HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D3.2 Serverless Compute Engine Design and Prototypes

Due date of deliverable: 30-06-2020
Actual submission date: 04-08-2020

Start date of project: 01-01-2019 Duration: 36 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v1.0
Number of pages	45
WP/Task related to this document	WP3 / T3.1, T3.2, T3.3, T3.4
WP/Task responsible	IBM
Leader	Gil Vernik, David Breitgand (IBM)
Technical Manager	Peter Pietzuch (Imperial)
Quality Manager	Josep Sampé (URV)
Author(s)	Gil Vernik, David Breitgand, Omer Belhasin, Josep Sampe, Gerard París, Pedro García, Rut Palmero
Partner(s) Contributing	IBM, URV, ATOS
Document ID	CloudButton_D3.2_Public.pdf
Abstract	This document describes our progress with the architecture design and initial prototypical implementation of the CloudButton platform for data intensive computations.
Keywords	FaaS, serverless, Kubernetes, hybrid cloud, workflow orchestration, data intensive computations, SLA monitoring.

History of changes

Version	Date	Author	Summary of changes
0.1	18-06-2020	Gil Vernik (IBM)	Table of Contents
0.2	19-06-2020	Rut Palmero (ATOS)	SLA monitoring and management
0.3	22-06-2020	Josep Sampé (URV)	Executors
0.4	02-07-2020	Gerard París (URV)	Geospatial use case evaluation
0.5	13-07-2020	Gerard París (URV), Pedro García (URV)	Section on Triggerflow and cost-efficiency evaluation
0.6	16-07-2020	David Breitgand (IBM)	Section on serverless workflows (ArgoNotes)
1.0	04-08-2020	Gil Vernik (IBM)	Final version.

Table of Contents

List of Abbreviations and Acronyms

ADF	Azure Durable Functions
API	Application programming interface
ASF	Amazon Step Functions
CD	Continued Development
CLI	Command-line interface
CNCF	Cloud Native Computing Foundation
COS	Cloud Object Storage
CPU	Central Processing Unit
CRC	Custom Resource Controller
CRD	Custom Resource Definition
DAG	Directed Acyclic Graph
DSL	Domain Specific Lanaguage
ETL	Extract, Transform, Load
FaaS	Function as a Service
FDR	False Discovery Rate
GPU	Graphics Processor Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ICT	Information and Communication Technology
JSON	JavaScript Object Notation
K8s	Kubernetes
PaaS	Platform as a Service
QoS	Quality of Service
REST	Representational State Transfer
SDK	Software Development Kit
SLA	Service Layer Agreement
SLO	Service Layer Objective
SOTA	State of the art
UI	User interface
VM	Virtual Machine
YAML	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 initial prototypical implementation of the CloudButton platform for data intensive computations. The CloudButton platform comprises five main parts:

- **CloudButton Toolkit (we also term this component "Execution Engine")**: 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 CloudButton Core;

- **CloudButton Core (we also term this component "Compute Engine")**: 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 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 CloudButton Toolkit and CloudButton Core (e.g., Cloud Object Storage (COS))

- **Caching Service**: this is another independently deployed component providing caching services to the CloudButton Core (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 CloudButton Toolkit and CloudButton Core and their interplay. The architecture follows a cloud-native microservices based approach.

In Section 4 we describe our initial prototypical implementation illustrated by its application to one of the use cases. We take a step-wise agile approach for implementing the overall architecture. Hence our current initial prototype is a first Minimal Viable Product (MVP) that allows us to start experimentation, bottleneck analysis and accumulation of the hand-on experience with the use cases. The initial prototype demonstrates how the public cloud services can be leveraged in the proposed architecture. Specifically, the prototype uses IBM Cloud Functions [2] as a serverless backend framework, and IBM Cloud Storage (COS) [3] as storage backend. The resulting prototype is made available publicly as PyWren over IBM Cloud Functions and IBM Cloud Object Storage project [4].

2 Motivation and Background

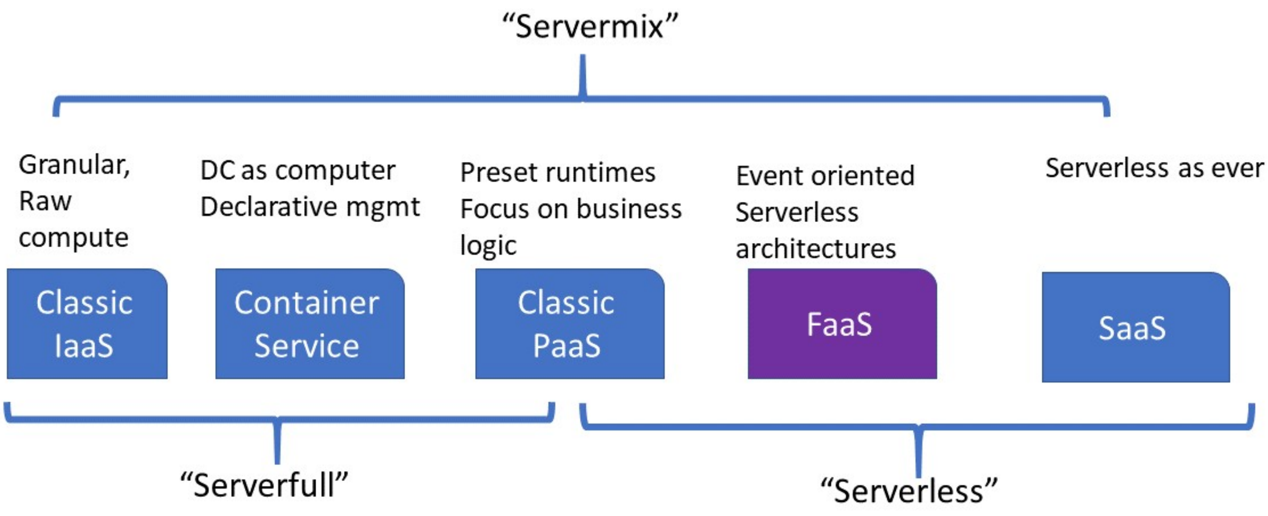


Figure 1: Serverless Taxonomy

2.1 Serverless Computing Overview

The serverless programming model, also known as Function-as-a-Service (FaaS[1]) has gained considerable momentum since its introduction in 2014 [5]. 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)[2]

[1]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.

[2]FaaS extensions, such as AWS Step Functions [6] and Azure Durable Functions [7] 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[3]
- 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.

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 [2] and Apache OpenWhisk community resources [13] 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 [14], 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.

[3]The orchestrators are typically external to the FaaS frameworks. Apache Composer [8] 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 [9], Kubeflow [10], Argo Flows [11], Fission Workflows [12]. 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 [15] 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[4].

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 [16], 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 and enterprise data scattered among them. Moreover, nowadays, most enterprises use multi-cloud strategy for their ICT [17] 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 [18], 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 [19, 20, 21] with CNCF's Kubernetes (K8s) [21] 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-

[4]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 [11] and Kubeflow [10] 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) [22] 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 "plug-ins" 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 [23].

Event-based triggering has also been extensively employed in the past to provide reactive coordination of distributed systems [24, 25]. Event-based mechanisms and triggers have also been extensively used [26, 27, 28, 29] 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 [30], they propose to use synchronous aggregation triggers to coordinate massively parallel data processing jobs.

An interesting related work is [29]. 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 [31] 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 [32] 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 [33, 34, 35, 36, 37, 38]. 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 applicabillity 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 [39, 40] 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.

4 Design and Implementation

Find below a figure showing CloudButton High Level Architecture, from D3.1[41].

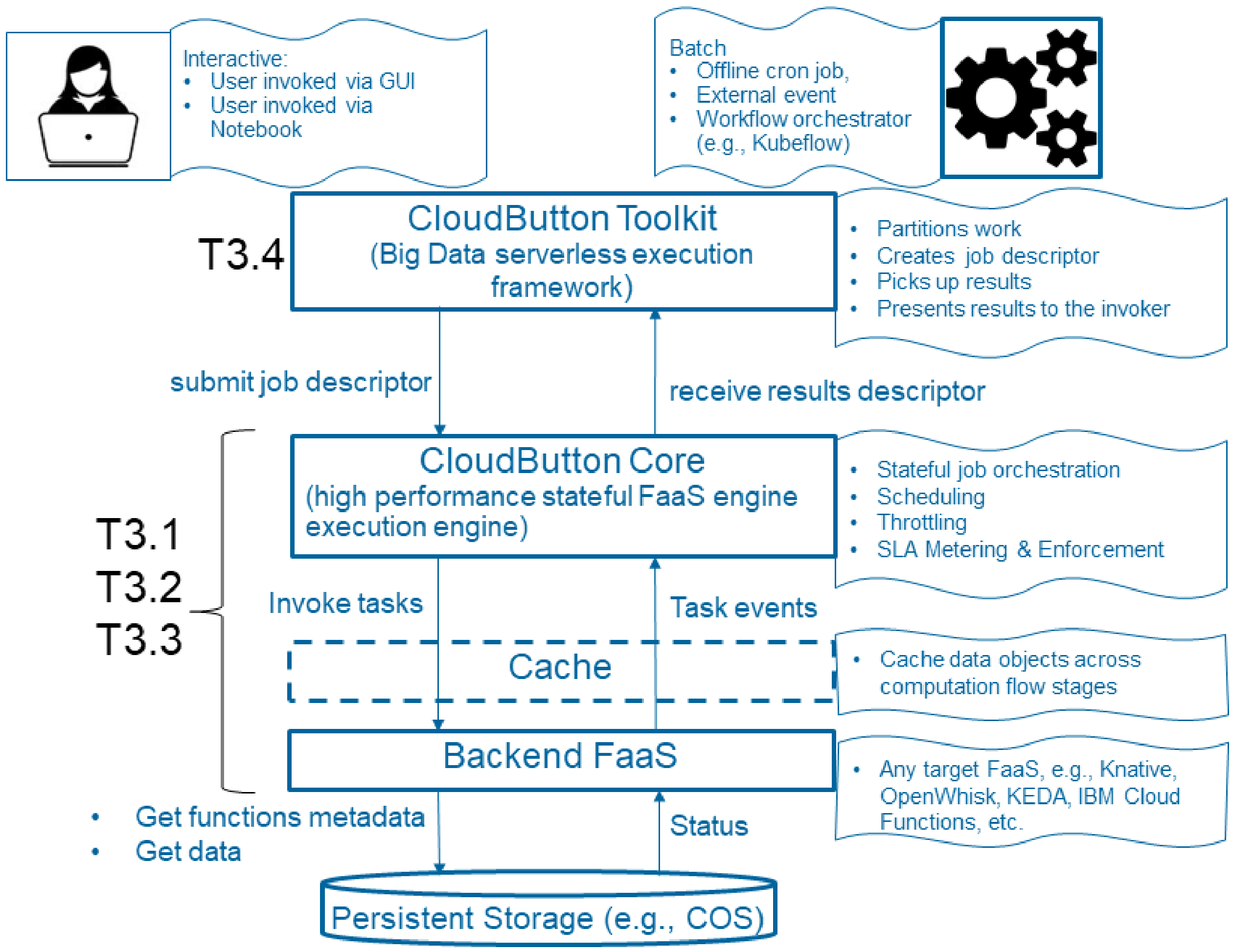

Interactive:
User invoked via GUI
User invoked via Notebook
Batch
Offline cron job,
External event
Workflow orchestrator (e.g., Kubeflow)
T3.4
CloudButton Toolkit
(Big Data serverless execution framework)
Partitions work
Creates job descriptor
Picks up results
Presents results to the invoker
submit job descriptor
receive results descriptor
T3.1
T3.2
T3.3
CloudButton Core
(high performance stateful FaaS engine execution engine)
Stateful job orchestration
Scheduling
Throttling
SLA Metering & Enforcement
Invoke tasks
Task events
Cache
Cache data objects across computation flow stages
Backend FaaS
Any target FaaS, e.g., Knative, OpenWhisk, KEDA, IBM Cloud Functions, etc.
Get functions metadata
Get data
Status
Persistent Storage (e.g., COS)

Figure 2: CloudButton High Level Architecture.

4.1 CloudButton Toolkit general architecture and design

The CloudButton toolkit is a multicloud framework that enables the transparent execution of unmodified, regular Python code against disaggregated cloud resources. With the CloudButton toolkit, there is no new API to learn. It provides the same API as Python's standard `multiprocessing` [42] and `concurrent.futures` [43] and PyWren-IBM [4] 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. The CloudButton toolkit 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 the CloudButton Toolkit was designed to work with IBM Cloud Functions [2] 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, the cloudbutton toolkit 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, the cloudbutton toolkit 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: Cloudbutton toolkit backends

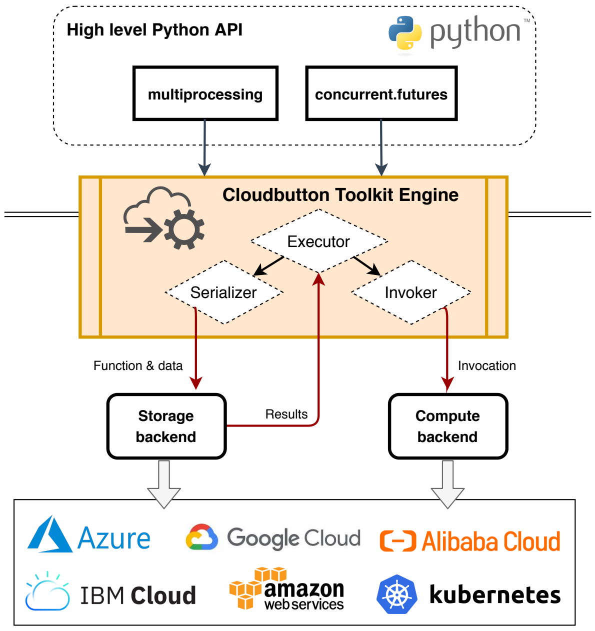


Figure 3: High level representation of the Cloudbutton toolkit

The high-level architecture is depicted in Figure 3. Internally, the Cloudbutton toolkit 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 in the serverless compute backend. Instead, they are stored in the storage backend. Then, the Cloudbutton toolkit 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, the Cloudbutton toolkit 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.

The Cloudbutton toolkit 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, the Cloudbutton toolkit 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 CloudButton Toolkit 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.

Our current CloudButton toolkit implementation contains an initial Partitioner implementation, 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.

4.2 Multiple APIs

CloudButton toolkit exposes different APIs that can be used based on the user requirements. In D5.1 we presented a first API definition based on map-reduce. Now, the flexibility of the Cloudbutton toolkit is substantially increased by mimicking the Python's `multiprocessing` API and components.

The Map-Reduce API is the basic API used by the Cloudbutton Toolkit, and it integrates the basic, low-level methods to spawn functions in the cloud. The primary object in the Map-Reduce API of the Cloudbutton toolkit is the **FunctionExecutor**, that provides the following API methods: `call_async`, `map`, `map_reduce`, `wait` and `get_result`.

As an addition to the previous API, CloudButton toolkit now supports most of the Python `multiprocessing` abstractions, such as the Process, Pool, Queue, Pipe, Lock, Semaphore, Event, Barrier, and also remote memory in Manager objects. This means that most of the current multiprocessing-based applications can be moved and scaled to the cloud by only changing the import statement of the script.

Both APIs are further described in D5.2 *CloudButton Prototype of Abstractions, Fault-tolerance and Porting Tools* [45].

4.3 Executors

4.3.1 Local executor

The local executor is the basic executor in the Cloudbutton toolkit, and it allows to run applications by using local processes. 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.

4.3.2 Docker executor

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.

In the cloudbutton toolkit, the deocker executor 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. A 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.

4.3.3 Apache OpenWhisk executor

Apache OpenWhisk 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 the cloudbutton toolkit, the Apache OpenWhisk executor allows to execute functions in any vanilla OpenWhisk installation by comunicating 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.

4.3.4 IBM Cloud Function executor

The 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.

4.3.5 Knative executor

Knative 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 the cloudbutton toolkit, 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 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 the CloudButton toolkit. The process usually not visible to the end user and he is not aware of this type of data. As example CloudButton Toolkit generates a single JSON file 4 per each invocation. This file includes invocation status, completion timestamp, and other metadata that is needed by CloudButton toolkit. Once all invocation completed, Cloudbutton toolkit reads all generated files, obtain statuses of each invocation and decides what response return to the user.

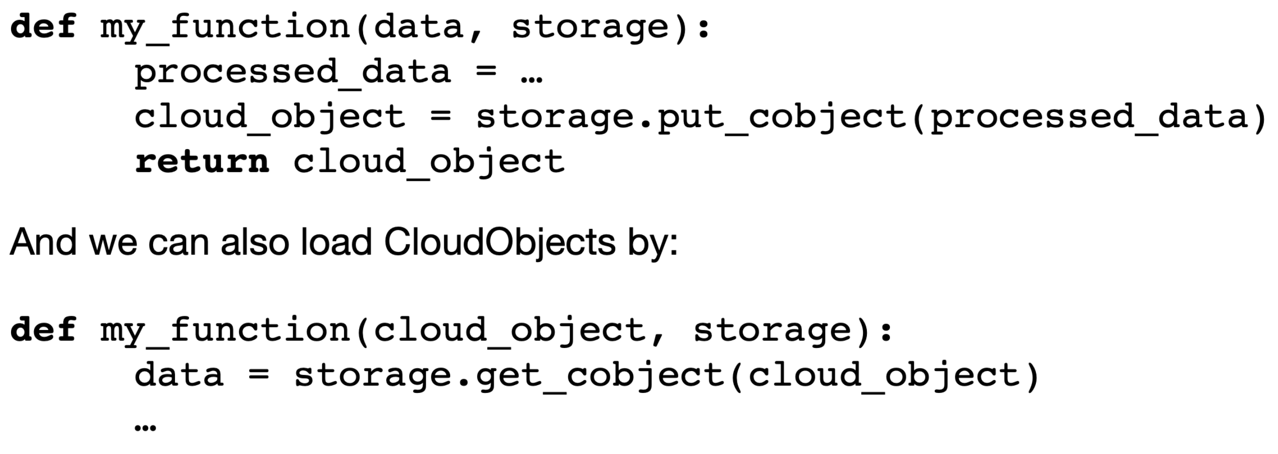
```
{"exception": false, "host_submit_tstamp": 1592715321.0335932, "start_tstamp":
    1592715331.4192479, "python_version": "3.6.9", "call_id": "00000", "job_id":
    "M000", "executor_id": "0d76ac/0", "activation_id": "0b566331cc2100000",
    "type": "__end__", "function_download_time": 0.01882219, "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 4: 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.

4.5 CloudObject

We noticed that data sharing in variery 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 simple store CloudObjects as follows: By this approach, the CloudObject



```
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 5: CloudObject

can be considered as a pointer to the stored data, without the necessity of managing its storage path exactly, and the CloudButton Toolkit 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.

4.6 Sorting and shuffle

As part of the serverless limitations challenges to implement a serverless pipeline for metabolites detection task, we encountered in a challenge of a simple operation - sorting data. The whole preprocess for the main searching process is composed by 3 sorting logics points. Dealing this challenge naively (load the whole data and sort it in the same process) isn't matched for a serverless approach because of the duration and memory limitations of each worker. moreover, we also can't use a naive sorting logic in the local machine cause there could be variety of datas that can't be fitted into local memory. Therefore, external chunks-wise sorting algorithms are required to deal with this challenge like the well known "merge-sort" and "quick-sort" or even dedicated memory-wise algorithms that sort carefully without reaching beyond a memory consumption limit. However, using these algorithms sequentially in the same process can take a lot of time and if we apply parallelism, the processes will still be limited by the available memory, compute resources and network performance. Utilising serverless capabilities, we implemented a sorting algorithm (for data located in a storage service) that stores an ordered sequence of data partitions such that each partition is associated with unique ranged values and its size is memory bounded. The framework is built to support in massive parallelism appliance under memory and time constrains and also to be cost-efficient as possible regarding serverless limitations by fitting each serverless job with maximum memory available for its function. Diving into the framework implementation, it contains a preprocess part and two main stages. Firstly, the algorithm predefines data values ranges by sampling the data, each values range defines a data partition that will include these values - let "M" be the number of ranges. The algorithm ensures that the ranged values will be estimated memory bounded by the sampled distribution. Secondly, the algorithm launches parallel serverless jobs to partition memory bounded data chunks - let "N" be the number of chunks. Each job divides a given data chunk into data sub- segments such that each sub-segment includes data values out of several predefined ranges (up to the maximum possible memory size which can be fitted into the serverless job) - Let "C" be the number of sub-segments of each job. At last, the algorithm launches another C parallel serverless jobs to merge sub-segments of different chunks that includes values out of same ranges. Each job also divides the resulted merged segment

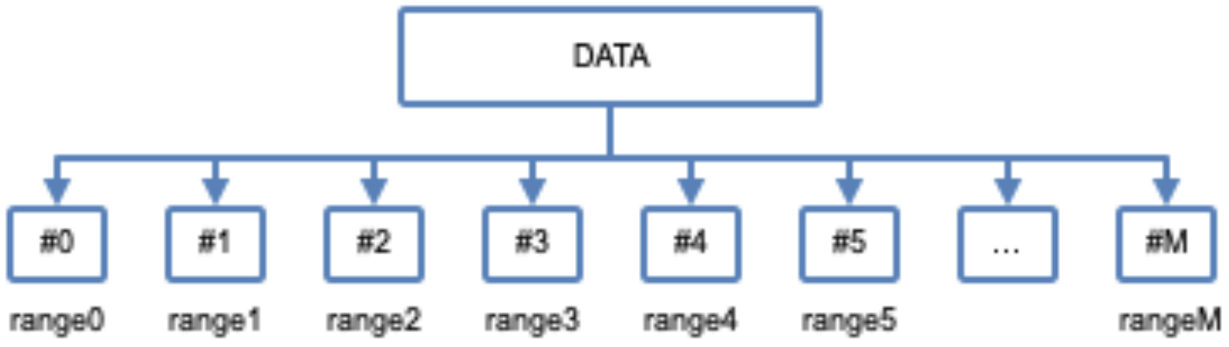


Figure 6: Data partitioning into ranges

into sub-segments again such that the total number of sub-segments out of all parallel processes will be M. The algorithm correctness comes from the fact that each parallel job (first and second stage) consumes maximum allowed memory so it will use minimum required serverless workers number: N parallel jobs and then C parallel jobs (cost-efficient). In addition, it is guaranteed that each job won't reach beyond a memory limit: in the first stage the memory consumption depends on each memory bounded data chunks and in the second stage, it is guaranteed that each N sub- segments to merge are memory bounded in estimate due to the sampling part. This framework is also designed to use the storage service efficiently as possible. During this processes, N*C intermediate storage objects are created because serverless jobs are isolated from each other. The algorithm tunes N and C such that it will be the minimum number required for each serverless job to work. By this way, it reduces significantly the number of intermediate storage objects and also the number of storage parallel requests: N parallel storage requests in the first stage, and N*C parallel storage requests in the second stage.

4.7 Integration with 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 memory only. This makes Infinspan as a perfect candidate for storing temporary data. We designed CloudButton Toolkit 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 CloudButton Toolkit to access additional storage backend, all what is required is to implement internal-storage interface and configure CloudButton Toolkit 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 CloudButton Toolkit

```
#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 CloudButton Toolkit 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.

4.8 Serverless Workflows

4.8.1 ArgoNotes

CloudButton caters for a typical scenario, when a data scientist needs to execute a lengthy analytical pipeline in an unattended mode. Usually, a data scientist uses a notebook, such as Jupiter notebook to perform the analytic computations. The control flow of a notebook is usually sequential, even though it is possible to create more complex dependencies among the cells of a notebook. After a data scientist debugs and fine tunes her computations (this fine tuning might include the level of parallelism, the timeout values, the number of retries, etc.), she might wish to save her notebook and possibly share it with the rest of the team. The latter part is very important, because different data sets are being produced by different members of the team (and perhaps even by different teams) and computations might involve multiple data sets. For example a satellite and a LIDAR data sets can be independently processed and then the results can be combined to produce an annotated map, which in turn would be treated as a new data set for some additional analytics. We refer to this event-driven process as big data serverless analytic *pipelines*.

One notbook can be imported into another notebook. In Jupiter Notebooks, a notebook can be imported as a module [5]. Therefore a general structure of the computation is a Directed Acyclic Graph (DAG) rather than a sequence. In the example above, a Sattelite data set and a LiDAR data sets can be processed in parallel, because there are no dependencies between them. The annotated map computation can only start after these two tasks (regarded as dependencies) are completed.

Once a notebook is debugged and fine tuned it can be reused to handle similar data, when a new instance of that data becomes available. Moreover, since the notebook is already prepared, no human attendance is required to execute it. We differentiate between two basic situations that might trigger a pipeline execution:

- Manual submission event of a notebook to CloudButton: this is equivalent of the "push-to-the-cloud" concept that PyWren-IBM execution framework provides for a single python function, generalized to a multi-step, inter-dependent computations;

- An event-driven automated execution of the pipeline: a data life cycle management event, such as a data set arrival to a data store, a data set departure, update, etc. Usually, these life cycle management events are associated with an overarching *data lake* architecture, that sets up a framework of the data scientists' work. Serverless technology is being used widely to implement such architectures[6]. An implementation of the data lake and its associated interfaces is out of the scope for this project. However, it is important to keep consider this as a background for possible industrial exploitation of the CloudButton results, since the CloudButtons ArgoNotes mechanism can be seamlessly connected to a data lake architecture, such as IBM Cloud Data Lake[7] and we plan to explore this opportunity as part of our exploitation road map.

To achieve portability of our serverless pipelines orchestration mechanism, ArgoNotes, we use Argo Workflows[8], a CNCF[9] hosted project and Argo Events[10], a CNCF landscape project. The Argo Workflow project is container native Kubernetes workflow engine. It follows the Kubernetes Operator software design pattern. The Operator comprises a Custom Resource Definition (CRD), which is a schema for the yaml documents, called Custom Resources (CRs) and a Controller. When CR is applied to the Kubernetes API Server, a Controller, which is watching workflow CRs executes as reconciliation cycle, in which it continously tries to bring the observed state of the workflow to it its

[5]https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Importing%20Notebooks.html
[6]https://aws.amazon.com/training/course-descriptions/serverless-datalake/
[7]https://www.ibm.com/cloud/architecture/architectures/cloud-data-lake
[8]https://github.com/argoproj/argo
[9]https://www.cncf.io/
[10]https://github.com/argoproj/argo-events

desired state (i.e., to execute all its steps according to dependencies specified in the CR). Each step in Argo Workflow is a pod executing in Kubernetes. Argo Workflow offers a very rich set of features that allows to specify arbitrary complex flows.

Kubernetes has become a de facto standard for container orchestration, with all major cloud vendors offering it as a service. Also, Kubernetes has became a platform of choice for the private cloud. Being Kubernetes native, Argo Workflows and Argo Events offer portability out of the box.

Argo Events is Kubernetes native event dependency resoulution system that is designed to work seamlessly with Argo workflows. Argo Events offers a rich collection of event sources. These sources are termed *Gateways*. a Gateway receives an event and passed it to another entity called *Sensor*. The sensor then triggers any Kubernetes resource (including Argo Workflow), if an event dependency is satisfied. Both Gateway and Sensor instances are CRs managed by their respective Gateway and Sensors controllers. The CRs are defined according to the Gateway and Sensor CRD respectively.

Listing 1 shows the Argo Workflow CR (a yaml document that describes the workflow), which is obtained automatically from the EMBL use case notebook. The workflow CR definition comprises two main constructs: steps and templates. Steps (and dependencies among them) correspond to the computational tasks. The templates describe the executable images that back these tasks.

Each step of an Argo workflow is a pod in Kubernetes. Therefore it is being possible to obtain metrics related to each step execution using native metering, monitoring, and visualisation tools of Kubernetes, such as Prometheus and Grafana. The execution of the workflows is being monitored by the SLA Monitoring and Management tools (See Section 5) that use these native Kubernetes monitoring and metering instrumentation.

The data scientist is never exposed to this level of details. Rather, we've implemented a custom exporter[11] to automatically create an Argo Workflow from the Jupiter Notebook.

Our goal is to relieve the data scientist from learning new programming paradigms and minimize the use of auxiliary tools that she needs to use in order to execute her computations. In the true spirit of serverless computing that promises to relieve a developer from the low level details of the execution environment, allowing her to focus only on the business logic of her code.

To that end we utilize a standard tagging mechanism of Jupiter Notebooks. The tagging allows to allocate the notebook cells to steps and templates. Tags are part of the general metadata mechanism in Jupiter Notebooks. Using metadata, dependencies among cells (e.g., cells that import different notebooks) to specify a DAG, can be specified. A simple example of tagging is shown in Figure 7. This tagging effectively creates the Argo Workflow CR shown in Listing 1. There are two templates and two steps in this workflow: prepare and process. But they could have been named differently as well. The cells with the same name are allocated to the same templates. In the EMBL use case, the templates are created out of Python code, because the notebook comprises PyWren computations only. However, in a more general case, the workflow can comprise arbitrary steps that can be serverless or serverful, massively parallel or sequential. We term this model Servermix [46].

Figure 8 shows Argo GUI that allow to trace a workflow progress. The goal of ArgoNotes is to take all the boilerplate code related to orchestration out of the way of the data scientist. Therefore, as explained above, a human attendance is not required to execute a serverless pipeline. Yet, a data scientist or an operator can check the progress of the flow at any time. As shown in Figure 9, a detailed log is available for every step. This information is retained by Argo also for the flows that have terminated. It can be deleted when a flow is purged from Kubernetes.

```

apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: dag-steps-
spec:
  entrypoint: steps
  templates:
  - dag:
```

[11] https://nbconvert.readthedocs.io/en/latest/externa_exporters.html

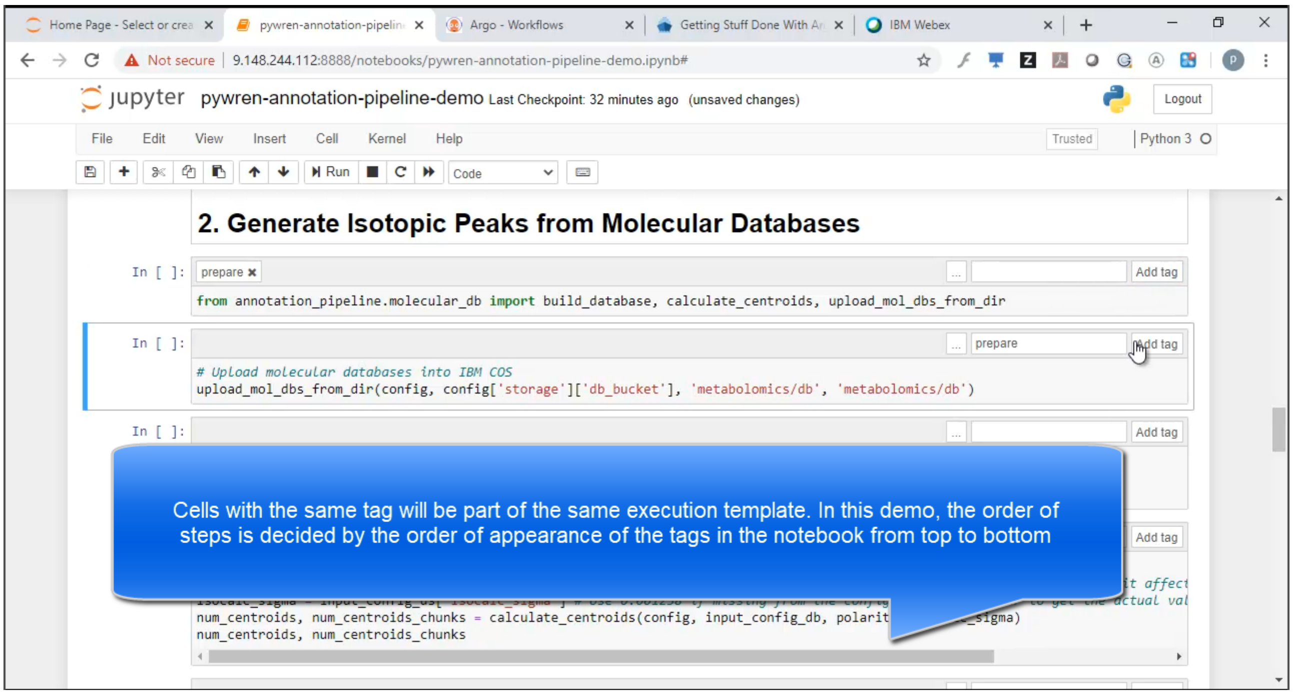
```
  tasks:
  - arguments: {}
    dependencies: []
    name: prepare
    template: prepare
  - arguments: {}
    dependencies:
    - prepare
    name: process
    template: process
 name: steps
- inputs:
  parameters:
  - name: data
    value: '[]'
 name: prepare
 script:
  command:
  - python
  env:
  - name: PYWREN_CONFIG
    value: |-
     {"ibm": {"iam_api_key": "7hlUkZvvZMDkx85h5ek9tVysE6XoGsKj7KT9KgUVLTNv"}, "
       ibm_cf": {"endpoint": "https://us-east.functions.cloud.ibm.com", "
       namespace": "washns", "namespace_id": "2f5f6183-fac7-4f57-89b1-099
       c9132702d"}, "ibm_cos": {"endpoint": "https://s3.us-east.cloud-object-
       storage.appdomain.cloud", "private_endpoint": "https://s3.private.us-
       east.cloud-object-storage.appdomain.cloud", "access_key": "
       c98b17b8084b4cd9a9a1638a8992f9a5", "secret_key": "69
       a86f9be05f5993195270e015aabe8cf78ce4ba7fb04cb5"}, "pywren": {"
       storage_bucket": "embl-bucket", "runtime": "kpavel/my-runtime:cf2", "
       include_modules": ["annotation_pipeline"], "workers": 256}, "storage":
       {"ds_bucket": "kpmybucket42", "db_bucket": "kpmybucket42", "
       output_bucket": "kpmybucket42"}}
  image: artifactory.haifa.ibm.com:5130/kpavel/pywren-annotation-pipeline:0.1
  name: prepare
  source: |-
    # Display IBM PyWren version

    import pywren_ibm_cloud as pywren

    pywren.__version__

    import json

    #config = json.load(open('config.json'))

    import os

    config = json.loads(os.environ.get('PYWREN_CONFIG', ''))
    import json

    #input_config = json.load(open('metabolomics/input_config_small.json'))

    #input_config = json.load(open('metabolomics/input_config_big.json'))

    #input_config = json.load(open('metabolomics/input_config_huge.json'))

    #input_config = json.load(open('metabolomics/input_config_huge2.json'))

    #input_config = json.load(open('metabolomics/input_config_huge3.json'))

    input_config = json.load(open('input_config.json'))

```

```
        input_config_ds = input_config['dataset']

        input_config_db = input_config['molecular_db']

        # Please note that some input_configs specify a `mol_db6`, which is not yet publicly available.

        # This will remove it from the config to prevent later errors. Results will still be generated for other databases.

        input_config_db['databases'] = [db for db in input_config_db['databases'] if 'mol_db6' not in db]
        from annotation_pipeline.molecular_db import build_database, calculate_centroids, upload_mol_dbs_from_dir
        # Upload molecular databases into IBM COS

        upload_mol_dbs_from_dir(config, config['storage']['db_bucket'], 'metabolomics/db', 'metabolomics/db')
        # Generate formulas dataframes into IBM COS

        num_formulas, num_formulas_chunks = build_database(config, input_config_db)

        num_formulas, num_formulas_chunks
        # Generate isotopic peaks dataframes into IBM COS

        polarity = input_config_ds['polarity'] # Use '+' if missing from the config, but it's better to get the actual value as it affects the results

        isocalc_sigma = input_config_ds['isocalc_sigma'] # Use 0.001238 if missing from the config, but it's better to get the actual value as it affects the results

        num_centroids, num_centroids_chunks = calculate_centroids(config, input_config_db, polarity, isocalc_sigma)

        num_centroids, num_centroids_chunks
  - inputs:
      parameters:
      - name: data
        value: '[]'
    name: process
    script:
      command:
      - python
      env:
      - name: PYWREN_CONFIG
        value: |-
          {"ibm": {"iam_api_key": "7hlUkZvvZMDkx85h5ek9tVysE6XoGsKj7KT9KgUVLTNv"}, "ibm_cf": {"endpoint": "https://us-east.functions.cloud.ibm.com", "namespace": "washns", "namespace_id": "2f5f6183-fac7-4f57-89b1-099c9132702d"}, "ibm_cos": {"endpoint": "https://s3.us-east.cloud-object-storage.appdomain.cloud", "private_endpoint": "https://s3.private.us-east.cloud-object-storage.appdomain.cloud", "access_key": "redacted", "secret_key": "redacted"}, "pywren": {"storage_bucket": "embl-bucket", "runtime": "kpavel/my-runtime:cf2", "include_modules": ["annotation_pipeline"], "workers": 256}, "storage": {"ds_bucket": "kpmybucket42", "db_bucket": "kpmybucket42", "output_bucket": "kpmybucket42"}}
      image: artifactory.haifa.ibm.com:5130/kpavel/pywren-annotation-pipeline:0.1
      name: process
      source: |-
        # Display IBM PyWren version

        import pywren_ibm_cloud as pywren
```

```

pywren.__version__

import json

#config = json.load(open('config.json'))

import os

config = json.loads(os.environ.get('PYWREN_CONFIG', ''))
import json

#input_config = json.load(open('metabolomics/input_config_small.json'))

#input_config = json.load(open('metabolomics/input_config_big.json'))

#input_config = json.load(open('metabolomics/input_config_huge.json'))

#input_config = json.load(open('metabolomics/input_config_huge2.json'))

#input_config = json.load(open('metabolomics/input_config_huge3.json'))

input_config = json.load(open('input_config.json'))

input_config_ds = input_config['dataset']

input_config_db = input_config['molecular_db']

# Please note that some input_configs specify a `mol_db6`, which is not yet
    publicly available.

# This will remove it from the config to prevent later errors. Results will
    still be generated for other databases.

input_config_db['databases'] = [db for db in input_config_db['databases'] if
    'mol_db6' not in db]
from annotation_pipeline.pipeline import Pipeline

pipeline = Pipeline(config, input_config)
# Load the dataset's parser

pipeline.load_ds()
# Parse dataset chunks into IBM COS

pipeline.split_ds()
# Sort dataset chunks to ordered dataset segments

pipeline.segment_ds()
# Sort database chunks to ordered database segments

pipeline.segment_centroids()
# Annotate the molecular database over the dataset by creating images into
    IBM COS

pipeline.annotate()
# Discover expected false annotations by FDR (False-Discovery-Rate)

pipeline.run_fdr()
# Display statistic results

results_df = pipeline.get_results()

results_df
```

```
status: {}
```

Listing 1: EMBL Use Case as Argo Workflow

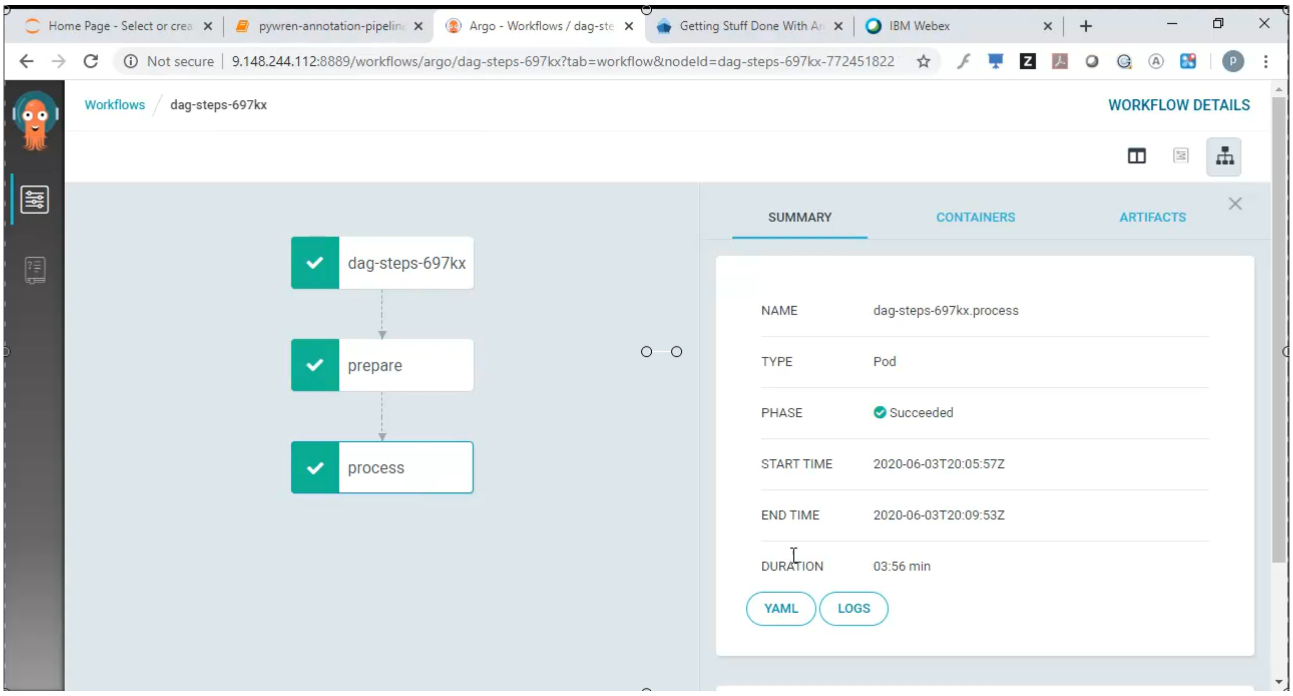


2. Generate Isotopic Peaks from Molecular Databases

```
In [ ]:  prepare
from annotation_pipeline.molecular_db import build_database, calculate_centroids, upload_mol_dbs_from_dir

In [ ]:  prepare
# Upload molecular databases into IBM COS
upload_mol_dbs_from_dir(config, config['storage']['db_bucket'], 'metabolomics/db', 'metabolomics/db')

In [ ]:

num_centroids, num_centroids_chunks = calculate_centroids(config, input_config_db, polarit
num_centroids, num_centroids_chunks
```

pywren-annotation-pipeline-demo Last Checkpoint: 32 minutes ago (unsaved changes)
9.148.244.112:8888/notebooks/pywren-annotation-pipeline-demo.ipynb#
Cells with the same tag will be part of the same execution template. In this demo, the order of steps is decided by the order of appearance of the tags in the notebook from top to bottom
Add tag
Python 3

Figure 7: Tagging of a Jupiter Notebook to automatically obtain an Argo Workflow definition

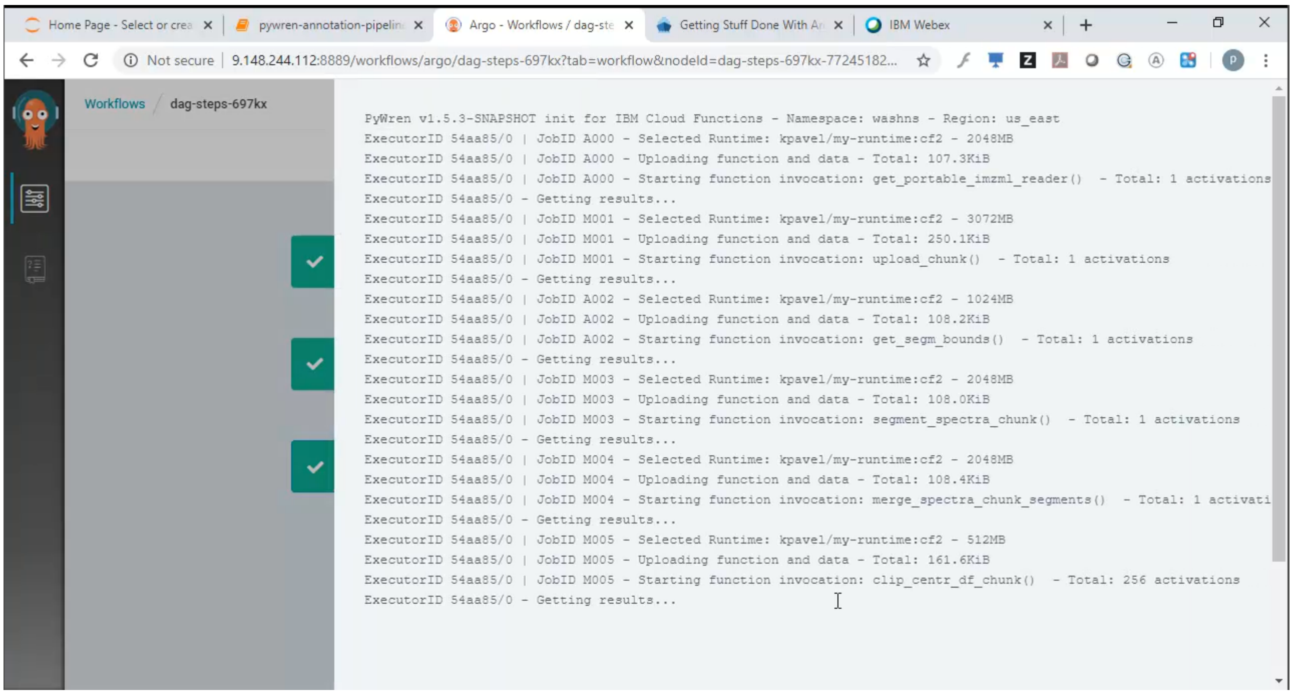


9.148.244.112:8889/workflows/argo/dag-steps-697kx?tab=workflow&nodeId=dag-steps-697kx-772451822
Workflows / dag-steps-697kx
WORKFLOW DETAILS
dag-steps-697kx
prepare
process
SUMMARY
CONTAINERS
ARTIFACTS
YAML
LOGS

NAME	dag-steps-697kx.process
TYPE	Pod
PHASE	Succeeded
START TIME	2020-06-03T20:05:57Z
END TIME	2020-06-03T20:09:53Z
DURATION	03:56 min

Figure 8: Using Argo GUI the progress of a flow can be inspected at any time (also for the completed flows)

Figure 10 depicts the overall ArgoNotes architecture. A data scientist commits a pipeline to a ArgoNotes (Step 1), when she is happy with her notebook and wishes to reuse and share it. The notebook is automatically translated by the custom external exporter into Argo artifacts that comprise


Home Page - Select or crea
pywren-annotation-pipelin
Argo - Workflows / dag-ste
Getting Stuff Done With A
IBM Webex
Not secure | 9.148.244.112:8889/workflows/argo/dag-steps-697kx?tab=workflow&nodeId=dag-steps-697kx-77245182...
Workflows / dag-steps-697kx

```
PyWren v1.5.3-SNAPSHOT init for IBM Cloud Functions - Namespace: washns - Region: us_east
ExecutorID 54aa85/0 | JobID A000 - Selected Runtime: kpavel/my-runtime:cf2 - 2048MB
ExecutorID 54aa85/0 | JobID A000 - Uploading function and data - Total: 107.3KiB
ExecutorID 54aa85/0 | JobID A000 - Starting function invocation: get_portable_imzml_reader()  - Total: 1 activations
ExecutorID 54aa85/0 - Getting results...
ExecutorID 54aa85/0 | JobID M001 - Selected Runtime: kpavel/my-runtime:cf2 - 3072MB
ExecutorID 54aa85/0 | JobID M001 - Uploading function and data - Total: 250.1KiB
ExecutorID 54aa85/0 | JobID M001 - Starting function invocation: upload_chunk()  - Total: 1 activations
ExecutorID 54aa85/0 - Getting results...
ExecutorID 54aa85/0 | JobID A002 - Selected Runtime: kpavel/my-runtime:cf2 - 1024MB
ExecutorID 54aa85/0 | JobID A002 - Uploading function and data - Total: 108.2KiB
ExecutorID 54aa85/0 | JobID A002 - Starting function invocation: get_segm_bounds()  - Total: 1 activations
ExecutorID 54aa85/0 - Getting results...
ExecutorID 54aa85/0 | JobID M003 - Selected Runtime: kpavel/my-runtime:cf2 - 2048MB
ExecutorID 54aa85/0 | JobID M003 - Uploading function and data - Total: 108.0KiB
ExecutorID 54aa85/0 | JobID M003 - Starting function invocation: segment_spectra_chunk()  - Total: 1 activations
ExecutorID 54aa85/0 - Getting results...
ExecutorID 54aa85/0 | JobID M004 - Selected Runtime: kpavel/my-runtime:cf2 - 2048MB
ExecutorID 54aa85/0 | JobID M004 - Uploading function and data - Total: 108.4KiB
ExecutorID 54aa85/0 | JobID M004 - Starting function invocation: merge_spectra_chunk_segments()  - Total: 1 activati
ExecutorID 54aa85/0 - Getting results...
ExecutorID 54aa85/0 | JobID M005 - Selected Runtime: kpavel/my-runtime:cf2 - 512MB
ExecutorID 54aa85/0 | JobID M005 - Uploading function and data - Total: 161.6KiB
ExecutorID 54aa85/0 | JobID M005 - Starting function invocation: clip_centr_df_chunk()  - Total: 256 activations
ExecutorID 54aa85/0 - Getting results...
```

Figure 9: Detailed logs of the flow can be expected (also for the completed flows)

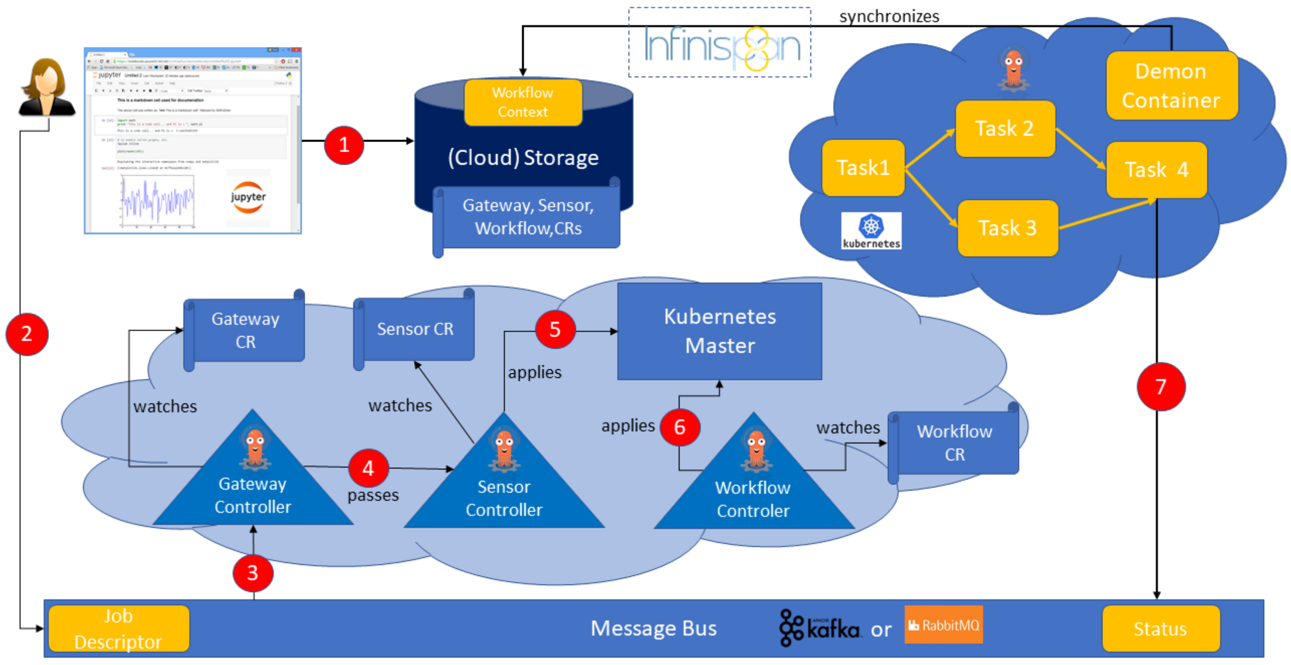


Figure 10: ArgoNotes High Level Architecture

three Custom Resource definitions: Gateway CR, Sensor CR, and Workflow CR. Gateway, Sensor, and Workflow controllers are preinstalled on Kubernetes that can be located anywhere. These three controllers represent the control plane of ArgoNotes serverless pipelines.

At some point (not shown on the figure) the pipeline should be bootstrapped. More specifically, CRs for the Gateway and Sensor should be applied. The Gateway and Sensor controllers watch these CRs and at this point, the pipeline is ready for operation. At some point in time (Step 2) either the data scientist or an automation create a Job Descriptor on a message bus that represents an event that should be handled by the pipeline. This event is caught by the Gateway controller (Step 3) that passes it to the Sensor controller (Step 4), which (in accordance with the event dependency encoded in a CR that it watches), applies a Workflow CR, i.e., triggers it, agains the Kubernetes Master (Step 5). This creates a Workflow CR instance that triggers a reconciliation cycle in the Workflow Controller that watches Workflow CR instances. At this point, the Workflow Controller starts a workflow on Kubernetes (the Kubernetes cluster where the actual execution of a workflow happens, does not have to be the same one as where the control plane of the workflow runs). Steps in the flow are pods that can execute any computation. In particular, they can execute PyWren computations against IBM Cloud Functions as the executor backend or against KNative executor backend in the same or a different Kubernetes cluster. Throughut the execution of the workflow, its context (i.e., its state is being checkpointed on a non-volatile storage and possibly cached in main memory). The last task of the workflow posts the status of the workflow execution on the message bus. The data scientist can inspect the results at any time asynchronously using Job Descriptor to read from an appropriate topic.

As we go to press the main building blocks of ArgoNotes stateful serverless orchestrator powering complex big data analytics have been implemented. An integration of these basic blocks and tighter integration with the rest of CloudButtin components will be performed in the next reporting period.

4.8.2 Triggerflow

In the context of the CloudButton project, we have created Triggerflow [47], 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, termination 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 11 an overall diagram of the Triggerflow Architecture. The Trigger service follows an extensible Event-ConditionAction 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 11 also shows the basic API exposed by TriggerFlow: createWorkflow initializes the context for a given workflow, addTrigger adds 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.

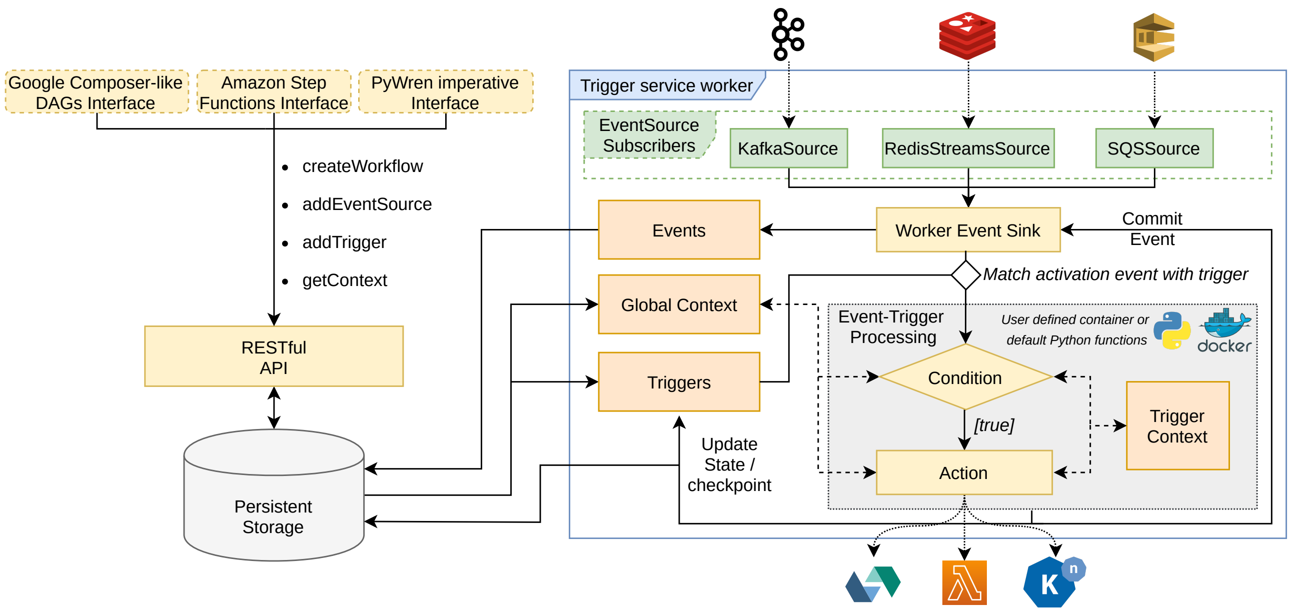


Figure 11: Triggerflow Architecture

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 [26, 27, 28], 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 (PyWren for IBM Cloud).

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 PyWren 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 PyWren 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 PyWren API, the functions call_async and map are used to invoke one or many functions. PyWren 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 PyWren in a reactive way? The solution is to run this PyWren code in Triggerflow reacting to events. Here, prior to perform any invocation, PyWren 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 PyWren 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.

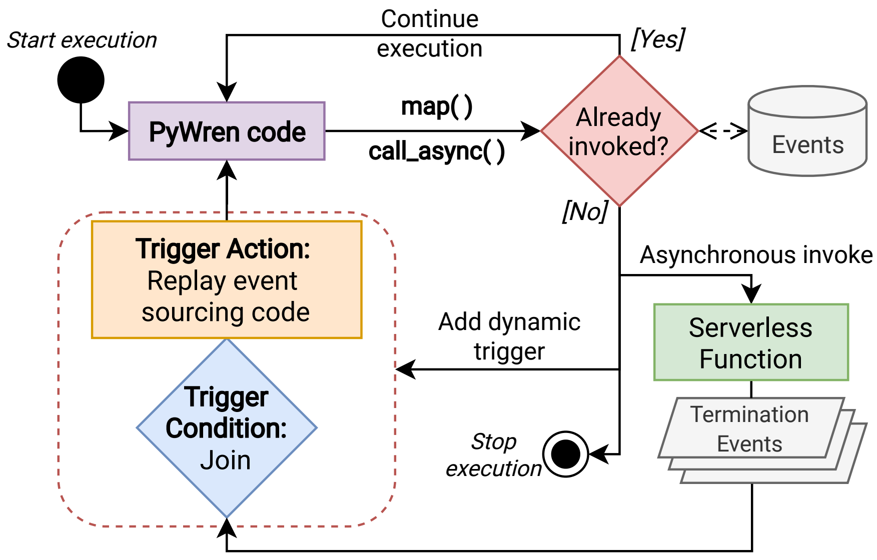


Figure 12: Life cycle of an event sourcing-enabled workflow as code with IBM-PyWren as external scheduler

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 IBM PyWren [4], 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 awaken by our TF-Worker once all map() function activations finished their execution. Moreover, to use our event sourcing version of PyWren, 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 PyWren workflows without requiring any modification. The life cycle of a workflow using an external scheduler can be seen in Figure 12.

5 SLA Monitoring and Management

As described in D3.1[41], the SLA Manager serves the complete lifecycle of an SLA: Template-based SLA agreement description, continuous monitoring, and notification of breaches. During this period in the context of CloudButton, the use of the CloudButton-SLA has focused on enabling the continuous monitoring of Pywren functions execution by using the metrics provided by Knative and Kubernetes at runtime. In this scenario, the collected data will be used to validate the software and to identify candidate performance improvements and I/O problems. As Pywren functions are the basis of our Serverless Data Analytics Platform, by measuring them we will get a close view of the performance of the system. Find below the figure 13 showing the architecture of the SLA Manager.

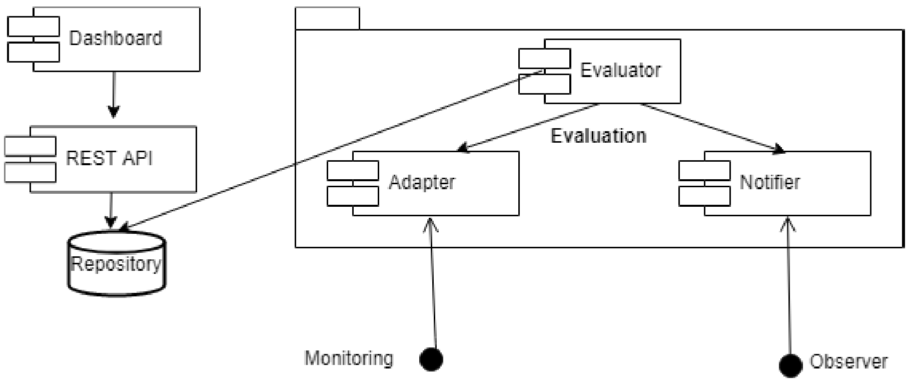


Figure 13: Architecture of SLA Manager.

The monitoring information is collected by an Adapter. The collected metrics should serve to evaluate/validate the agreement in order to identify a breach in the SLA. The Evaluator test agreement(s) accomplishment in runtime and informs the Notifier of any break so that the Observer can take immediate action. The Evaluator obtains the active agreements from a Repository. Several agreements can be active at the same time. In order to adapt our SLA Manager to CloudButton ecosystem we need to answer several questions: What monitoring data do we have available? What are the parameters that describe the QoS of CloudButton ecosystem? Can we define a SLA agreement(s) to meet QoS out of this data? How can we make the QoS information available to CloudButton components?

5.1 CloudButton QoS metrics selection

Find below figure 14 showing CloudButton High Level Architecture, from D3.1

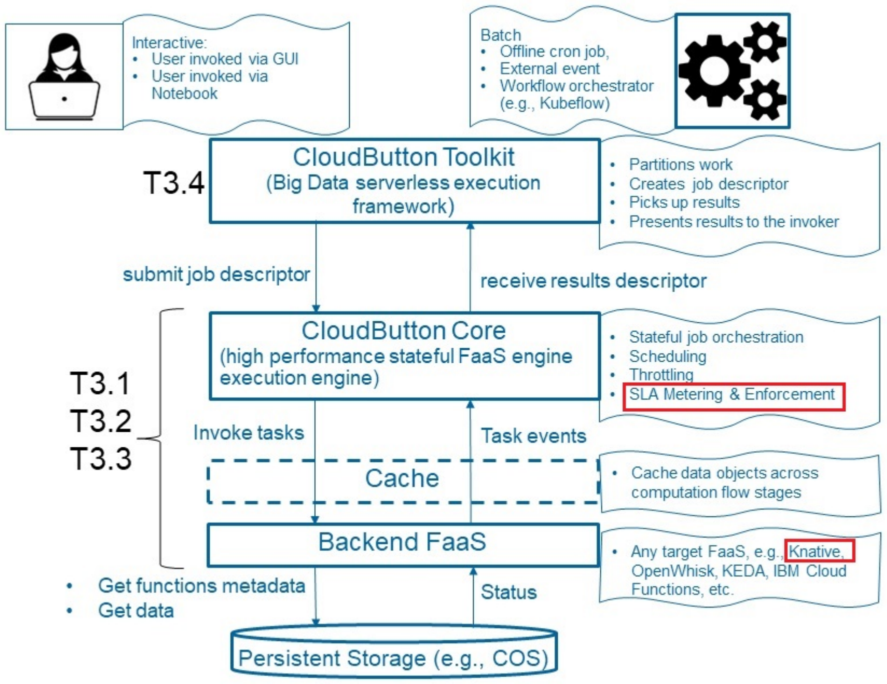

Interactive:
User invoked via GUI
User invoked via Notebook
Batch
Offline cron job,
External event
Workflow orchestrator (e.g., Kubeflow)
T3.4
CloudButton Toolkit
(Big Data serverless execution framework)
Partitions work
Creates job descriptor
Picks up results
Presents results to the invoker
submit job descriptor
receive results descriptor
T3.1
T3.2
T3.3
CloudButton Core
(high performance stateful FaaS engine execution engine)
Stateful job orchestration
Scheduling
Throttling
SLA Metering & Enforcement
Invoke tasks
Task events
Cache
Cache data objects across computation flow stages
Backend FaaS
Any target FaaS, e.g., Knative, OpenWhisk, KEDA, IBM Cloud Functions, etc.
Get functions metadata
Get data
Status
Persistent Storage (e.g., COS)

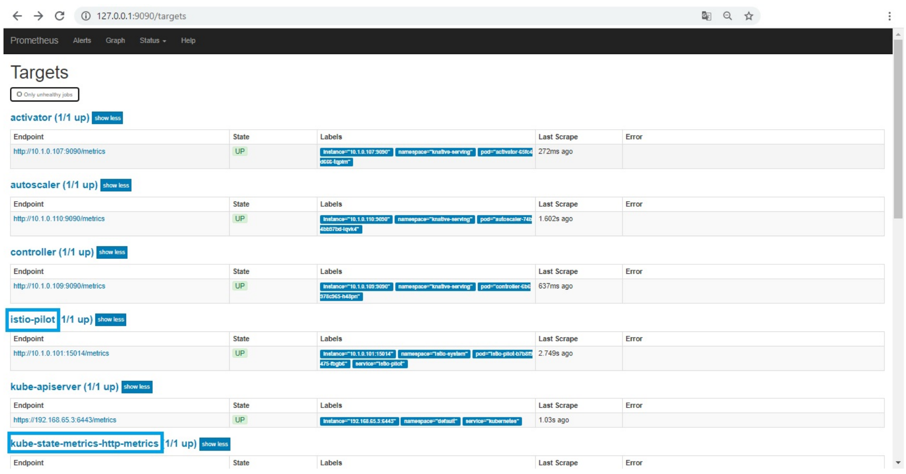
Figure 14: Architecture of SLA Manager.

As shown in the figure 14, CloudButton-SLA sits on the CloudButton Core. A Service Level

Agreement (SLA) allows to express requirements in terms of QoS. The parameters that describe the QoS of CloudButton ecosystem will be obtained from the monitoring of the Backend FaaS, in this context, Knative. As explained in D2.1[48], Knative [49] is a "*Kubernetes-based platform to deploy and manage modern serverless workloads*" It make available the components needed to deploy, run and manage serverless applications based on a Kubernetes cluster (between others). It provides a similar functionality as AWS Lambda [50], Google Cloud Functions[51], or Azure Functions[52] on their public clouds and it can also be used in conjunction with them, or on a private Kubernetes cluster. Knative has two main components:

- **Serving**, that oversees the execution of serverless containers on k8s, leveraging the details of revision tracking, autoscaling and networking.
- **Eventing**, that oversees the distribution of the events between sources and consumers. Istio is customized to be used with Knative Eventing.

Knative also has an **Observability plugin** [53] to supervise the health of the pods running on Knative. It collects metrics, logs and traces on monitoring systems, like Prometheus/Grafana, ElasticSearch/Kibana and Jaege/Zipkin, respectively. They all collect information from Knative Serving component, that is, the execution of containers inside Knative. Once installed, if we access Prometheus, we have available a lot of metrics from different Prometheus targets (sources), but no information is provided by Knative documentation or any other source about what do these metrics measure, other than the name of the measure, that not always is self-explaining. Nevertheless,



Targets

activator (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://10.1.0.107:9090/metrics	UP	instance="10.1.0.107:9090" namespace="knative-serving" pod="activator-65fc4d666-tqplm"	272ms ago	

autoscaler (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://10.1.0.110:9090/metrics	UP	instance="10.1.0.110:9090" namespace="knative-serving" pod="autoscaler-74b4bb97bd-lqvk4"	1.602s ago	

controller (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://10.1.0.109:9090/metrics	UP	instance="10.1.0.109:9090" namespace="knative-serving" pod="controller-6b6978c965-h48pn"	637ms ago	

istio-pilot 1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://10.1.0.101:15014/metrics	UP	instance="10.1.0.101:15014" namespace="istio-system" pod="istio-pilot-b7b8f5475-fbgb6" service="istio-pilot"	2.749s ago	

kube-apiserver (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
https://192.168.65.3:6443/metrics	UP	instance="192.168.65.3:6443" namespace="default" service="kubernetes"	1.03s ago	

kube-state-metrics-http-metrics 1/1 up)

Endpoint	State	Labels	Last Scrape	Error

Figure 15: Some Prometheus target for Knative metrics.

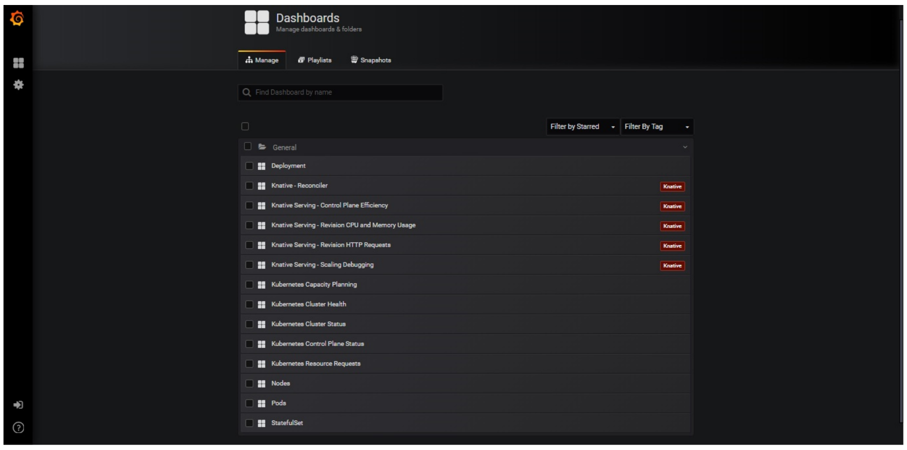
some are interesting target to obtain information about the state of health of our Knative containers, like Kube-state-metrics, or Istio metrics, etc. This replies our first question: *What monitoring data do we have available?*. But still no much information about how to stablish QoS metrics. Together with Prometheus, Knative Observability plugin also installs several Grafana Dashboards, that put together information related with Knative behaviour, and that can be used to monitor the health of the system. The following dashboards are pre-installed with Knative Serving:

In order to produce some metrics to be displayed on the dashboards, we run a Metabolomics use case[54] experiment on the testbed, from a Python Jupyter[55] notebook. More information about the testbed can be found on Deliverable D2.3[56].

The pipeline makes use of Pywren to create a runtime (rutpal/pywren-knative-v37:170) that is executed in Knative Serving, and thus, the metrics that we need are generated.

If we look more in deep to these dashboards, we can identify the Prometheus query used to obtain then, and thus, the metric considered as candidate to QoS metric.

So, this will reply our second question: *What are the parameters that describe the QoS of CloudButton ecosystem?* Quality of Service can be described as a combination of the metrics being used by the


Dashboards
Manage dashboards & folders

Manage | Playlists | Snapshots

Find Dashboard by name

Filter by Starred | Filter By Tag

General
- Deployment
- Knative - Reconciler — Knative
- Knative Serving - Control Plane Efficiency — Knative
- Knative Serving - Revision CPU and Memory Usage — Knative
- Knative Serving - Revision HTTP Requests — Knative
- Knative Serving - Scaling Debugging — Knative
- Kubernetes Capacity Planning
- Kubernetes Cluster Health
- Kubernetes Cluster Status
- Kubernetes Control Plane Status
- Kubernetes Resource Requests
- Nodes
- Pods
- StatefulSet

Figure 16: Grafana Dashboards created by Knative Observability Plugin.

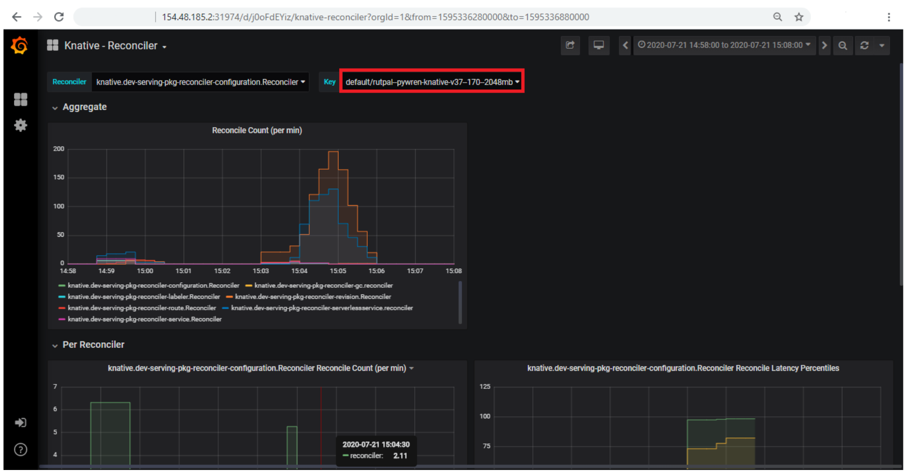
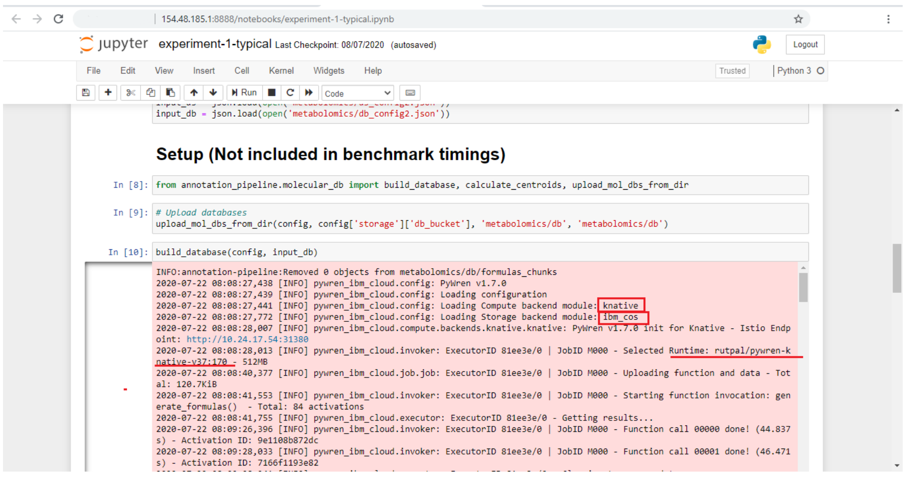

154.48.185.2:31974/d/j0oFdEYiz/knative-reconciler?orgId=1&from=1595336280000&to=1595336880000
Knative - Reconciler
2020-07-21 14:58:00 to 2020-07-21 15:08:00
Reconciler
knative.dev-serving-pkg-reconciler-configuration.Reconciler
Key
default/rutpal-pywren-knative-v37-170-2048mb
Aggregate
Reconcile Count (per min)
knative.dev-serving-pkg-reconciler-configuration.Reconciler
knative.dev-serving-pkg-reconciler-gc.reconciler
knative.dev-serving-pkg-reconciler-labeler.Reconciler
knative.dev-serving-pkg-reconciler-revision.Reconciler
knative.dev-serving-pkg-reconciler-route.Reconciler
knative.dev-serving-pkg-reconciler-serverlessservice.reconciler
knative.dev-serving-pkg-reconciler-service.Reconciler
Per Reconciler
knative.dev-serving-pkg-reconciler-configuration.Reconciler Reconcile Count (per min)
knative.dev-serving-pkg-reconciler-configuration.Reconciler Reconcile Latency Percentiles
2020-07-21 15:04:30
reconciler: 2.11

Figure 17: Grafana Knative Dashboard to control Reconciliation.


154.48.185.1:8888/notebooks/experiment-1-typical.ipynb

jupyter experiment-1-typical Last Checkpoint: 08/07/2020 (autosaved)

File Edit View Insert Cell Kernel Widgets Help — Trusted — Python 3

```
input_db = json.load(open('metabolomics/db_config2.json'))
```

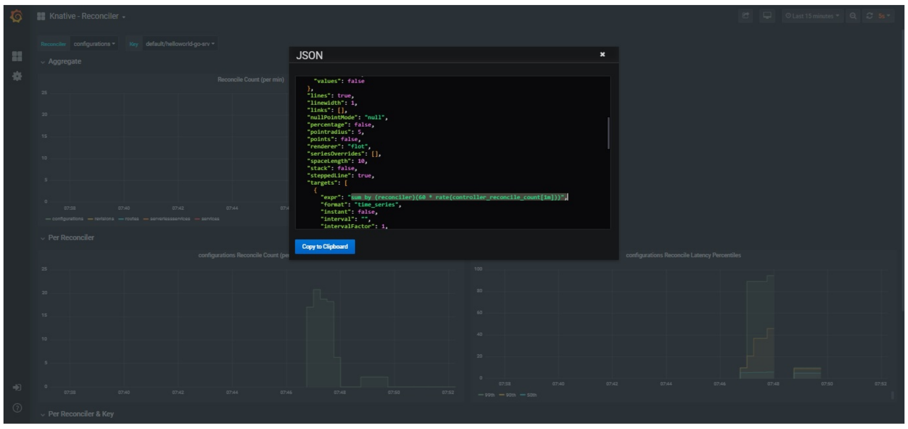
Setup (Not included in benchmark timings)

```
In [8]: from annotation_pipeline.molecular_db import build_database, calculate_centroids, upload_mol_dbs_from_dir

In [9]: # Upload databases
        upload_mol_dbs_from_dir(config, config['storage']['db_bucket'], 'metabolomics/db', 'metabolomics/db')

In [10]: build_database(config, input_db)
```

```
INFO:annotation-pipeline:Removed 0 objects from metabolomics/db/formulas_chunks
2020-07-22 08:08:27,438 [INFO] pywren_ibm_cloud.config: PyWren v1.7.0
2020-07-22 08:08:27,439 [INFO] pywren_ibm_cloud.config: Loading configuration
2020-07-22 08:08:27,441 [INFO] pywren_ibm_cloud.config: Loading Compute backend module: knative
2020-07-22 08:08:27,772 [INFO] pywren_ibm_cloud.config: Loading Storage backend module: ibm_cos
2020-07-22 08:08:28,007 [INFO] pywren_ibm_cloud.compute.backends.knative.knative: PyWren v1.7.0 init for Knative - Istio Endp
oint: http://10.24.17.54:31380
2020-07-22 08:08:28,013 [INFO] pywren_ibm_cloud.invoker: ExecutorID 81ee3e/0 | JobID M000 - Selected Runtime: rutpal/pywren-k
native-v37:170 - 512MB
2020-07-22 08:08:40,377 [INFO] pywren_ibm_cloud.job.job: ExecutorID 81ee3e/0 | JobID M000 - Uploading function and data - Tot
al: 120.7KiB
2020-07-22 08:08:41,553 [INFO] pywren_ibm_cloud.invoker: ExecutorID 81ee3e/0 | JobID M000 - Starting function invocation: gen
erate_formulas()  - Total: 84 activations
2020-07-22 08:08:41,755 [INFO] pywren_ibm_cloud.executor: ExecutorID 81ee3e/0 - Getting results...
2020-07-22 08:09:26,396 [INFO] pywren_ibm_cloud.invoker: ExecutorID 81ee3e/0 | JobID M000 - Function call 00000 done! (44.837
s) - Activation ID: 9e1108b872dc
2020-07-22 08:09:28,033 [INFO] pywren_ibm_cloud.invoker: ExecutorID 81ee3e/0 | JobID M000 - Function call 00001 done! (46.471
s) - Activation ID: 7166f1193e82
```

Figure 18: Grafana Knative Dashboard to control Reconciliation.

Knative Observability plugin to control different aspects of the service, like CDR reconciliation, CPU and Memory usages, control plane efficiency, etc



```
    "values": false
  },
  "lines": true,
  "linewidth": 1,
  "links": [],
  "nullPointMode": "null",
  "percentage": false,
  "pointradius": 5,
  "points": false,
  "renderer": "flot",
  "seriesOverrides": [],
  "spaceLength": 10,
  "stack": false,
  "steppedLine": true,
  "targets": [
    {
      "expr": "sum by (reconciler)(60 * rate(controller_reconcile_count[1m]))",
      "format": "time_series",
      "instant": false,
      "interval": "",
      "intervalFactor": 1,
```

Figure 19: Prometheus query to create Grafana Dashboard.

5.2 Prometheus CloudButton-SLA Monitor

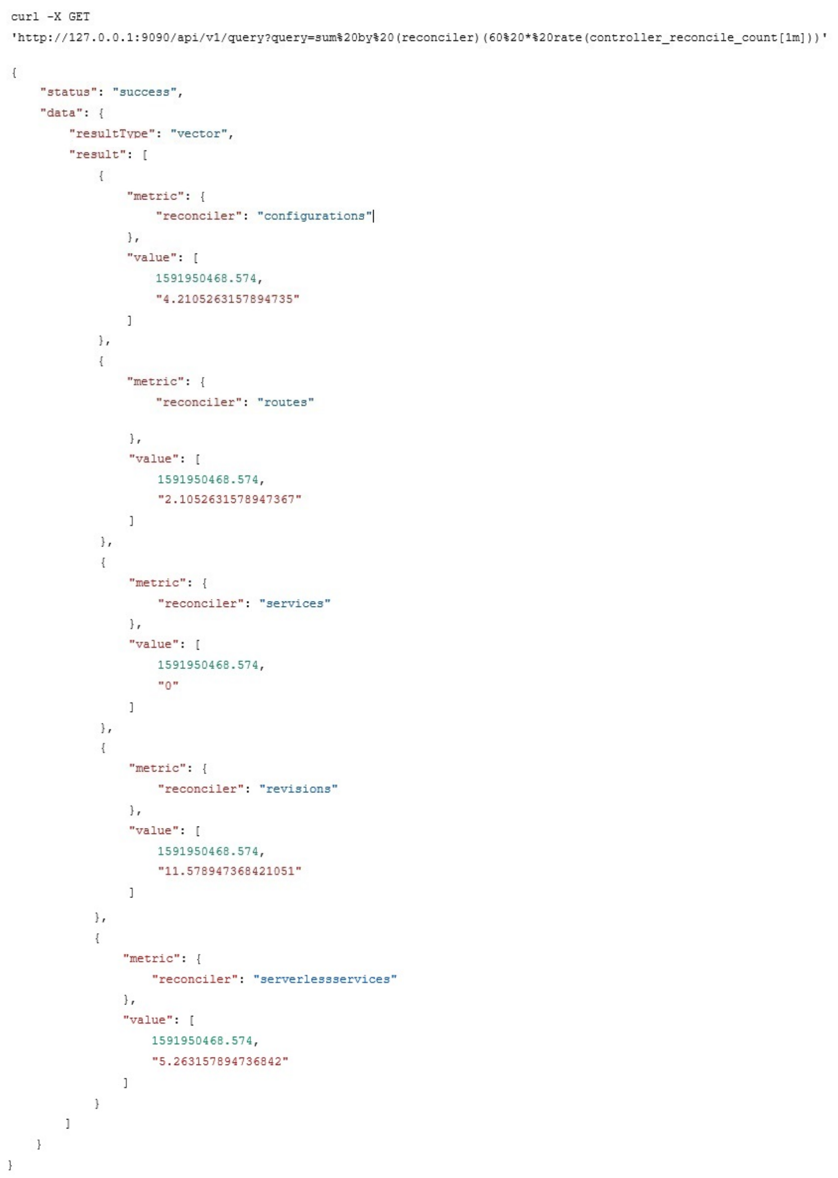
In order to collect the data that is relevant to our QoS and thus, be able to assess SLA compliance, we need to implement a Prometheus Monitor into our SLALite. Prometheus count with a REST API [57] that will be used to obtain the metrics at a certain point in time or from a period. The output with the metric values will be used to assess the different Agreements that are active in our SLA, and to notify possible violations of the agreement, for the system to take actions. The queries/replies to Prometheus metrics are in the form of:

The data section of the query result has one of the following formats:

```
{
  "resultType": "matrix" | "vector" | "scalar" | "string",
  "result": <value>
}
```

In the example, the metrics are the sum of the reconciler (i.e: revisions). Value contains the metric itself (i.e: 1591950468.574) and the time in which it is measured (1591950468.574), in UNIX format. You can find more information about "reconciler" in Kubernetes in here [58] The SLA is defined on the following sample Agreement.

```
{
    "id": "a4",
    "name": "an-agreement-name",
    "state": "started",
    "details":{
        "id": "a4",
        "type": "agreement",
        "name": "an-agreement-name",
        "provider": { "id": "a-provider", "name": "A provider" },
        "client": { "id": "a-client", "name": "A client" },
        "creation": "2020-01-01T17:09:45Z",
        "expiration": "2021-01-01T17:09:45Z",
        "variables": [
            {
             "name": "Reconciler",
             "metric": "sum by(reconciler)(60*rate(controller_reconcile_count[1m]))"
            }
        ],
        "guarantees": [
```



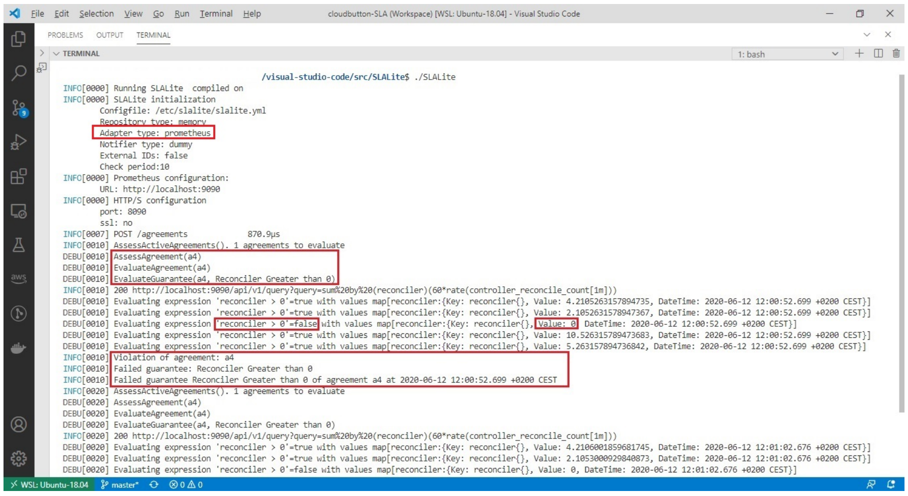
```
curl -X GET
'http://127.0.0.1:9090/api/v1/query?query=sum%20by%20(reconciler)(60%20*%20rate(controller_reconcile_count[1m]))'

{
    "status": "success",
    "data": {
        "resultType": "vector",
        "result": [
            {
                "metric": {
                    "reconciler": "configurations"
                },
                "value": [
                    1591950468.574,
                    "4.2105263157894735"
                ]
            },
            {
                "metric": {
                    "reconciler": "routes"

                },
                "value": [
                    1591950468.574,
                    "2.1052631578947367"
                ]
            },
            {
                "metric": {
                    "reconciler": "services"
                },
                "value": [
                    1591950468.574,
                    "0"
                ]
            },
            {
                "metric": {
                    "reconciler": "revisions"
                },
                "value": [
                    1591950468.574,
                    "11.578947368421051"
                ]
            },
            {
                "metric": {
                    "reconciler": "serverlessservices"
                },
                "value": [
                    1591950468.574,
                    "5.263157894736842"
                ]
            }
        ]
    }
}
```

Figure 20: Prometheus metric JSON output.

```
            {
             "name": "Reconciler Greater than 0",
             "constraint": "reconciler > 0"
            }
        ]
    }
}
```

The Service Level Agreement can be made up by several **Agreements** that are assessed and notify violations on an independent manner. Also, a **Guarantee** can be defined as a complex combination of different metric to define **Constraints**. Finally, the **Agreement** is only assessed if it is on a started **State** and between its **Creation** and **Expiration** dates. So this replies our third question, *we can define a SLA agreement(s) to meet QoS and assess it from the data provided by Knative Observability plugin?*. Taking this into account, we have coded a monitor that queries the metrics provided by Prometheus' Knative Observability plugin and assess its compliance, sending a violation in case any of the agreements is not fulfilled.

Here follow the logs of the CloudButton-SLA identifying a violation on our sample agreement guarantee:


File Edit Selection View Go Run Terminal Help
cloudbutton-SLA (Workspace) [WSL: Ubuntu-18.04] - Visual Studio Code
PROBLEMS
OUTPUT
TERMINAL
TERMINAL
1: bash
WSL: Ubuntu-18.04
master*
0 0

```
                                   /visual-studio-code/src/SLALite$ ./SLALite
INFO[0000] Running SLALite  compiled on
INFO[0000] SLALite initialization
        Configfile: /etc/slalite/slalite.yml
        Repository type: memory
        Adapter type: prometheus
        Notifier type: dummy
        External IDs: false
        Check period:10
INFO[0000] Prometheus configuration:
        URL: http://localhost:9090
INFO[0000] HTTP/S configuration
        port: 8090
        ssl: no
INFO[0007] POST /agreements            870.9µs
INFO[0010] AssessActiveAgreements(). 1 agreements to evaluate
DEBU[0010] AssessAgreement(a4)
DEBU[0010] EvaluateAgreement(a4)
DEBU[0010] EvaluateGuarantee(a4, Reconciler Greater than 0)
INFO[0010] 200 http://localhost:9090/api/v1/query?query=sum%20by%20(reconciler)(60*rate(controller_reconcile_count[1m]))
DEBU[0010] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 4.2105263157894735, DateTime: 2020-06-12 12:00:52.699 +0200 CEST}]
DEBU[0010] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 2.1052631578947367, DateTime: 2020-06-12 12:00:52.699 +0200 CEST}]
DEBU[0010] Evaluating expression 'reconciler > 0'=false with values map[reconciler:{Key: reconciler{}, Value: 0 DateTime: 2020-06-12 12:00:52.699 +0200 CEST}]
DEBU[0010] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 10.526315789473683, DateTime: 2020-06-12 12:00:52.699 +0200 CEST}]
DEBU[0010] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 5.263157894736842, DateTime: 2020-06-12 12:00:52.699 +0200 CEST}]
INFO[0010] Violation of agreement: a4
INFO[0010] Failed guarantee: Reconciler Greater than 0
INFO[0010] Failed guarantee Reconciler Greater than 0 of agreement a4 at 2020-06-12 12:00:52.699 +0200 CEST
INFO[0020] AssessActiveAgreements(). 1 agreements to evaluate
DEBU[0020] AssessAgreement(a4)
DEBU[0020] EvaluateAgreement(a4)
DEBU[0020] EvaluateGuarantee(a4, Reconciler Greater than 0)
INFO[0020] 200 http://localhost:9090/api/v1/query?query=sum%20by%20(reconciler)(60*rate(controller_reconcile_count[1m]))
DEBU[0020] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 4.2106001859681745, DateTime: 2020-06-12 12:01:02.676 +0200 CEST}]
DEBU[0020] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 2.1053000929840873, DateTime: 2020-06-12 12:01:02.676 +0200 CEST}]
DEBU[0020] Evaluating expression 'reconciler > 0'=false with values map[reconciler:{Key: reconciler{}, Value: 0, DateTime: 2020-06-12 12:01:02.676 +0200 CEST}]
```

Figure 21: CloudButton-SLA identifies a violation of our sample SLA agreement.

5.3 RabbitMQ CloudButton-SLA Notifier

Now that we have a way to define a QoS SLA for CloudButton Core, and to identify violations of agreements, we'll like to take appropriated remediation actions to get back to "normal". There will be different components of CloudButton Core that would need to be aware of the breach of any of the active SLA agreements. Notifications of guarantee violations will be produced into a message broker, where can be easily consumed by other entities. RabbitMQ[59] is a message broker that implements Advanced Message Queuing Protocol (AMQP). The new RabbitMQ CloudButton-SLA Notifier will create a queue called CloudButton where violations will be notified. The information produced and shared in the queue will be in the format:

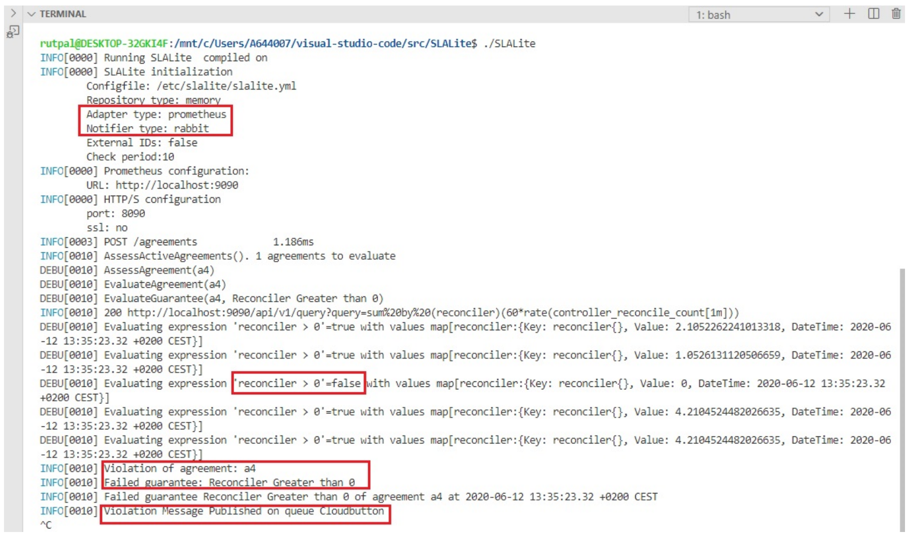
```
{
"Application":"a4",
"Fields":{
"Guarantee":"Reconciler Greater than 0",
"IdAgreement":"a4",
"ViolationTime":"2020-06-12T13:34:17.824+02:00"
},
"Message":"QoS_Violation"
}
```

Here follow the message queued by the CloudButton-SLA on the RabbitMQ CloudButton queue after identifying a violation on our sample agreement guarantee:

5.4 Future Actions

In order to make CloudButton-SLA component fully operative, we consider the following improvements:

- Fully define an SLA to supervise CloudButton Core components and running processes. Each of the applications might need to define its own guarantees. To accomplish this objective, we'll have to do a benchmarking of the system, supervising the metrics already identified to stablish a mainline and notify above and below deviations.
- Consider adding dynamic information to the SLA definition, like number of pods, namespace in which the application is running, etc. Also capability to modify the agreement dynamically



```
rutpal@DESKTOP-32GKI4F:/mnt/c/Users/A644007/visual-studio-code/src/SLALite$ ./SLALite
INFO[0000] Running SLALite  compiled on
INFO[0000] SLALite initialization
        Configfile: /etc/slalite/slalite.yml
        Repository type: memory
        Adapter type: prometheus
        Notifier type: rabbit
        External IDs: false
        Check period:10
INFO[0000] Prometheus configuration:
        URL: http://localhost:9090
INFO[0000] HTTP/S configuration
        port: 8090
        ssl: no
INFO[0003] POST /agreements            1.186ms
INFO[0010] AssessActiveAgreements(). 1 agreements to evaluate
DEBU[0010] AssessAgreement(a4)
DEBU[0010] EvaluateAgreement(a4)
DEBU[0010] EvaluateGuarantee(a4, Reconciler Greater than 0)
INFO[0010] 200 http://localhost:9090/api/v1/query?query=sum%20by%20(reconciler)(60*rate(controller_reconcile_count[1m]))
DEBU[0010] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 2.1052262241013318, DateTime: 2020-06
-12 13:35:23.32 +0200 CEST}]
DEBU[0010] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 1.0526131120506659, DateTime: 2020-06
-12 13:35:23.32 +0200 CEST}]
DEBU[0010] Evaluating expression 'reconciler > 0'=false with values map[reconciler:{Key: reconciler{}, Value: 0, DateTime: 2020-06-12 13:35:23.32
+0200 CEST}]
DEBU[0010] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 4.2104524482026635, DateTime: 2020-06
-12 13:35:23.32 +0200 CEST}]
DEBU[0010] Evaluating expression 'reconciler > 0'=true with values map[reconciler:{Key: reconciler{}, Value: 4.2104524482026635, DateTime: 2020-06
-12 13:35:23.32 +0200 CEST}]
INFO[0010] Violation of agreement: a4
INFO[0010] Failed guarantee: Reconciler Greater than 0
INFO[0010] Failed guarantee Reconciler Greater than 0 of agreement a4 at 2020-06-12 13:35:23.32 +0200 CEST
INFO[0010] Violation Message Published on queue Cloudbutton
^C
```

Figure 22: AMQP Message from a Violation.

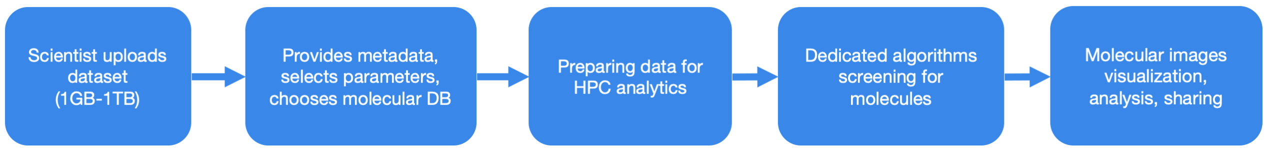
to adapt to changes in the workload, for example, or to define on-the-fly agreements for supervising specific situations.

- As we mentioned on D3.1[41], the existing baseline of metrics in Prometheus can be complemented with specific exporters if necessary, i.e. in order to collect information for the status of GPUs, for example.

6 Initial evaluation

6.1 EMBL use case

In our previous work [] we built initial prototype which enables metabolomics "Imaging mass spectrometry" application to be executed against serverless backend. Before that Imaging mass spectrometry application deployed over Apache Spark running in the dedicated cluster. Figure 23 shows general application flow, where all the steps executed inside Apache Spark cluster



Scientist uploads dataset (1GB-1TB)
Provides metadata, selects parameters, chooses molecular DB
Preparing data for HPC analytics
Dedicated algorithms screening for molecules
Molecular images visualization, analysis, sharing

Figure 23: "Imaging mass spectrometry" flow

In our initial prototype D3.1[41] we modifed "Imaging mass spectrometry" application by replacing calls to Apache Spark with the calls to the CloudButton toolkit. As a proof of concept we demonstrated how our architecture enables to process small datasets and small databases by dynamically obtain computational resources from the FaaS backend. We used IBM Cloud Object storage to store the input datasets and databases and also for the generated output images.

Continuing our efforts beyond initial implementation we had to address the main objective and challenges as follows

6.1.1 Enablement to process any size datasets and databases

Processing any size datasets and databases is one of the main challenges of the CloudButton Toolkit. Our initial implementation in [] used small dataset with 1,477,568,961 records and a database with 2,412,804 molecules. While this worked fine, our objective was to ensure our architecture can process any size datasets and databases. We decided to experiment with dataset with 91,266,700,713 records (we call it "Dataset5") and molecular database with 18,664,433 molecules (we call it "Database4"). This combination is in particular importance as running such configuration was impossible over Apache Spark deployed the cluster 4 VMs. Diving into these challenges, a preprocessing part is required. The input dataset is rearranged into ordered memory bounded segments, the order is defined by m/z values (mass divided by charge number of ions) which considered as unique identifiers for molecules. Because of the dataset isn't memory bounded and could be too large to process, we need CloudButton Toolkit to shuffle unordered memory bounded dataset chunks and then generate dataset segments for each chunk separately. After that, CloudButton Toolkit collects the same segments from different chunks independently and merges them into the desirable ordered memory bounded segments. With this approach, we can divide the dataset quickly into separated segments by reading the whole dataset only once. The entire flow can be seen in 24. The same distributed algorithm is applied over a molecular database contains molecules formulas and their m/z values for annotation.

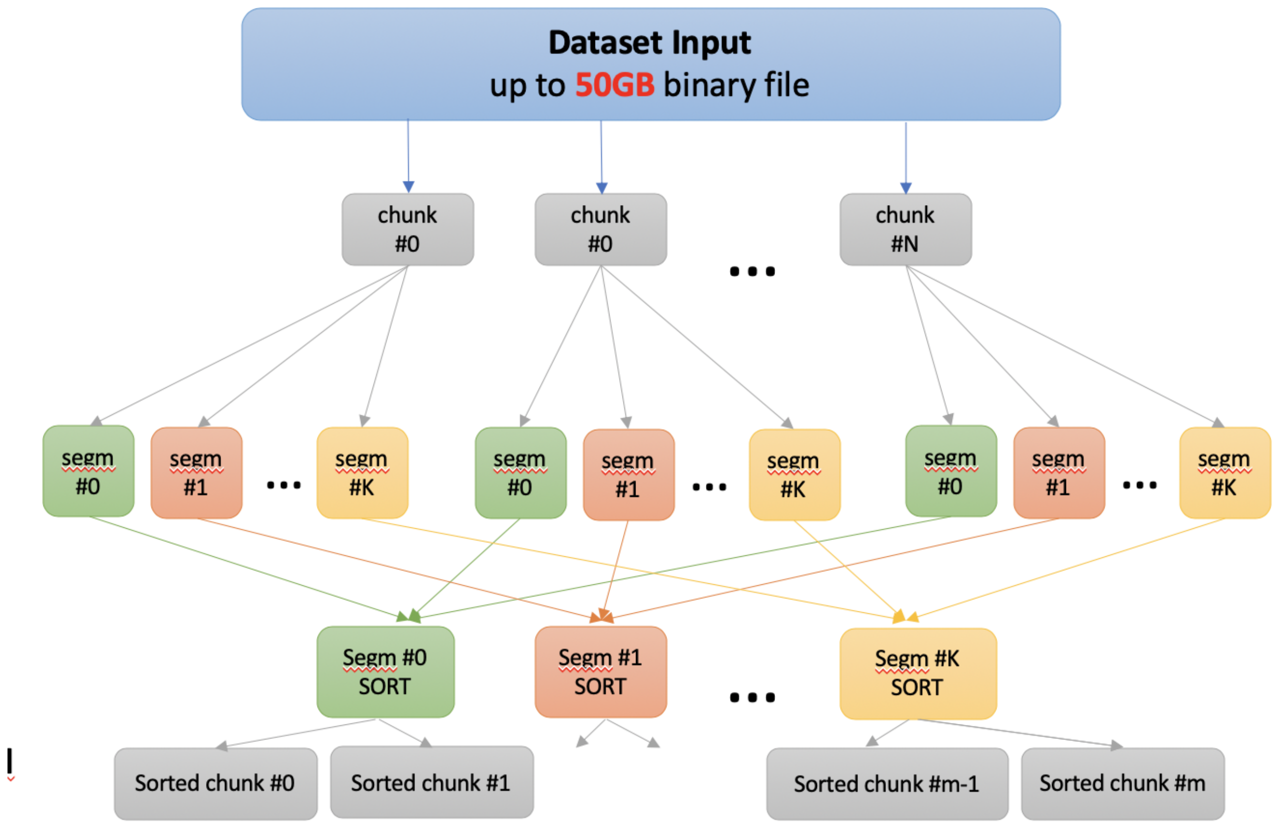


Figure 24: Dataset chunking algorithm

Using our unique approach allowed us to process any size datasets and any size databases.

6.1.2 Imaging mass spectrometry application with CloudButton

We present now details on the Metabolomics use case architecture which integrates general CloudButton framework. Figure 25 contains high level flow with various steps and stages. We now explain in details each step

6.1.3 Mol DB processing

Gets input molecular database contains base molecules formulas and applies serverless sorting algorithm to produce sorted formulas chunks. Afterwards, each chunk is fitted with required molecular details for the molecular search.

1. **generate formulas** Launches parallel IBM Cloud Fucntion serverless jobs which generate similar formulas by each base one. Each serverless job downloads a molecular subset and applies config-

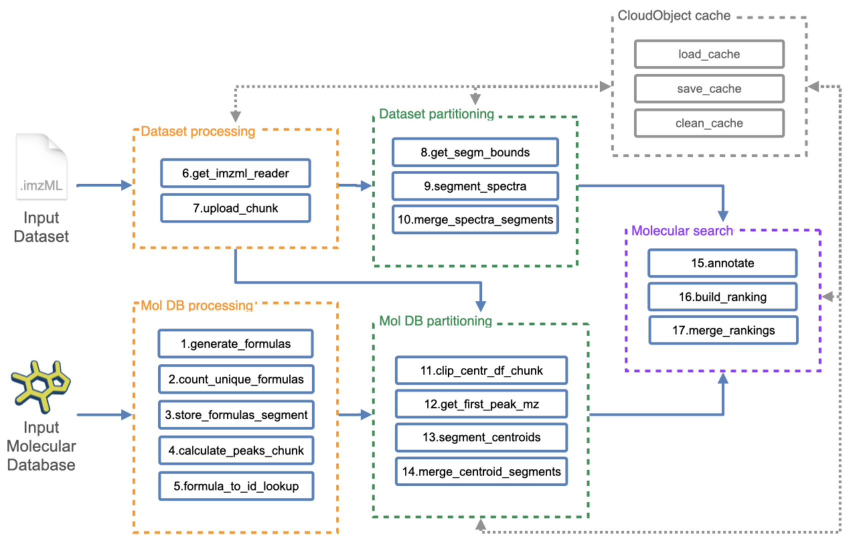


Figure 25: Metabolomics and CloudButton architecture

urable adducts and modifiers for each formula. The adducts and modifiers appliance simulates the chemical reactions that each base molecule can be transformed by, therefore the formulas generation can embarrassedly scale. After the generation process, each job stores the resulted formulas into IBM Cloud Object Storage objects and because of the fact that each job is done independently, there are a lot of cases in which same formulas are generated by different jobs and thus duplications are saved during the process. This behaviour enlarge significantly the total amount of formulas to process in vain. For solving this issue, The decision of which formula goes into which object index is done by hashing method that applied on each formula string. By this way, the process ensures that all same formula strings are mapped into the same intermediate objects indices of each parallel job.

2. **count unique formulas** A preprocess stage that launches parallel IBM Cloud Function serverless jobs according to the number of different intermediate storage objects indices from the previous stage and calculates the number of unique formulas strings of each objects index. Each job downloads the relevant intermediate objects for a specific index so it is guaranteed that all same formulas strings were stored into the same objects index.

3. **store formulas segments** Launches parallel IBM Cloud Function serverless jobs according to the number of different intermediate storage objects indices from the 1'st stage, manipulates each intermediate object and store the final formulas chunks into new IBM Cloud Object Storage objects. Each job downloads the relevant intermediate objects for a specific index, merges all formulas groups together and removes redundant formulas duplications. Again, it is guaranteed that all same formulas strings were stored into the same objects index. After this process, the final sorted formulas chunks are stored and the intermediate objects are cleaned.

4. **calculate peaks chunk** Launches parallel IBM Cloud Function serverless jobs according to the number of formulas chunks that were produced by the previous stage and calculates its molecular details. Each job downloads a specific formulas chunk and calculates molecular details for each formula. The resulted details are attached with its relevant formulas ids and stored into new IBM Cloud Object Storage objects. These details are the estimated input dataset values that indicates if the corresponding formulas are included inside the dataset or not.

5. **formula to id lookup** Launches parallel IBM Cloud Function serverless jobs according to the maximum size of formulas chunks that can be fitted into a serverless job. Each job downloads a group of formulas chunks and build a user friendly python dictionaries to convert formulas ids to formulas strings and vice versa.

6.1.4 Dataset processing

Gets input binary dataset files that are stored in IBM Cloud Object Storage and parses the dataset into memory bounded raw values chunks. The dataset input files are composed by a main compressed binary ibd file that contains molecular details and positions, and by a metadata imzML file that contains technical details for parsing the ibd file.

6. **get imzml reader** Launches a single IBM Cloud Function serverless job utilising IBM Cloud network performance. The job downloads the imzML file and load it into a dedicated portable parser. Afterwards, the parser is stored into an IBM Cloud Object Storage object.

7. **upload chunk** Launches parallel IBM Cloud Function serverless jobs according to the maximum size that each serverless job can fit and process. Each job downloads the portable parser from the previous stage and parses an indexed memory bounded chunk of the ibd file. Afterwards, the resulted chunks are stored into new IBM Cloud Object Storage objects.

6.1.5 Dataset partitioning

Gets unordered memory bounded dataset chunks and applies serverless sorting algorithm to produce sorted memory bounded dataset segments.

8. **get segm bounds** A preprocess stage that launches a single IBM Cloud Function serverless job utilising IBM Cloud network performance and defines dynamically the number of required dataset segments. The job downloads the portable parser and samples a minor subset of the dataset rows to estimate the dataset distribution. Afterwards, the process calculates estimated ranges of molecular details such that all ranged molecular details will be equally sized. These ranges represent the molecular details to be stored into each resulted dataset segment.

9. **segment spectra** Launches parallel IBM Cloud Function serverless jobs according to the number of dataset chunks that were produced by the 7'th stage and divides each chunk into estimated segments according to the molecular details ranges that were produced by the previous stage. Each job downloads an unordered memory bounded dataset chunk, sorts its values and divides it into sub-segments. The number of sub-segments for each chunk is predefined such that the whole sorting process will use the minimum required number of serverless jobs and storage requests. At the end of each job, each sub-segment is stored into an intermediate IBM Cloud Object Storage object.

10. **merge spectra segments** Launches parallel IBM Cloud Function serverless jobs according to the number of intermediate sub-segments from the previous stage and merges corresponding ranged values. Each job downloads same index intermediate storage objects, merges and sorts its values and stores the resulted segments into IBM Cloud Object Storage. Because of the fact that each indexed intermediate storage objects are associated with a predefined range, the final result is an ordered dataset segment regarding for the whole dataset. In addition, the process ensures in an estimated way that each resulted segment is also memory bounded.

6.1.6 Mol DB partitioning

Gets unordered database chunks and applies serverless sorting algorithm to produce sorted database segments.

11. **clip centr df chunk** Launches parallel IBM Cloud Function serverless jobs according to the number of database chunks that were produced by the 4'th stage and removes redundant molecular formulas. Each job downloads a database chunk, clips its data regarding expected minimum and

maximum dataset values and stores each resulted chunk into a new IBM Cloud Object Storage object.

12. **get first peak mz** A preprocess part that launches parallel IBM Cloud Function serverless jobs according to the number of database chunks that were produced by the 4'th stage and samples it to estimate the data distribution. Afterwards, all samples are merged in the local machine and the process calculates estimated ranges of molecular details such that all ranged molecular details will be equally sized. These ranges represent the molecular details to be stored into each resulted dataset segment.

13. **segment centroids** Launches parallel IBM Cloud Function serverless jobs according to the number of database chunks that were produced by the 4'th stage and generates estimated segments for each chunk according to the molecular details ranges that were produced by the previous stage. Each job downloads an unordered memory bounded database chunk, sorts its values and divides it into sub-segments. The number of sub-segments for each chunk is predefined such that the whole sorting process will use the minimum required number of serverless jobs and storage requests. At the end of each job, each sub-segment is stored into an intermediate IBM Cloud Object Storage object.

14. **merge centroid segments** Launches parallel IBM Cloud Function serverless jobs according to the number of intermediate sub-segments from the previous stage and merges corresponding ranged values. Each job downloads same index intermediate storage objects, merges and sorts its values and stores the resulted segments into IBM Cloud Object Storage. Because of the fact that each indexed intermediate storage objects are associated with a predefined range, the final result is an ordered dataset segment regarding for the whole dataset. In addition, at the end of he process, it decides dynamically whether split again the resulted database segments or not, according to the total summary of the relevant ordered dataset segments sizes that will fit into next stages serverless jobs (using dataset segments bounds that were produced by the 8'th stage).

6.1.7 Molecular search

15. **annotate** The main method that gets each database segment and searches its appearance and corresponding positions over the relevant dataset segments. The method launches parallel IBM Cloud Function serverless jobs according to the number of database segments that were produced by the previous stage. Each job downloads a database segment, downloads relevant dataset segments (according to the database segments values range) and generates images representing the relative positions of each database formula over the input dataset. Due to previous sorting stages, Each job is time and memory efficient because it can load only a memory bounded subset of the total dataset segments for each given database segment. In addition, each job uses a dedicated image manager that responsible for managing intermediate images results memory consumption during the process. If the memory consumption of each job gets too close to its memory limit, the manager stores the resulted data and flushes its pointers. by this way, the annotation processes utilise the balance between memory and time serverless limitations.

16. **build ranking** Launches parallel IBM Cloud Function serverless jobs according to different combinations of configurable modifiers and adducts of each database formula and attaches FDR ranking for each database formula. FDR, False Discovery Rate, is a measurement for expected false annotations. The resulted rankings are stored into IBM Cloud Object Storage.

17. **merge rankings** Launches parallel IBM Cloud Function serverless jobs according to the number of unique rankings that were produced by the previous stage and merges similar formulas ranking data. The resulted data is stored in IBM Cloud Object Storage and the previous storage objects are cleaned.

Summary Our core approach is decentralized and completely serverless, where we let the CloudButton framework determine the appropriate scale of parallelism needed to process input datasets.

To achieve this, our code evaluates the input datasets and then decides on the number of serverless actions required, with the aim to maximize performance and the costs of the processing. This approach allows us to dynamically adjust the amount of compute resources while the data is being processed—which is in contrast to a Spark-based approach, where the amount of compute resources is determined before starting the data processing and can't be adjusted as the processing progresses. As part of initial evaluation we choose combination of input dataset of 14.25GB size that contains 14,940,245,452 fields and molecular database of 0.22GB size with 2,412,804 molecules. We processed given input over Apache Spark deployed over 4 EC2 machines of 32GB each and compared to processing the same input with CloudButton Toolkit over IBM Cloud Functions and IBM Cloud Object Storage. By using CloudButton Toolkit we processed given datasets 4 times faster over IBM Cloud Functions comparing to serverfull approach based on Apache Spark. The CloudButton serverless solution allowed us to process datasets which were previously out of reach, and without additional efforts for infrastructure maintenance, Apache Spark cluster setup and configuration, and code deployment.

6.2 Geospatial Use Case

The 3 experiments of the geospatial use case have completed their transition from single-machine code to serverless functions. The prototypes use Jupyter notebooks with calls to the CloudButton toolkit API. We use IBM Cloud Functions and IBM Cloud Storage as compute and storage backend, respectively.

The transition to serverless has implied some changes to the previous single-machine approach. For instance, large satellite images or LiDAR point clouds have to be partitioned due to the memory restrictions of serverless functions. Some auxiliar libraries or frameworks have also been changed to adapt to serverless limitations, for instance using scikit-learn instead of Spark standalone for machine learning training. Other tools are third party libraries that require a lot of memory and cannot be parallelized. In this cases, we opted for running the workflows with some serverless steps and some sequential steps.

Current efforts are centered on running the experiments in the CloudButton testbed, with Knative and Ceph backends. This will be possible thanks to the backend-agnostic design of the CloudButton toolkit. We also are working in the development of benchmarks to evaluate the performance and cost of the serverless implementations.

6.3 Cost-efficiency

As part of task T3.2, we aim to explore cost-effectiveness tradeoffs involved with applying serverless computing model to Big Data analytics. In this context, in our recent vision paper entitled "Serverless End Game: Disaggregation enabling transparency"[60] we present two experiments that analyze the trade-offs between performance and cost when comparing serverless and local computing resources.

To evaluate the feasibility of compute disaggregation with state of the art cloud technologies, we will compare a compute-intensive algorithm running in local threads in a VM compared to the same algorithm running over serverless functions. We also provide code transparency, since we execute the same code in both cases. To achieve this transparency, we rely on a Java Serverless Executor [61] that can execute Java threads over remote Lambda functions. In this case, all state is passed as parameters to the functions/threads, and functions are in warm state, like VMs which are already provisioned.

This experiment runs a Monte Carlo simulation to estimate the value of π. At each iteration, the algorithm checks if a random point in a 2D square space lies inside the inscribed quadrant. We run 48 billion iterations of the algorithm. For AWS Lambda, the iterations are evenly distributed to 16, 36, 48 or 96 functions with 1792 MB of memory.[12] For virtual machines, we run a parallel version of the simulation in different instance sizes: c5.4xlarge (16 vCPUs), c5.9xlarge (36 vCPUs), c5.12xlarge (48 vCPUs), c5.24xlarge (96 vCPUs). The algorithm is implemented in Java.

As we can see in Figure 26, the major difference now is cost: for an equivalent execution, dis-

[12]According to AWS documentation, at 1,792MB a function has the equivalent of one full vCPU

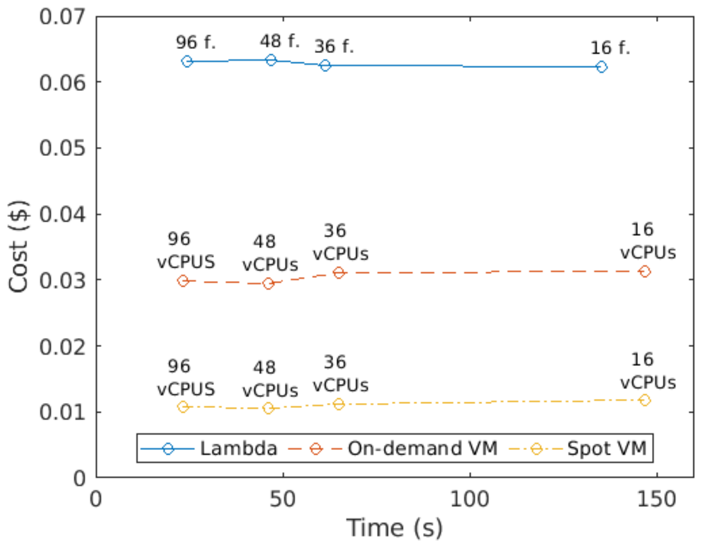


Figure 26: Monte Carlo simulation in VMs versus Amazon Lambda Functions

aggregated functions cost 2x more compared to on-demand VMs, and 6x more compared to Spot instances. Surprisingly, computation time is equivalent in the local and remote version using Lambdas. Even considering all the network communication overheads, container management and remote execution, the results for disaggregated computations are already competitive in performance in existing clouds. This is of course happening because this experiment is embarrassingly parallel, and the duration of compute tasks is long enough to make milliseconds (15/100ms) overheads negligible.

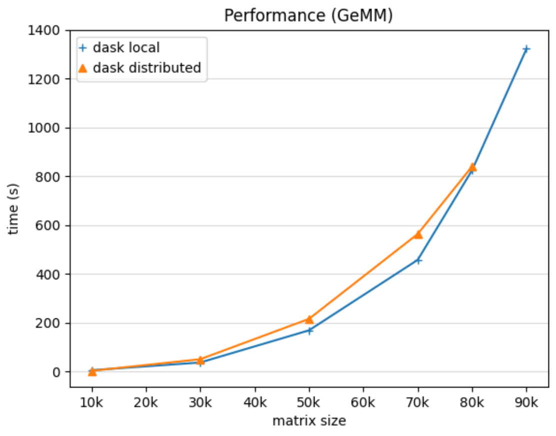
The second experiment evaluates the feasibility and costs of both memory and compute disaggregation with existing cloud technologies. In this case, we evaluate a linear algebra algorithm, Matrix Multiplication (GEMM) which is a good use-case for testing parallel processing on large in-memory data structures.

We rely on Python frameworks used by data scientists like NumPy and Dask. Dask transparently enables to run the same code in a single multi-core machine or VM, and in a distributed cluster of nodes. We also compare Dask to a serverless implementation of NumPy called numpywren [62] using serverless functions that access data in disaggregated Cloud Object Storage (Amazon S3).

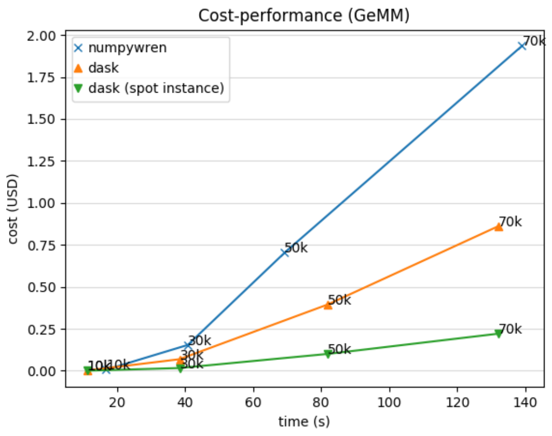
The first part of the experiment compares the performance of Matrix Multiplication (GEMM) using Dask in a local VM (1x r5.24xlarge) and in a distributed cluster (6x r5.4xlarge) using the same resources (96 vCPUs, 768 GiB memory, 10Gb network). Figure 27a shows that the local version perform slightly better than the distributed one while costing the same. In this case, locality is avoiding unnecessary data movements and serialization costs, and cluster provisioning. Experiments with 90Kx90K matrices can be executed in the local VM, but not in the equivalent distributed cluster due to resource exhaustion.

The second part of the experiment compares the cost and performance of Matrix Multiplication (GEMM) using Dask in a distributed cluster (on demand VMs or Spot instances) and using numpywren over Amazon Lambda and Amazon S3. We calculate compute resources in numpywren (vCPUs) as the ratio between the sum of the duration of every Lambda and the wall-clock time of the experiment. In GeMM (70Kx70K) numpywren uses 553.8 vCPUs and in Dask we use equivalent resources: 552 vCPUs (5x c5.24xlarge, 1x c5.18xlarge).

Figure 27b shows that Dask obtains the same performance in VMs and Spot instances, but Spot instances are 4x cheaper than on demand VMs. numpywren obtains good performance numbers for large matrices, obtaining equivalent performance results for an equivalent Dask cluster in running time. numpywren also shows automatic scaling for any size, whereas the Dask cluster must always be provisioned in advance with the desired amount of resources. Finally, numpywren is much more expensive than the Dask cluster using Spot instances (14x for 10K, 9x for 30K, 6.9x for 50K, 8.7x for



(a) Comparing Vertical vs. Horizontal Scaling: GEMM Matrix Multiplication in Dask Local vs. Distributed



(b) Comparing Horizontal Scaling Options: GEMM Matrix Multiplication in Dask Distributed (Spot Instances and on demand VMs) and numpywren (Lambda) for different matrix sizes

Figure 27: Performance of Matrix Multiplication

70K).

We see in these experiments what can be achieved today with existing state-of-the-art Cloud infrastructure. Monetary cost is now the strongest reason for locality in Cloud providers as we see in the pricing models for Lambda, on demand VMs and Spot instances. But even if elastic disaggregated resources are now more expensive, some large scale compute intensive problems like linear algebra are now already competitive in compute time and scalability. Further improvements in cloud management control planes and locality-aware placement could reduce costs for elastic resources.

References

[1] Infinispan, "Infinispan." https://infinispan.org/.

[2] IBM, "IBM Cloud Functions." https://www.ibm.com/cloud/functions.

[3] IBM, "IBM Cloud Object Storage." https://www.ibm.com/cloud/object-storage.

[4] CloudButton Consortium, "CloudButton Toolkit implementation for IBM Cloud Functions and IBM Cloud Object Storage." https://github.com/pywren/pywren-ibm-cloud.

[5] SD Times, "Amazon Introduces Lambda, Containers at AWS re:Invent." https://infinispan.org/, 2014.

[6] AWS, "Amazon Step Functions." https://aws.amazon.com/step-functions/.

[7] Microsft Azure, "What are Durable Functions?." https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[8] OpenWhisk, "Apache OpenWhisk Composer." https://github.com/apache/incubator-openwhisk-composer.

[9] A. Airflow, "Apache Airflow documentation." http://airflow.apache.org/. Accessed on June 2019.

[10] Kubeflow, "Kubeflow: The Machine Learning Toolkit for Kubernetes." https://www.kubeflow.org/.

[11] Argo, "Argo Workflows & Pipelines: Container Native Workflow Engine for Kubernetes supporting both DAG and step based workflows." https://argoproj.github.io/argo/.

[12] Fission, "Fission WorkFlows." https://github.com/fission/fission-workflows.

[13] "Apache OpenWhisk." https://openwhisk.apache.org/.

[14] 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.

[15] J. Italo, "Facial mapping (landmarks) with Dlib + Python." https://towardsdatascience.com/facial-mapping-landmarks-with-dlib-python-160abcf7d672.

[16] 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/.

[17] 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/.

[18] 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.

[19] "Apache Mesos." http://mesos.apache.org/.

[20] Docker, "Docker, Swarm Mode." https://docs.docker.com/engine/swarm/.

[21] Kubernetes, "Kubernetes: Production-Grade Container Orchestration." https://kubernetes.io/.

[22] "Plasma Object Storage."

[23] N. W. Paton and O. Díaz, "Active database systems," ACM Computing Surveys (CSUR), vol. 31, no. 1, pp. 63–103, 1999.

[24] 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.

[25] 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.

[26] A. Geppert and D. Tombros, "Event-based distributed workflow execution with eve," in Middleware'98, pp. 427–442, Springer, 1998.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] S. Joyner, M. MacCoss, C. Delimitrou, and H. Weatherspoon, "Ripple: A practical declarative programming framework for serverless compute," arXiv preprint arXiv:2001.00222, 2020.

[35] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions," Future Generation Computer Systems, in press.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] CloudButton Consortium, "Deliverable D3.1: Initial specs of the Serverless Compute and Execution Engine." `https://cloudbutton.eu/docs/deliverables/CloudButton_D3.1_Public.pdf`.

[42] Python, "Multiprocessing." `https://docs.python.org/3/library/multiprocessing.html`.

[43] Python, "concurrent.futures." `https://docs.python.org/3/library/concurrent.futures.html`.

[44] Cloudbutton team, "Cloudbutton Toolkit." `https://github.com/cloudbutton`.

[45] CloudButton Consortium, "Deliverable D5.2: CloudButton Prototype of Abstractions, Fault-tolerance and Porting Tools."

[46] P. G. López, M. S. Artigas, S. Shillaker, P. R. Pietzuch, D. Breitgand, G. Vernik, P. Sutra, T. Tarrant, and A. J. Ferrer, "Servermix: Tradeoffs and challenges of serverless data analytics," CoRR, vol. abs/1907.11465, 2019.

[47] 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.

[48] CloudButton Consortium, "Deliverable D2.1: Experiments and Initial Specifications." `https://cloudbutton.eu/docs/deliverables/CloudButton_D2.1_Public.pdf`.

[49] "Knative:." https://knative.dev/, 2020.

[50] "Amazon Lambda:." https://aws.amazon.com/lambda/, 2020.

[51] "Google Cloud Functions:." https://cloud.google.com/functions, 2020.

[52] "Azure Functions:." https://azure.microsoft.com/en-us/services/functions/, 2020.

[53] "Knative Observability plugin:." https://knative.dev/docs/install/any-kubernetes-cluster/#installing-the-observability-plugin, 2020.

[54] "Metabolomics Usecase:." https://github.com/metaspace2020/pywren-annotation-pipeline, 2020.

[55] "Jupiter Notebook:." https://jupyter.org/, 2020.

[56] CloudButton Consortium, "Deliverable D2.3: CloudButton Architecture Specs and Early Prototypes." `https://cloudbutton.eu/docs/deliverables/CloudButton_D2.3_Public.pdf`.

[57] "Prometheus' HTTP API:." https://prometheus.io/docs/prometheus/latest/querying/api/, 2020.

[58] "ReconcilePackage:." https://godoc.org/sigs.k8s.io/controller-runtime/pkg/reconcile, 2020.

[59] "RabbitMQ Message Broker:." https://www.rabbitmq.com/, 2020.

[60] P. García-López, A. Slominski, S. Shillaker, M. Behrendt, and B. Metzler, “Serverless end game: Disaggregation enabling transparency,” arXiv preprint arXiv:2006.01251, 2020.

[61] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, “On the FaaS track: Building stateful distributed applications with serverless architectures,” in Proceedings of the 20th International Middleware Conference, pp. 41–54, 2019.

[62] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, “numpywren: serverless linear algebra,” 2018.