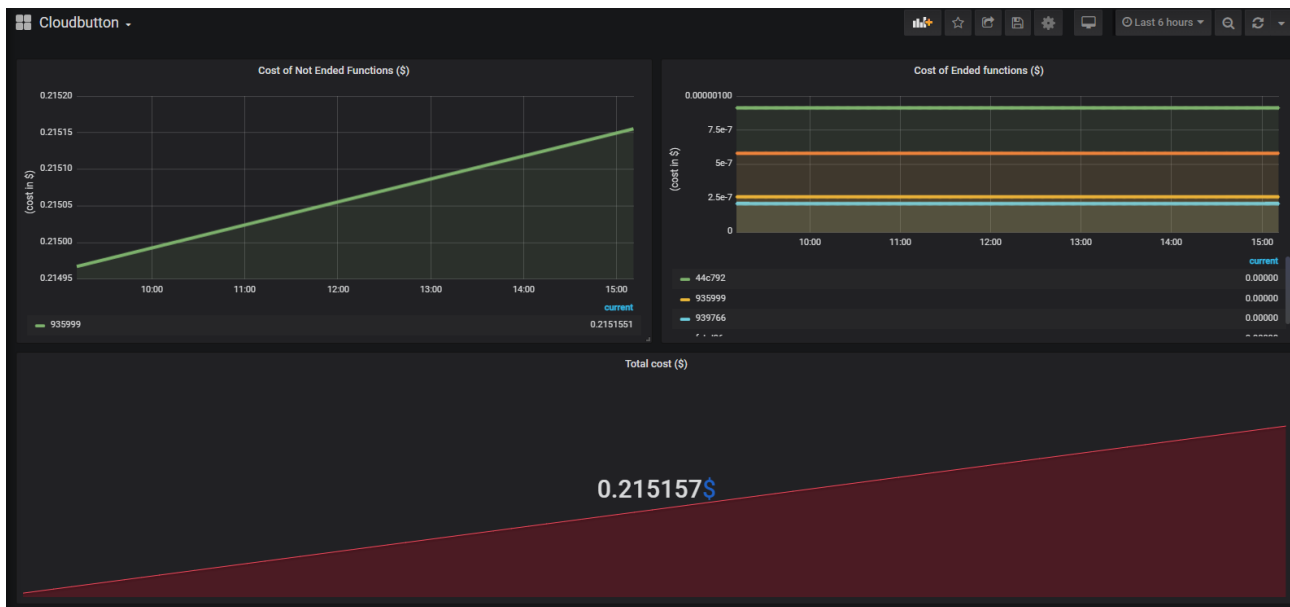


Figure 32: Grafana User Interface - Costs panel

This panel shows the cost of finished or ended functions, the cost of unfinished functions, and the total. In the previous image, we can see how the total cost is continuously growing as the unfinished functions are still running.

### 7.3 Predicted metrics with Prometheus holt\_winters and predictor\_linear

In the first part of the project the idea was to obtain information about the execution of the Serverless Data workflows on the testbed looking at the metrics generated by different sources, described in D2.5. Lately we considered that generating specific metrics provided a better solution to QoS.

Nevertheless, the configuration of CloudButtonSLA has been enhanced with new features that are not directly applied to CloudButton project, but that could be used in a context of QoS assessment in Serverless Data Analytics workflows. The variables to use to define constraints on an agreement are not limited to the metrics that are specifically generated by Lithops that we have presented in this document. Any metric on Prometheus can be used to define an SLA. The code located in (<https://github.com/cloudbutton/sla-management>) provides an example (in folder "/resources/samples" - template-cpu.json) of how a metric generated by node-exporter can be used to identify percentage of idle jobs and launch a violation. You can find more information about system metrics and node-exporter in D2.5.

Together with the possibility to use system metrics, in the second half of the project we have enhanced the SLA Manager with the option to predict states of the system, and thus, anticipate the occurrence of a violation. For these we use the Prometheus query functions `holt_winters()` and `predict_linear()`. They use statistical time series forecasting to identify trends and seasonal data and anticipate future values of the data looking at past values. Both `holt_winters` and `predict_linear` need a Prometheus range vector to obtain a prediction, that is, a time series of the values of the metric to predict over a past period.

- `holt_winters(v range-vector, sf scalar, tf scalar)` produces a smoothed value for time series based on the range in `v`. The lower the smoothing factor `sf`, the more importance is given to old data. The higher the trend factor `tf`, the more trends in the data is considered. Both `sf` and `tf` must be between 0 and 1.
- `predict_linear(v range-vector, t scalar)` predicts the value of time series `t` seconds from now, based on the range vector `v`, using simple linear regression.

The use of this functions is limited to gauge data in Prometheus, a metric that represents a single numerical value that can arbitrarily go up and down. The value is stored together with the moment in time in which it was produced, and thus they are treated as mathematical time series. Many of the metrics obtained by the system and Kubernetes are Prometheus gauges metrics.

For the CloudButtonSLA to work on prediction mode, some configuration variables have been created:

- `prometheusPredictor` : Prometheus Predictor type. By default, is empty (Real metrics), it can be set to "holt\_winters" or "predict\_linear"
- `HWSmoothingFactor` : Holt-Winters Smoothing Factor. By default, is 0.5
- `HWTrendFactor`: Holt-Winters Trend Factor. By default, is 0.5
- `PLScalar`: Prediction Linear Scalar. By default, is 30

Here is an example of a `/etc/slalite/slalite.yml` file to configure CloudButtonSLA with predictivity options:

```
adapter: prometheus -> (The monitoring metrics are obtained from Prometheus)
checkPeriod: 10s -> (Time to wait before querying metrics and performing the assessment)
transientPeriod: 60s -> (Time to wait before repeating a violation already announced)
notifier: rabbitpushg -> (A violation will be notified to both Prometheus and rabbit)
rabbitMQ: amqp://CloudButton:XXXXXXXX@77.231.202.2:5672/ -> (Rabbit system access)
prometheusUrl: http://77.231.202.2:30090 -> (URL to access Prometheus )
```

```
prometheusPredictor: holt_winters -> (Function to use on prediction of Gauge metrics)
HWSmoothingFactor: 0.7 -> (Smooth Factor for the Holt Winter prediction)
HWTrendFactor: 0.7 -> (Trend Factor for the Holt Winter prediction)
```

Now, let's think that we need to supervise the CPU usage of the Lithops pods created to execute Serverless Data Analytics workflows. In Prometheus the metric named `container_cpu_usage_seconds_total` stores the cumulate CPU usage for pods in a Kubernetes cluster. We can filter Lithops containers only with `container_cpu_usage_seconds_total{container="lithops"}` and to get a range vector of values, we can query the time series over the last hour with `container_cpu_usage_seconds_total{container="lithops"}[60m]`

We can create an agreement with the following constraint and start supervising it with CloudButtonSLA.

```
1 "variables": [
2   {
3     "name": "lithops_cpu_usage",
4     "metric": "container_cpu_usage_seconds_total{container=\"lithops\"}[60m]"
5   }
6 "guarantees": [
7   {
8     "name": "CPU Guarantee",
9     "constraint": "lithops_cpu_usage <= 20"
10  }
```

If we configure CloudButtonSLA with Real Metrics, it will send a violation when the cumulate CPU usage of a Lithops container is more than 20s, BUT, if we configure CloudButtonSLA with `holt_winter` prediction and `sf=tf=0.7`, it will send a violation when the cumulate CPU usage of a Lithops container is expected to reach 20s, before it happens, and the remediation can take place in time. CloudButtonSLA will automatically substitute the metric in the agreement by the predicted metric when calling Prometheus:

```
"container_cpu_usage_seconds_total{container=\"lithops\"}[60m]" ->
"holt_winters(container_cpu_usage_seconds_total{container="lithops"}[60m],0.7,0.7)"
```

For the prediction to be accurate we need to test with different `sf` and `tf` factors, and see which ones give the best prediction. The same with `predict_linear` and the `t` scalar.

## 8 Summary, conclusions and the next steps

We developed Serverless Compute Engine platform for Big Data which is based on the Lithops framework and also contains advanced SLA and Monitoring. We released Lithops to the open source and it is now mature, production stable framework. Lithops enables users to easily deploy their production workloads against any public or private clouds, while users focus on their business logic and Lithops handle all deployments, invocations, parallelism, monitoring and so on. This unique experience enables users to deploy more workloads to the cloud without being tied to particular cloud and without vendor lock-in. Lithops is an open source project, with growing community and being used in various use cases outside of the CloudButton scope. In context of CloudButton, Lithops is also used as a "glue" between various packages developed by other participants of the project. In particular, C++ - Fasm integration with Lithops and WebAssembly, Shell scripts and Crucial, are all using Lithops to connect to swrverless platforms.

### 8.1 Applications

Lithops enables variety of use cases and applications in the broad scope, beyond CloudButton use cases. We successfully demonstrated how Lithops and technologies we developed enable users to deploy various workloads against serverless backend engines. Among applications and use cases, we have successfully demonstrated the following use cases

- AirBnB sentiment analysis of the comments from multiple cities

- Lithops toolkit: Compute and Storage Benchmarks
- Cloudbutton Mandelbrot Set Calculation Example
- GROMACS Computations
- Lithops Moments in Time dataset example
- Monte Carlo Simulations with Lithops
- Hyperparameter tuning grid search example

All examples and demos are public and can be accessed here [102]

## **8.2 Adoption of the platform by 3rd party developers**

We maintain Lithops as an open source project with constantly growing community. The visibility of the project goes way beyond CloudButton and by date we have 32 contributors, more than 220 stars and project is being used by 40+ different consumers. To easy adoption of the project we provided detail documentation web site with tutorials, usage examples as well detailed instructions how to integrate Lithops into other third-party application without major disruption or rewriting them from scratch. Also, since Lithops exposes standard Python's API it became very easy to integrate Lithops into existing code or application, without major disruption to those applications.

## **8.3 Next steps**

We continue to maintain and further develop Lithops framework. Lithops is being used in production by EMBL and there is interest from other organizations to leverage Lithops into their production use cases. We continue efforts for better visibility of the project and we plan to publish more blogs and identify additional use cases and application where Lithops can provide benefit. We continue to maintain the open source aspects of the project and invest efforts into growing the open source community around this project.

## References

- [1] Infinispan, "Infinispan." <https://infinispan.org/>.
- [2] "Lithops - github." <https://github.com/lithops-cloud/lithops>, 2021.
- [3] SD Times, "Amazon Introduces Lambda, Containers at AWS re:Invent." <https://infinispan.org/>, 2014.
- [4] AWS, "Amazon Step Functions." <https://aws.amazon.com/step-functions/>.
- [5] Microsoft Azure, "What are Durable Functions?." <https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview>.
- [6] OpenWhisk, "Apache OpenWhisk Composer." <https://github.com/apache/incubator-openwhisk-composer>.
- [7] A. Airflow, "Apache Airflow documentation." <http://airflow.apache.org/>. Accessed on June 2019.
- [8] Kubeflow, "Kubeflow: The Machine Learning Toolkit for Kubernetes." <https://www.kubeflow.org/>.
- [9] Argo, "Argo Workflows & Pipelines: Container Native Workflow Engine for Kubernetes supporting both DAG and step based workflows." <https://argoproj.github.io/argo/>.
- [10] Fission, "Fission WorkFlows." <https://github.com/fission/fission-workflows>.
- [11] IBM, "IBM Cloud Functions." <https://www.ibm.com/cloud/functions>.
- [12] "Apache OpenWhisk." <https://openwhisk.apache.org/>.
- [13] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, p. 445–451, 2017.
- [14] J. Italo, "Facial mapping (landmarks) with Dlib + Python." <https://towardsdatascience.com/facial-mapping-landmarks-with-dlib-python-160abcf7d672>.
- [15] Denis Kennely, "Three Reasons most Companies are only 20 Percent to Cloud Transformation." <https://www.ibm.com/blogs/cloud-computing/2019/03/05/20-percent-cloud-transformation/>.
- [16] TechRepublic, "Rise of Multi-Cloud: 58% of businesses using combination of AWS, Azure, or Google Cloud." <https://www.techrepublic.com/article/rise-of-multicloud-58-of-businesses-using-combination-of-aws-azure-or-google-cloud/>.
- [17] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study," Journal of Systems and Software, vol. 126, pp. 1 – 16, 2017.
- [18] "Apache Mesos." <http://mesos.apache.org/>.
- [19] Docker, "Docker, Swarm Mode." <https://docs.docker.com/engine/swarm/>.
- [20] Kubernetes, "Kubernetes: Production-Grade Container Orchestration." <https://kubernetes.io/>.
- [21] "Plasma Object Storage."

- [22] N. W. Paton and O. Díaz, "Active database systems," ACM Computing Surveys (CSUR), vol. 31, no. 1, pp. 63–103, 1999.
- [23] C. Mitchell, R. Power, and J. Li, "Oolong: asynchronous distributed applications made easy," in Proceedings of the Asia-Pacific Workshop on Systems, p. 11, ACM, 2012.
- [24] S. Han and S. Ratnasamy, "Large-scale computation not at the cost of expressiveness," in Presented as part of the 14th Workshop on Hot Topics in Operating Systems, 2013.
- [25] A. Geppert and D. Tombros, "Event-based distributed workflow execution with eve," in Middleware'98, pp. 427–442, Springer, 1998.
- [26] W. Chen, J. Wei, G. Wu, and X. Qiao, "Developing a concurrent service orchestration engine based on event-driven architecture," in OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", pp. 675–690, Springer, 2008.
- [27] W. Binder, I. Constantinescu, and B. Faltings, "Decentralized orchestration of composite web services," in 2006 IEEE International Conference on Web Services (ICWS'06), pp. 869–876, IEEE, 2006.
- [28] G. Li and H.-A. Jacobsen, "Composite subscriptions in content-based publish/subscribe systems," in ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, pp. 249–269, Springer, 2005.
- [29] D. Dai, Y. Chen, D. Kimpe, and R. Ross, "Trigger-based incremental data processing with unified sync and async model," IEEE Transactions on Cloud Computing, 2018.
- [30] P. Soffer, A. Hinze, A. Koschmider, H. Ziekow, C. Di Ciccio, B. Koldehofe, O. Kopp, A. Jacobsen, J. Sürmeli, and W. Song, "From event streams to process models and back: Challenges and opportunities," Information Systems, vol. 81, pp. 181–200, 2019.
- [31] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," in Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, pp. 89–103, 2017.
- [32] B. Carver, J. Zhang, A. Wang, and Y. Cheng, "In search of a fast and efficient serverless dag engine," arXiv preprint arXiv:1910.05896, 2019.
- [33] S. Joyner, M. MacCoss, C. Delimitrou, and H. Weatherspoon, "Ripple: A practical declarative programming framework for serverless compute," arXiv preprint arXiv:2001.00222, 2020.
- [34] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions," Future Generation Computer Systems, in press.
- [35] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," Proceedings of the ACM on Programming Languages, vol. 3, no. OOPSLA, pp. 1–26, 2019.
- [36] E. Van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. Abad, and A. Iosup, "The spec-rg reference architecture for faas: From microservices and containers to serverless platforms," IEEE Internet Computing, 2019.
- [37] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), (Renton, WA), pp. 475–488, USENIX Association, July 2019.



- [38] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto, "Comparison of faas orchestration systems," in 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 148–153, IEEE, 2018.
- [39] D. Barcelona-Pons, P. García-López, A. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas, "Faas orchestration of parallel workloads," in Proceedings of the 5th International Workshop on Serverless Computing, WOSC '19, (New York, NY, USA), p. 25–30, Association for Computing Machinery, 2019.
- [40] Ray, "Scaling Python ML workloads." <https://www.ray.io>.
- [41] Python, "Multiprocessing." <https://docs.python.org/3/library/multiprocessing.html>.
- [42] Python, "concurrent.futures." <https://docs.python.org/3/library/concurrent.futures.html>.
- [43] CloudButton Consortium, "CloudButton Toolkit implementation for IBM Cloud Functions and IBM Cloud Object Storage." <https://github.com/pywren/pywren-ibm-cloud>.
- [44] Cloudbutton team, "Cloudbutton Toolkit." <https://github.com/cloudbutton>.
- [45] "Amazon Lambda:." <https://aws.amazon.com/lambda/>, 2020.
- [46] "AWS Batch:." <https://aws.amazon.com/batch/>, 2022.
- [47] "Google Cloud Functions:." <https://cloud.google.com/functions>, 2020.
- [48] "Google Cloud Run:." <https://cloud.google.com/run>, 2022.
- [49] "Azure Functions:." <https://azure.microsoft.com/en-us/services/functions/>, 2020.
- [50] Lithops, "Lithopscld cli." <https://github.com/lithops-cloud/lithopscld>, 2022.
- [51] CloudButton Consortium, "Deliverable D4.3: Full implementation of the BLOSSOM middleware."
- [52] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in 29th Annual ACM Symposium on Theory of Computing, STOC, 1997.
- [53] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," SIGOPS Oper. Syst. Rev., vol. 44, Apr. 2010.
- [54] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings, 2019.
- [55] The JPype Project, "JPype." <https://github.com/jpype-project/jpype>.
- [56] "The crucial project - github." <https://github.com/crucial-project>, 2020.
- [57] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2017.
- [58] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in IEEE Symposium on Security and Privacy, 2009.

- [59] Asmjs.org, “asm.js.” <http://asmjs.org/>.
- [60] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in USENIX Annual Technical Conference (USENIX ATC), USENIX Association, 2020.
- [61] A. Hall and U. Ramachandran, “An execution model for serverless functions at the edge,” in Proceedings of the International Conference on Internet of Things Design and Implementation, 2019.
- [62] R. Gurdeep Singh and C. Scholliers, “Warduino: A dynamic webassembly virtual machine for programming microcontrollers,” in Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Association for Computing Machinery, 2019.
- [63] Fastly, “Terrarium.” <https://wasm.fastlylabs.com/>, 2020.
- [64] Microsoft Research, “Krustlet.” <https://deislabs.io/posts/introducing-krustlet/>.
- [65] WebAssembly, “WebAssembly Specification.” <https://github.com/WebAssembly/spec/>, 2020.
- [66] C. Watt, “Mechanising and Verifying the WebAssembly Specification,” in ACM SIGPLAN International Conference on Certified Programs and Proofs, 2018.
- [67] W. Fu, R. Lin, and D. Inge, “TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly,” arXiv preprint arXiv:1802.01050, 2018.
- [68] D. Lehmann and M. Pradel, “Wasabi: A Framework for Dynamically Analyzing WebAssembly,” in ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.
- [69] LLVM Project, “LLVM 9 Release Notes.” <https://releases.llvm.org/9.0.0/docs/ReleaseNotes.html>, 2020.
- [70] Assemblyscript, “AssemblyScript.” <https://github.com/AssemblyScript/assemblyscript>, 2020.
- [71] SwiftWasm, “SwiftWasm.” <https://swiftwasm.org/>.
- [72] A. Andreev, “TeaVM.” <http://www.teavm.org/>, 2020.
- [73] WebAssembly Working Group, “Roadmap.” <https://webassembly.org/roadmap/>, 2022.
- [74] Bytecode Alliance, “WebAssembly Binary Toolkit.” <https://github.com/WebAssembly/wabt>, 2022.
- [75] WebAssembly, “WebAssembly Dynamic Linking.” <https://webassembly.org/docs/dynamic-linking/>, 2020.
- [76] C. Reference, “The C++ language.” <https://en.cppreference.com/w/cpp/language>, 2022.
- [77] LLVM Project, “LLVM Compiler Infrastructure.” <https://llvm.org/docs/>, 2020.
- [78] L. Project, “Clang: a C language family front-end for LLVM.” <https://clang.llvm.org/>, 2022.
- [79] The Rust Language, “Rust Toolchains.” <https://rust-lang.github.io/rustup/concepts/toolchains.html>, 2020.
- [80] Google, “The Go Programming Language.” <https://go.dev/>, 2020.

- [81] Bytecode Alliance, "WASI: WebAssembly System Interface." <https://wasi.dev/>, 2022.
- [82] A. Scheidecker, "WAVM." <https://github.com/WAVM/WAVM>, 2020.
- [83] Bytecode Alliance, "The WebAssembly Micro Runtime." <https://github.com/bytecodealliance/wasm-micro-runtime>, 2020.
- [84] Bytecode Alliance, "Wasmtime." <https://wasmtime.dev/>, 2020.
- [85] Google, "V8 Engine." <https://github.com/v8/v8>, 2020.
- [86] Mozilla, "SpiderMonkey." <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey>.
- [87] Microsoft, "ChakraCore." <https://github.com/microsoft/ChakraCore>.
- [88] Apple, "WebKit." <https://webkit.org/>.
- [89] Bytecode Alliance, "The Bytecode Alliance." <https://bytecodealliance.org/>, 2022.
- [90] Bytecode Alliance, "WASI Design Principles." <https://github.com/WebAssembly/WASI/blob/main/docs/DesignPrinciples.md>, 2020.
- [91] C. Watt, A. Rossberg, and J. Pichon-Pharabod, "Weakening WebAssembly," Proceedings of the ACM on Programming Languages (PACMPL), 2019.
- [92] CloudButton Consortium, "Deliverable D5.2: CloudButton Prototype of Abstractions, Fault-tolerance and Porting Tools."
- [93] Faasm Contributors, "High-performance stateful serverless runtime based on WebAssembly." <https://github.com/faasm/faasm>, 2021.
- [94] Faasm Contributors, "Tools for building C/C++ to WebAssembly for Faasm." <https://github.com/faasm/cpp>, 2021.
- [95] The Knative Authors, "Knative - Enterprise-grade Serverless on your own terms.." <https://knative.dev/docs/>, 2021.
- [96] The Linux Foundation, "Kubernetes." <https://kubernetes.io/>, 2020.
- [97] LDS Group, "Faasm Lithops Fork." <https://github.com/faasm/lithops>, 2020.
- [98] LDS Group, "Faasm Python Support." <https://github.com/faasm/python>, 2020.
- [99] LDS Group, "Faasm Python Documentation." <https://github.com/faasm/faasm/blob/main/docs/source/python.md>, 2020.
- [100] LDS Group, "Faasm C/C++ Support." <https://github.com/faasm/cpp>, 2020.
- [101] P. G. Lopez, A. Arjona, J. Sampe, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," in Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems, DEBS 2020, ACM, 2020.
- [102] Lithops, "Lithops applications." <https://github.com/lithops-cloud/applications>, 2022.