CloudButton

HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D3.1 Initial specs of the Serverless Compute and Execution Engine

Due date of deliverable: 30-06-2019
Actual submission date: 22-07-2019

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	52
WP/Task related to this document	WP3 / T3.1
WP/Task responsible	IBM
Leader	David Breitgand (IBM)
Technical Manager	Peter Pietzuch (Imperial)
Quality Manager	Josep Sampé (URV)
Author(s)	David Breitgand (IBM), Gil Vernik (IBM), Ana Juan Ferrer (ATOS), Amanda Gómez (URV), Aitor Arjona (URV), Gerard París (URV).
Partner(s) Contributing	IBM, ATOS, URV.
Document ID	CloudButton_D3.1_Public.pdf
Abstract	This document presents a design, specification, and initial prototypical implementation of CloudButton FaaS Compute Engine ("Core") and Execution Engine ("Toolkit")
Keywords	FaaS, serverless, Kubernetes, hybrid cloud, data intensive serverless, SLA/SLO management, cost-efficiency of data-intensive serverless computations, "servermix" model, workflow orchestration

History of changes

Version	Date	Author	Summary of changes
0.1	01-06-2019	David Breitgand; Gil Vernik	TOC
0.2	02-06-2019	David Breitgand; Gil Vernik	First Draft
0.3	03-06-2019	David Breitgand; Gil Vernik	Second Draft
0.4	06-06-2019	Amanda Gómez; Aitor Arjona; Gerard París	State of the Art on Workflow Orchestration. Airflow section.
0.5	18-06-2019	Ana Juan Ferrer	SOTA on SLA Management; Prototype
0.6	19-06-2019	David Breitgand; Gil Vernik	Third Draft: SOTA on FaaS, Workflows, etc
0.7	22-06-2019	David Breitgand; Gil Vernik	Fourth Draft: architecture; toolkit; initial prototype
0.8	22-06-2019	David Breitgand	First End-to-End Draft; Next steps section
0.9	24-06-2019	David Breitgand	Draft ready for peer-review: Bibliography; editing
1.0	27-06-2019	David Breitgand; Gil Vernik; Sadek Jbara; Avi Weit	Final version; proofreading

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 5 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].

In addition, initial experimentation with running PyWren over [4] over Knative [5] as a FaaS backend was performed. This work is not yet open sourced, but we expect to make it public within the next reporting period as we move forward with the overall architecture. We briefly describe this in Section 6.

Our progress on the functional blocks which are not yet integrated into the prototype, is reported in Section 6, which also outlines the next steps that will be taken in WP3 (in a joint effort with other WPs) to integrate these functionalities with the rest of the architecture to fully realize the CloudButton vision.

2 Motivation and Background

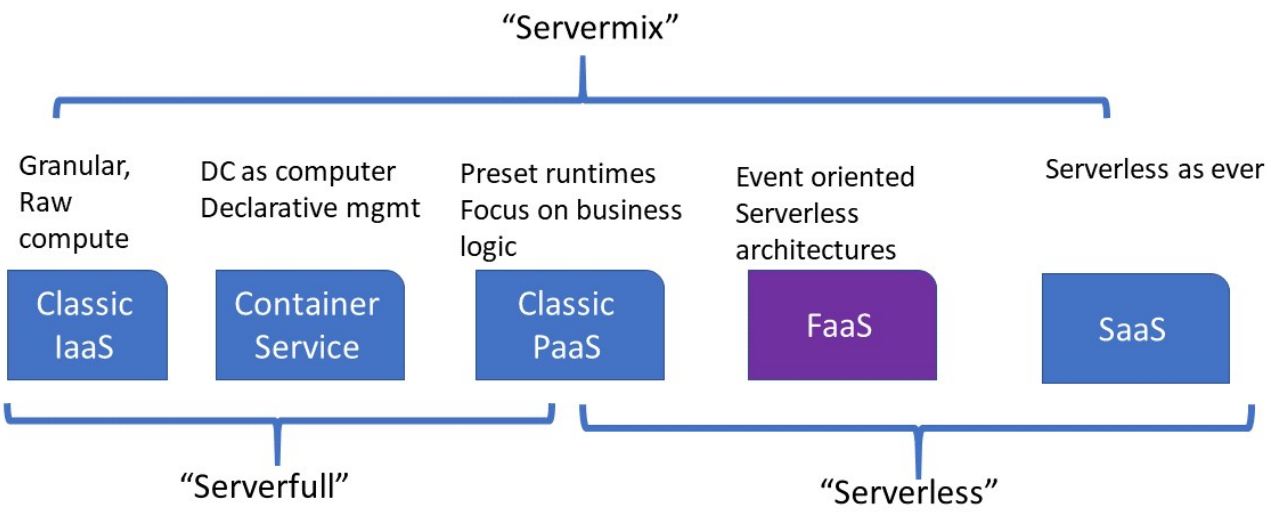

Figure 1: Serverless Taxonomy

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 [6]. 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 [7] and Azure Durable Functions [8] 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 [14] 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 [15], 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 [9] 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 [10], Kubeflow [11], Argo Flows [12], Fission Workflows [13]. 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 [16] 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 [17], 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 [18] 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 [19], 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 [20, 21, 22] with CNCF's Kubernetes (K8s) [22] 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 [12] and Kubeflow [11] 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) [23] 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 FaaS Frameworks

Since its inception in 2014, serverless computing gained a lot of attention. Multiple serverless frameworks have appeared over the last five years with ever more frameworks being developed constantly. A full survey of serverless platforms available today is out of the scope of this document. However, it should be stressed that the fluidity of the serverless technology landscape directly influences our design decisions (explained in Section 4.1). To ensure a lasting legacy and impact of the CloudButton project, the core technology that we develop should be inter-operable with today's platforms and the future ones that will undoubtedly appear throughout the project lifetime.

For an up-to-date overview of the serverless paradigm, see [24]. A recent performance evaluation comparing FaaS of the leading vendors in the public cloud can be found in [25]. Also, public on-line resources, e.g, [26] continuously evaluate performance of the public serverless offerings. In general, it can be argued that the public cloud based FaaS offerings are relatively well reported in the literature. Therefore, in this section, we focus on the open source FaaS frameworks that received less attention thus far (because of being new) and which are directly relevant to our CloudButton platform. We are specifically interested in evaluating serverless frameworks for K8s. In this study [27], some of the early contenders are surveyed. A more updated recent study [28] claims that currently there are five "serious" [*sic*] open source serverless projects competing for the K8s market share: Apache OpenWhisk, CNCF Knative, Kubeless, Fission, and OpenFaas. We briefly outline each of these frameworks. On May 9, 2019, another important serverless framework for K8s was announced, KEDA [29]. Therefore we add it to our brief survey below.

Finally, the other two frameworks we survey are Nuclio [30] because of its real time performance orientation and integration with Kubeflow Pipelines [31] (a K8s native ML workflows orchestrator being promoted by Google and aiming to become a CNCF project) and CloudFlare Workers [32], a public FaaS offering by CloudFlare because of it being "containerless" (the only such system today to the best of our knowledge) and its use of WebAssembly [33].

3.1.1 OpenWhisk

Apache OpenWhisk is an extensible serverless computing platform that supports functions (also known as “actions”) that can be written in multiple programming languages including Node.js, Python, Swift, Java, PHP, Go, and Rust. Also, OpenWhisk supports native binaries. With a native binary, any executable that is compatible with a standard OpenWhisk container may run as a serverless function. These functions are termed blackbox actions. Blackbox actions derive their container image from the base OpenWhisk container that includes some basic management services allowing the OpenWhisk framework to interact with the action.

In OpenWhisk, functions run within managed Docker containers that can be deployed on various target platforms including K8s. The platform's architecture supports Docker images for each language runtime. Using this approach, IBM Cloud Functions, which is powered by OpenWhisk, offers Node v8 with SDKs that is specific to the IBM Cloud portfolio, including Watson services and Cloud Object Storage and Python Docker container image with packages popular for numerical analysis and machine learning.

Apache OpenWhisk programming model is similar to other serverless frameworks. Apache OpenWhisk is an inherently event-driven framework. OpenWhisk allows to define events (“triggers”), and “rules” that connect the triggers to actions. Also, OpenWhisk supports sequential chaining of actions (“sequences”) with one action's output being another action's input. Apache Composer [9] allows to combine actions into arbitrary complex topologies with a stateful flow of execution, which is automatically supported by the compositions. A composition is an action in itself. Therefore the composition flow state is ephemeral.

OpenWhisk is a multi-tenant system. It supports user isolation at the level of name spaces and requires an API key to interact with resources. Ability to deploy on K8s allows makes OpenWhisk independent of the underlying cloud technology and achieves vendor neutrality. Deployment on K8s allows to leverage built in K8s:

- Support for networking;
- Actions discoverability as services;
- Placement policies;
- Specialized hardware utilization;
- Virtualization technology neutrality.

OpenWhisk allows programming artefacts to be grouped as *packages*, which can be shared. Furthermore, OpenWhisk development tools, such as wskdeploy, allow projects to be created that define multiple artefacts using a manifest and deploy and manage projects as units rather than dealing with multiple independent artefacts. An OpenWhisk distro with a small footprint (called Lean OpenWhisk [34, 35] is available.

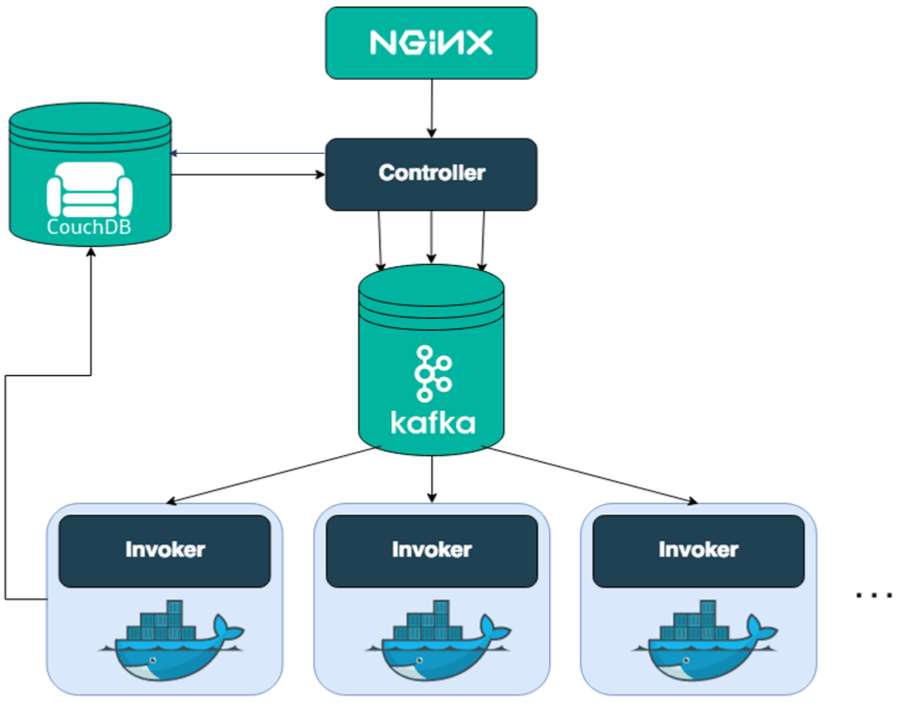

Figure 2: Apache OpenWhisk Architecture; *Source: [14]*

Figure 2 captures the main components of the Apache OpenWhisk architecture. At the core of the OpenWhisk project are two components: Controller and Invoker. The Controller performs load balancing among the invokers to distribute load evenly and to maximize the chances of hitting an Invoker node having a pre-warmed container for the action. The Controller publishes action invocation requests through the Kafka topics, to which Invokers subscribe. One of the following situations might occur when an action invocation request hits an Invoker node:

- There is no container to run an action and a new container should be created (this is termed a "cold start". A cold start might incur an invocation latency of 500-700ms).

- A pre-warmed non-initialized container exists. A pre-warmed non-initialized container is a container for a given run-time that has not yet been initialized with the specific action code. The base image of OpenWhisk Docker containers includes an HTTP service that serves two resources: `/init` and `/run`. The Invoker performs a REST call on the `/init` resource to "inject" the action code into the right place in the file system of the base image. Then the Invoker performs an HTTP POST request to the `/run` resource of the Docker container and the action starts executing. Initialization of a pre-warmed container is very fast and adds only a few tens of milliseconds.
- A pre-warmed initialized container exists for the action. This happens when a previous invocation of the action terminates. OpenWhisk does not terminate the container in anticipation that another invocation of the same action from the same user may arrive. If this indeed happens, the action invocation is almost instantaneous.

In 5G-MEDIA H2020 project [36], a federated architecture for OpenWhisk have been developed. Federation makes OpenWhisk suitable to the K8s-based hybrid cloud, an envisioned deployment environment for CloudButton.

3.1.2 Knative

Knative is a Custom Resource Definition (CRD) add-on for K8s. It is being developed by Google, Pivotal, IBM, Red Hat, and SAP. It is a set of open source components that facilitate building and deployment of container-based serverless applications that can be relocated across differed cloud providers and also across clusters in K8s federation.

Knative is using the market momentum behind Kubernetes to provide an established platform on which to support serverless deployments that can run across different public and private clouds. In that respect Knative aims at becoming a serverless platform of choice for K8s based hybrid cloud.

Many of the current serverless platforms are based on and tied to a specific cloud platform, which can lead to vendor lock-in for an organization adopting one of those platforms. Those include Amazon Web Services (AWS) Lambda, Microsoft Azure Functions, and Google Cloud Functions. This is what Knative purports to avoid, by providing a K8s-native blocks for building serverless applications and, therefore, making them portable by design.

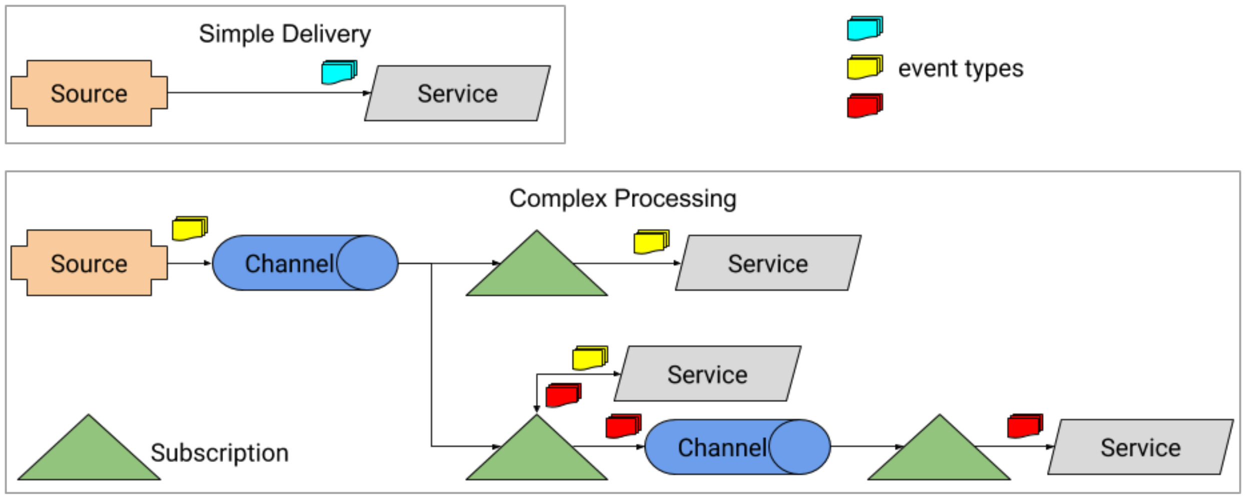

Figure 3: Knative Eventing; *Source: [37]*

The building blocks of Knative comprise:

- **Knative Serving:** facilitates rapid deployment and autoscaling of serverless containers. Essentially, Knative Serving is a special type of service, which can scale to 0 in absence of HTTP traffic. To get code deployed as a Knative Serving resource one has to (a) containerize the code code and push the container image to a registry and (b) create a service yaml file (i.e., service manifest) to specify to KNative where to find the container image and allow for its configuration, See Listing 1 for a simple example of a serverless function *autoscale-my-py-func*. To deploy *autoscale-my-py-func*, its yaml definition should be applied to the K8s API server, much like any other pod definition in K8s. Upon applying the yaml manifest, textitautoscale-my-py-func will obtain an IP address and port, by which it can be accessed and a routing to the version specified by the manifest (in this case, the latest one, will be created through Istio [38]. Specifying a little bit more sophisticated revision specification in the Knative Serving template allows for arbitrary allocation of the incoming HTTP traffic to different revisions, which is an enabling feature for gradual roll-out, canary tests, A-B deployment, etc. The traffic allocation among the revisions can be specified in per cents in the yaml definition of the serving resource.

A load balancer embedded with the K8s platform and an autoscaler for the deployed Knative serving revision(s) is automatically created upon applying the serving definition and the serving deployment shows up in the K8s list of deployments. However, until actual HTTP traffic to the serving resource occurs, no pod is being executed. Upon HTTP requests to the serving revision, a new pod is being created so that the average concurrency target per pod will be satisfied as specified in the yaml under the autoscaling annotations.

An interesting property of Knative serving, in fact the one, that makes it "serverless" is its ability to scale down to zero when no traffic to the serving is detected after some grace period, which is configurable. Also, it is possible to use different metrics for the autocaler, e.g., CPU or memory utilization. Likewise, it is possible to specify the minimum number of pods that should be up and running even in absence of the actual traffic. This feature might be useful for mitigating cold starts. For a recent study of the cold start problem evaluation, the reader is referred to [39].

- **Knative Eventing:** facilitates loosely coupled, event-driven services by allowing to define event channels (backed by persistent storage). The goal of Knative Eventing is to route events from a source to a service (one such consumer type can be Knative Serving described above). In Listing 2, we show a sample eventing definition that forwards all K8s events to Knative serving resource that consumes them.

  Knative Eventing architecture is shown in Figure 3. The primitives used to define Knative Eventing resource are: Source, Channel and Subscription. To ensure inter-operability between events producers and consumers, Knative Eventing is consistent with the CloudEvents specification [] that is being developed by the CNCF Serverless Work Group.

- **Knative Build**: facilitates "from code in git to container in a registry" DevOps workflows. There are four primitives related to Knative Build:
  - **Build:** Represents an in-cluster build job with one or more steps;
  - **BuildTemplate:** A set of ordered and parameterized build steps;
  - **Builder:** A container in a Build step that performs an action (e.g., builds an image)
  - **ServiceAccount:** used for authentication with DockerHub etc.

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: autoscale-my-py-func
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          containers:
            image: functions/autoscale-my-py-func:0.1
        spec:
            template:
                metadata:
                    annotations:
                        # Specifies 100 in-flight-requests per pod.
                        autoscaling.knative.dev/target: "100"
```

Listing 1: Sample yaml definition of Knative Serving resource

```
apiVersion: flows.knative.dev/v1alpha1
kind: Flow
metadata:
  name: k8s-event-flow
  namespace: default
spec:
  serviceAccountName: k8s-feed-service
  trigger:
    eventType: dev.knative.k8s.event
    resource: k8sevents/dev.knative.k8s.event
    service: k8sevents
    parameters:
      namespace: default
  action:
    target:
      kind: Route
      apiVersion: serving.knative.dev/v1alpha1
      name: read-k8s-events
```

Listing 2: Sample yaml definition of Knative Eventing resource

Knative is a relatively young project (it has been around for about one year). However, it enjoys a lot of attention because of the interest exhibited by the major commercial vendors, such as Google, who recently announced Knative based Cloud Run [40] beta and IBM who, announced an experimental Managed Knative on IBM Cloud Kubernetes Service [41]. Interesting open source projects, like Triggermesh [42] appeared recently developing Knative build templates that can be used to run an AWS Lambda function in a K8s cluster installed with Knative [42].

Presently, Knative is not a full featured serverless framework. Its level of abstraction is quite low for an average data scientist, because it requires a good command of K8s. Projects like Pivotal's Riff [43] offer to elevate the abstraction level through defining serverless functions (backed by the Knative resource definitions) via an added Riff layer that hides away complexities of Knative. However, currently Riff offers only a limited support for a handful of programming languages and container based actions (similar in the concept to the OpenWhisk's black box AKA Docker actions).

Given the community's interest in Knative, we believe that it will improve significantly throughout the CloudButton project duration and has a potential to become a de-facto platform for serverless computing in K8s. Hence we made a decision to add Knative to our target serverless frameworks to ensure the CloudButton lasting impact. The architectural way to achieve this is discussed in Section 4.

In the reminder of this subsection, we briefly overview other serverless frameworks for completeness.

3.1.3 Fission

Fission is conceptually similar to both OpenWhisk and Knative (albeit it appeared before Knative). A conceptual similarity to OpenWhisk is in the approach Fission (a) uses the function abstraction and not the pod abstraction and (b) to starting a new function instance. Similarly to OpenWhisk, Fission maintains a pool of "warm" containers, where each one contains a small footprint dynamic loader. When a function is first called, i.e. "cold-started", a pre-warmed container is chosen and the function is loaded. This pool is what makes Fission fast: cold-start latencies are typically about 100 msec.

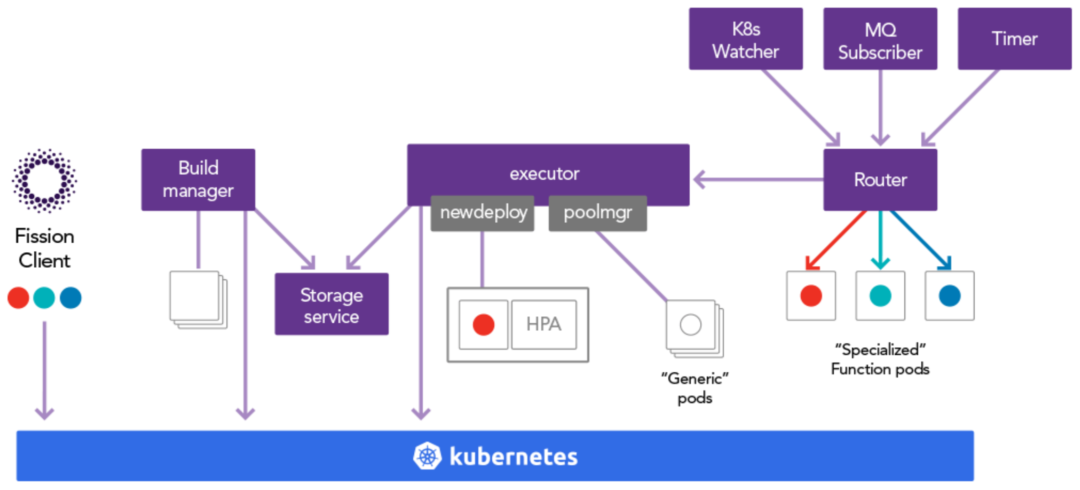

Figure 4: Fission Architecture; *Source: [44]*

Conceptually similar to Knative, Fission has a Build Manager to manage the functions devops cycle (even though release management in Fission is less sophisticated than in Knative).

One feature that sets Fission apart from the rest of the K8s based serverless framewirks is its own built-in support for workflows [13]. We discuss Fission Workflows in Subsection 3.2.2.

Fission is less integrated with the CNCF's ecosystem. Neither it appears to be on the track to become a CNCF project. The commercial uptake of Fission is limited at the moment and it remains to be seen whether this project will get sufficient traction during the CloudButton project lifetime or beyond this timeframe. Nonetheless, our architecture allows to incorporate Fission as a backend FaaS framework should this be deemed beneficial to the CloudButton ecosystem at a later stage.

3.1.4 Kubeless

Kubless [45] is an open source project driven by Bitnami. It was one of the first K8s-native serverless frameworks. Its initial motivation was to demonstrate feasibility of crossing over between serverless and K8s. In Kubeless, each function is deployed into a separate K8s deployment and K8s Horisontal Pod Autoscaler (HPA) is used to automatically scale function based on defined workload metrics. The default metric is CPU usage, but other metrics are possible to configure. To the best of our understanding, this project was inspirational to Knative and Fission and enjoyed high popularity in 2018, but the recent git statistics indicate that it might be past its peak of popularity and is likely to be superseded by the competing K8s native serverless projects.

3.1.5 OpenFaas

OpenFaaS started as an independent project by Alex Ellis in 2016. Initially started as a one person project, OpenFaas has received a warm welcome from the developers community appears to have attracted a growing community of users and developers. Conceptual architecture of OpenFaaS is shown in Figure 5. The main design objective of OpenFaaS is to be able to turn every container into a serverless function. In contrast to Fission and OpenWhsik, OpenFaaS does not use dynamic loading of the function code into pre-warmed non-specialized containers. Rather, each function code is burned into an immutable container image (that will serve invocations of just this function) and a special process, called *watchdog* is running in each OpenFaaS container. A new invocation request is forwarded to a function instance by the API Gateway. Upon receiving a request, the watchdog process forks a process that executes the function code and uses UNIX IPC (pipelines) to communicate input parameters and output results to and from the function. Recently OpenFaaS was enriched by scale to 0 capability [47].

Overall, OpenFaaS is still a very young project and is less mature than Apache OpenWhisk or Knative. However, it definitely has a potential to obtain a significant traction with the community

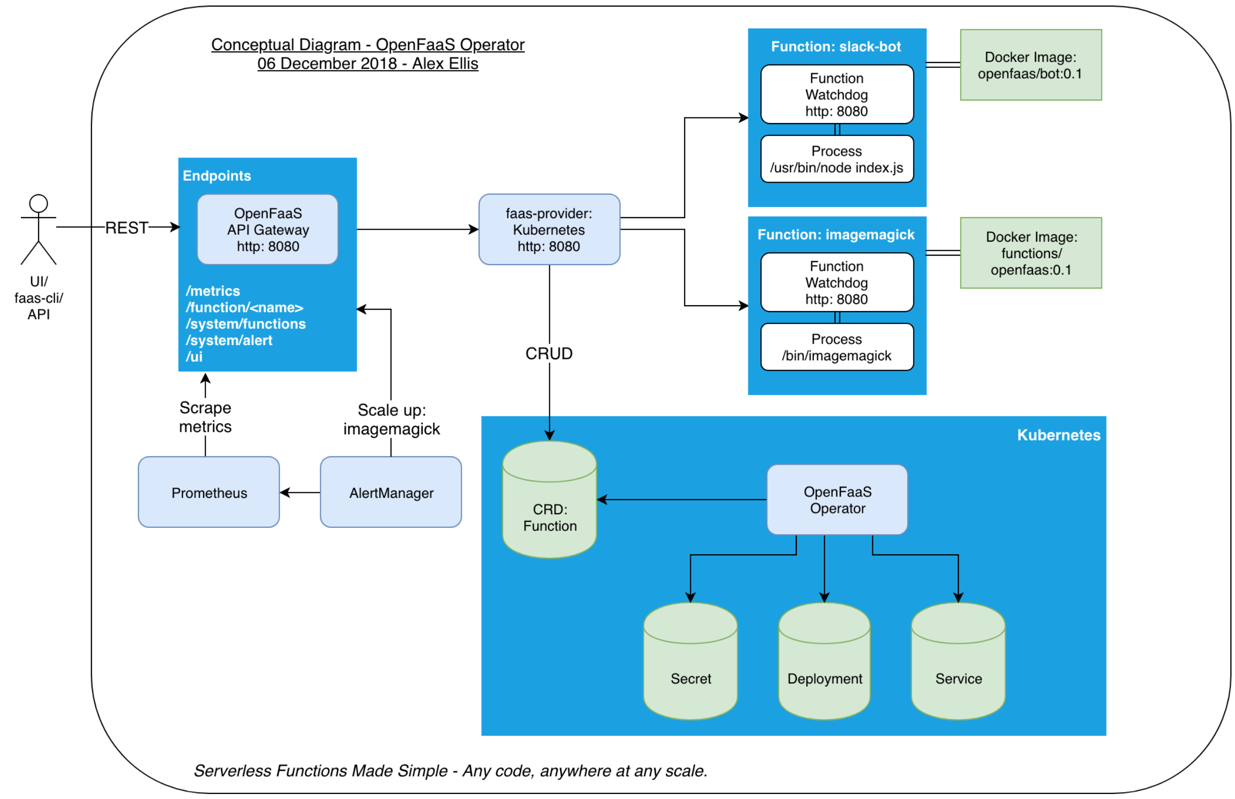

Figure 5: OpenFaaS Architecture; *Source: [46]*

and therefore should be watched closely throughout the CloudButton project duration.

3.1.6 KEDA

Kubernetes-based Event Driven Autoscaling (KEDA) [29] is a brand new component added by Microsoft and RedHat to the K8s ecosystem on May 2019. It provides event driven scale for any container running in K8s. KEDA is in its experimental phase and is not production worthy as is explicitly stated by its developers.

In Figure 6, KEDA's conceptual architecture is presented. KEDA enables any container to scale from zero to potentially thousands of instances based on event metrics like the length of a Kafka stream or an Azure Queue. This would allow for very efficient implementations. It also enables containers to consume events directly from the event source instead of decoupling with HTTP. KEDA can drive the scale of any container and is extensible to add new event sources.

KEDA's current focus is enablement of running Azure Functions efficiently on K8s. Because Azure Functions can be containerized, one can now deploy functions to any K8s cluster while maintaining the same scaling behavior of the Azure Functions cloud service. For workloads that may span the cloud and on-premises, one can now easily choose to publish across the Azure Functions service, in a cloud-hosted Kubernetes environment, or on-premises. The partnership with Red Hat enables Azure Functions to run integrated within OpenShift [48], providing the productivity and power of serverless functions with the flexibility and control to choose where and how to host it. The same application can move seamlessly between environments without any changes to development, code, or scaling.

While at this point, the project is very immature, in our appreciation, KEDA will rapidly improve and become important within the CloudButton's time frame.

3.1.7 Nuclio

Nuclio [30] is built with performance in mind. The key differentiating feature of Nuclio is its Function Worker component. Its implementation is optimized for real time processing by use of zero-copy,

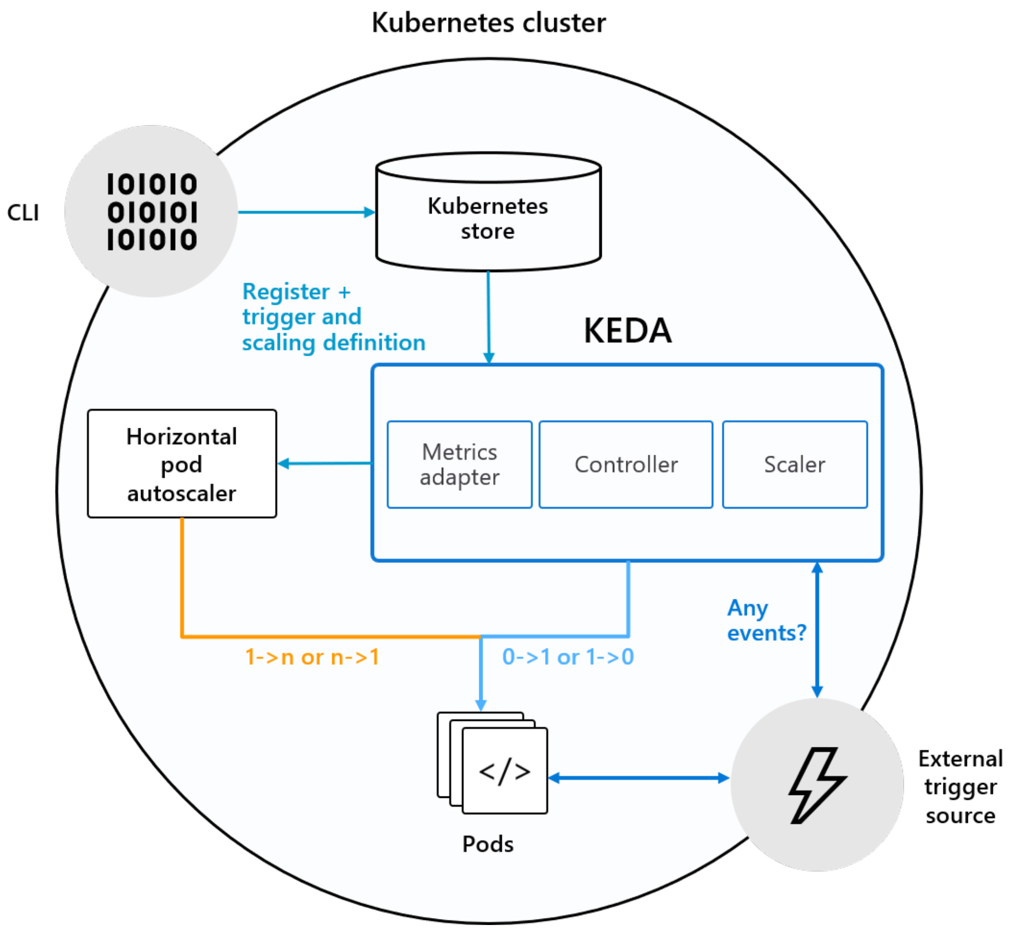

Figure 6: KEDA Architecture; *Source: [29]*

non-blocking IO, smart thread reuse, etc. A single nuclio function processor can run 400,000 function invocations per second with a simple Go function or up to 70,000 events per second using Python (PyPy) or node.js and respond in less than 0.1ms latency.

Nuclio supports four application modes: sync, async, stream and batch/interactive jobs, and dynamically distributes events, streams and job tasks among processors (the dealer). These make serverless applicable to new workloads including heavyweight backend and analytics tasks.

Nuclio functions can be called from within Kubeflow Pipelines [31]. These features of Nuclio make it an interesting project to follow while working on our CloudButton architecture.

3.1.8 CloudFlare Workers

CloudFlare Workers is not an open source project. Neither it targets K8s. Cloudflare Workers is modeled on the Service Workers [49] available in modern Web browsers.

The Service Worker API allows intercepting any request which is made to a Web site. Once a JavaScript is handling the request, any number of subrequests to this site or other sites can be spawned, and any response can be returned as a rsult.

Unlike standard Service Workers, Cloudflare Workers runs on Cloudflare's servers, not in the user's browser. Internally, Cloudflare uses the same V8 JavaScript engine which is used in the Google Chrome browser to run Workers on its infrastructure. V8 dynamically compiles JavaScript code into ultra-fast machine code, making it very performant. This makes it possible for the Worker code to execute in microseconds, and for Cloudflare to execute many thousands of Worker scripts per second.

WebAssembly [33] is a new feature of the Open Web Platform. It's a bytecode targeted, a new language layer that offers better, more predictable performance for applications that need it. It enables

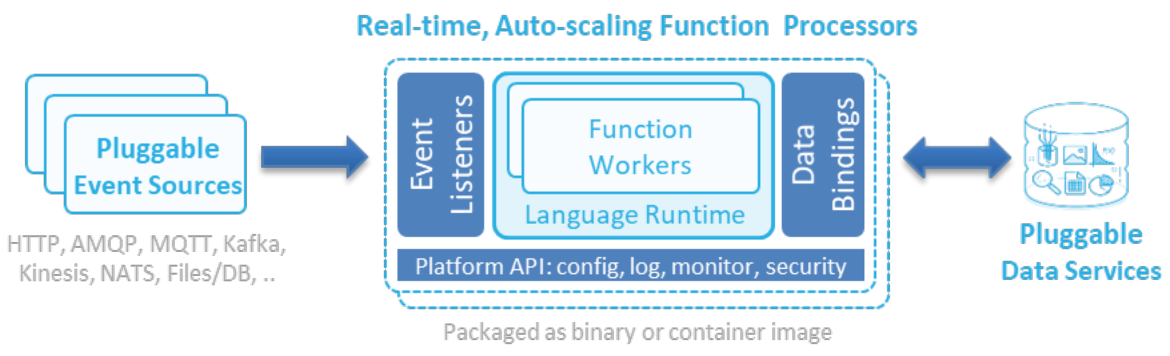
Real-time, Auto-scaling Function Processors
Pluggable Event Sources
HTTP, AMQP, MQTT, Kafka, Kinesis, NATS, Files/DB, ..
Event Listeners
Function Workers
Language Runtime
Data Bindings
Platform API: config, log, monitor, security
Packaged as binary or container image
Pluggable Data Services

Figure 7: Nuclio Architecture; *Source: [30]*

applications that were previously excluded from the Web to leverage it.

Cloudflare Workers runs in *V8 isolates*, which support both WebAssembly and JavaScript. Languages such as C, C++, AssemblyScript, and Go will eventually be supported by CloudFlare, but currently, its documentation focuses on Rust.

CloudFlare represents an interesting new development. With over 150 data centers worldwide, CloudFlare might become a competitor on FaaS to the more established vendors (even though collaboration is not precluded). The two defining features: (a) WebAssembly support and (b) V8 isolates instead ofcontainers for functions, set CloudFlare Workers apart. It remains to be seen whether CloudFlare workers will be supported in other environments. One straightforward development that can be expect is appearance of a Knative runtime environment to run CloudFlare Workers on K8s.

3.2 Workflow Orchestration

In the last years, efforts have been made in order to develop orchestration systems for serverless functions within the FaaS model. This section presents some of them and the state of the art in FaaS orchestration.

3.2.1 Vendor orchestration systems

Major vendors of Cloud services including Amazon, IBM and Azure have developed orchestration systems for their serverless functions in the recent years. These orchestration systems serve as coordination mechanisms between functions to create and orchestrate complex applications that follow workflow structures. We will explain their characteristics and limitations with information obtained from their websites and the study conducted by García-López et al. in their paper *Comparison of FaaS Orchestration Systems* [50]. This piece of work studied the orchestration systems in terms of their architectures, programming and billing models, and their effective support for parallel execution, among others.

Amazon Step Functions (ASF): ASF was the first project attempting to provide a composition service for Cloud functions. It relies on defining state machines as JSON Objects in a Domain Specific Language (DSL) called Amazon States Language. Also, it offers a graphical interface to implement the state machines.

Each of the states in the state machine contains a set of fields that describe the next state to run when current state finishes, the state type, if it is a terminal state and its input and output blocks of information. The next state field allows for function chaining and the state types define the actions and logic expected from the state, mainly:

- Pass: this type of state passes the input information to output without performing any work.
- Task: an unit of work is performed, like executing a FaaS function.

- Choice: this type of state allows for branching with *if* statements.
- Wait: the state machine execution is delayed for a specific amount of time.
- Succeed: it represents a terminal state that stops an execution successfully.
- Fail: this type is also a terminal state but it marks the stopping of the execution as a failure.
- Parallel: this kind of state is used to generate parallel branches that execute concurrently. The transition to the next state is done when all parallel branches finish their execution.

Figure 8 shows an example of state machine implemented in ASF by using its graphical interface. We have indicated the state type for each action in the workflow.

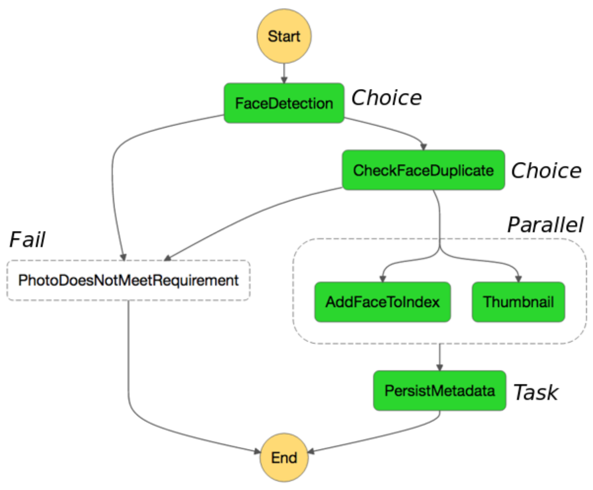

Figure 8: Image Processing with ASF. *Source: [51]*

Finally, the architecture of ASF is based on a client scheduler built as an external system separated from the FaaS platform. The client scheduler controls state transitions and logs each of them in order to follow the state machine situation at every point in time, while in each transition the previous state is recovered from the log and the next state is run.

IBM Composer: IBM's first attempt to compose functions was *IBM Sequences*, a mechanism that allowed to chain functions to create simple sequences of actions. Later, it presented IBM Composer (now Apache OpenWhisk Composer), a programming model for composing Cloud functions that provided new composition patterns apart from the sequences.

IBM Cloud functions platform is built on top of Apache OpenWhisk, and Composer utilizes OpenWhisk *conductor actions* as the basis to implement compositions. Conductor actions consist on special actions that embed logic to determine a flow of actions to be taken. The flow of actions is represented through steps and, for each step, the conductor action determines the action that needs to be performed. Listing 3 presents an example of a conductor action that defines a composition with a first step invoking *triple* action, a second step invoking the *increment* action and a last step returning the results.

```
function main(params) {
    let step = params.$step || 0
    delete params.$step
    switch (step) {
        case 0: return { action: 'triple', params, state:{ $step: 1 } }
        case 1: return { action: 'increment', params, state:{ $step: 2 } }
        case 2: return { params }
    }
}
```

Listing 3: OpenWhisk conductor action

Composer is a Javascript library that allows to create conductor actions that will later be executed in the OpenWhisk core platform. The library provides a variety of functions to build compositions, called *combinators*, such as:

- action: creates a composition with a single action. The action is invoked with the input data and the output is returned.
- sequence: used for chaining a group of actions, so each action is executed after its preceding action has finished.
- while: this combinator requires an action and a condition as parameters. It allows to repeatedly run the action while the condition evaluates to *true*.
- if: in this case the combinator requires a condition as parameter as well as the actions to invoke in case the condition evaluates to *true* and *false*. This function serves to branching purposes.

The combinators listed represent a subset of all the combinators offered by the library and new combinators were added in January 2019, including support for parallel executions. As of parallel compositions, Composer did not support them previously because conductor actions are not prepared for these type of works. For this reason, the offered parallel constructs rely on a Redis instance in order to save intermediate results and control the finalization of parallel executions. To do so, the OpenWhisk platform uses a blocking mechanism against Redis to deal with the execution and ending of conductor actions. New parallel combinators include:

- parallel: it takes a single input value and invokes a series of compositions in parallel over the value. The compositions are executed simultaneously over the same data.
- map: this combinator requires an array of values as input so for each element in the array, a sequence of compositions is invoked. The parallelism in this case is between the executions for each value of the array.

Listing 4 shows the necessary code in order to implement a composition that uses the *sequence* and *if* combinators.

```javascript
// Sequence definition
const composer = require('openwhisk-composer')
module.exports = composer.sequence(action1, action2, ...)

// Conditional definition
const composer = require('openwhisk-composer')
module.exports = composer.if(
  composer.action('authenticate', { action: function ({ password }) {
    return { value: password === 'abc123' }
  } }),
  composer.action('success', { action: function () {
    return { message: 'success' }
  } }),
  composer.action('failure', { action: function () {
    return { message: 'failure' }
  } }))
```

Listing 4: IBM Composer code for composition definition

With regard to the architecture, conductor actions are integrated into the core of the platform (i.e. OpenWhisk) and also the system includes a mechanism that relies on events to reduce latency between function invocations named *active ack*. This *active ack* mechanism bypasses the normal functioning of the system by directly forwarding results to the controller in order to reduce overheads, although it is treated as a secondary mechanism and not fully relied by the platform. Moreover, parallel executions require an external system (i.e. Redis) and blocking strategies.

Azure Durable Functions (ADF): ADF provides functionality to define workflows on top of Azure Functions. Workflows are defined programmatically using C# or Javascript languages.

Workflows are represented by a function type called *orchestrator function* that contains the code defining a workflow with the *async/await* constructs. After the orchestration functions is defined, an orchestrator client is in charge of starting and stopping its execution. The programming model in this case supports the following patterns, among others:

- Function chaining: this pattern chains a sequence of functions so they are executed in a specific order.
- Fan-out/fan-in: allows for the execution of functions in parallel and blocks the execution until all parallel functions have finished.
- Async HTTP APIs: provides a solution for the problem of coordinating long-running executions. Usually, an HTTP trigger is used to invoke actions and also an endpoint is exposed so the status of the long-running execution can be queried.
- Human interaction: allows to interact with human actions by using the orchestration function to wait for external events.

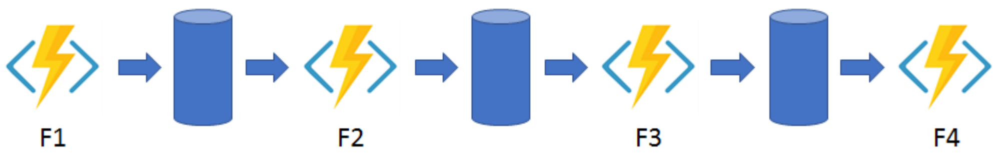

Figure 9: Function chaining with ADF. *Source: [52]*

The following listing is an example of function chaining by using the *await* construct to coordinate the sequence of functions from Figure 9.

```
public static async Task<object> Run(DurableOrchestrationContext context)
{
    try
    {
        var x = await context.CallActivityAsync<object>("F1");
        var y = await context.CallActivityAsync<object>("F2", x);
        var z = await context.CallActivityAsync<object>("F3", y);
        return  await context.CallActivityAsync<object>("F4", z);
    }
    catch (Exception)
    {
        // Error handling or compensation goes here.
    }
}
```

Listing 5: ADF code to compose a sequence

Regarding the architecture, the ADF platform is an extension of the core system of Azure and enables *event sourcing* by storing the events produced by invocations and using them to reconstruct state.

3.2.2 Fission Workflows

Fission is a serverless platform built on top of Kubernetes that allows to run functions as services within a Kubernetes-managed cluster so users can focus on the code rather than on managing the infrastructure [44]. Moreover, it offers events as triggers of functions through NATS [53] message queues, mapping queue topics to functions.

Fission Workflows is a framework for serverless function composition built on top of Fission. It allows to define workflows as YAML documents that are executed as dependency graphs. It offers a set of built-in flow control functions (foreach, if, repeat, switch, while, etc.) and, in addition, the user can create its own functions for flow control, offering wide flexibility.

Fission Workflows system relies on a set of components that interact with each other in order to orchestrate executions. The following list presents a high level description of these components.

- Event Store: The system architecture [54] includes an Event Store that consists on a NATS Streaming message queue. NATS is an open-source messaging system hosted as an incubation project by the CNCF [53]. The Event Store component is used to model the current state from arriving events and permits Event Sourcing techniques. Types of events received include the creation, deletion and invocation of workflows or the creation, completion, cancellation and failure of invocations and tasks.
- Projector: In order to construct the current state from events arriving to the Event Store, a component called Projector maintains objects in a cache, updating them by applying arriving events to the objects. These objects are called Entities.
- Controller: Once the current state is updated, it can be used by the Controller component to decide whether action is needed. The Controllers keep track on state updates through two different means: they receive a notification from the Projector every time the cache is updated and also the Controllers consist on a short and a long control loop that checks current states and updates on the cache [55]. Moreover, the Controller will execute the instructions sent by the Scheduler.
- Scheduler: when the Controller needs to evaluate the new states, it performs a call to the Scheduler providing it with the invocation information and the workflow definition. The Scheduler component analyses the data and returns the set of actions to be taken by the Controller, that include invoking or aborting functions, among others.
- API: the API exposes the functionality of the system and is the only component appending events to the Event Store, so it may be used by other components that need to append events.
- API Server: it exposes a subset of the API over HTTP, so it can be used by CLI or UI systems.
- Fission API / Proxy: it serves as an interface with Fission platform and allows to present the workflow engine as a Fission function in front of the platform.

Functions within Fission Workflows can be executed in one of two different function environments: Fission and Internal [56]. Fission execution environment is the underlying serverless platform, so the execution of functions is delegated to the cluster. The Internal environment is used for the execution of lightweight functions within the workflow engine, inside a limited and lightweight runtime. Using the Internal environment avoids the overkill overhead of running small functions as services. However, this approach entails some risks, as there are no sandboxing nor autoscaling options, so the workflow engine might be slowed down if a lot of computation is performed. All built-in flow control functions are run in this environment by default.

Fission Workflows life cycle Figure 10 shows the relationship between components of the system during the execution of a workflow. Red background circles indicate blocking calls. Whenever a user submits a new workflow to the system through the API Server (1), its information is stored in the cache system. After that, the workflow can be invoked and its execution will be controlled by the system components.

First of all, the API will append the events to the Event Store, for example the first event Invoke Workflow that starts its execution (2). The events arriving to the Event Store serve to control the execution of the workflow, which is done by the Controller in two different forms:

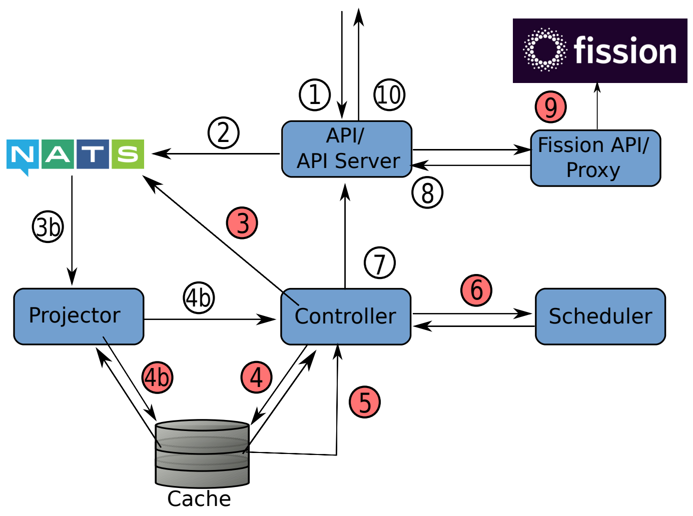

Figure 10: Fission Workflows life cycle overview

a The Controller consists on two Control Loops: the Long Control Loop checks the Event Store for active invocations (3) and refreshes the cache that stores the state of each invocation as Entity objects (4); the Short Control Loop checks the cache for updates (4).

b The Controller receives notifications from the Projector: the Projector is registered to the arrival of new events to the Event Store (3b), so when it receives an event it is used to update the Entity model in the cache and an update notification is sent to the Controller (4b). It must be noted that notifications from the Projector are only a "fast path" for communicating state updates and can fail. In case they fail, the Short Control Loop is the mechanism to recover the updates and evaluate the new state.

Whenever the Controller determines that action is needed after checking the cache or receiving a notification, it performs an evaluation that consists on retrieving data and sending it to the Scheduler to obtain a Scheduling plan. The data is retrieved from the cache and includes the workflow definition and the invocation information (5). The Scheduler evaluates the current state of the invocation and based on the workflow definition returns the instructions for the Controller (the scheduling plan) (6). Next, the Controller will execute the actions in the plan and in case the action needed is an invocation:

(a) If the invocation corresponds to a built-in control flow function (i.e. if, foreach...) it will be executed in the Internal environment, that is, in the workflow engine itself and specifically within the process in charge of invoking.

(b) Otherwise, the function will be invoked through the API to the Fission platform as an HTTP request (8). This invocation is synchronous as the process blocks until there is a Fission function response (9).

Finally, the execution of these actions by the Controller will generate new events in the Event Store (2).

The life cycle presented is the general logic for the execution of a workflow. However, different types of steps within the workflow require different specific actions by the system. Next, we present the differences in the life cycles of sequence, branching and parallel steps. The general life cycle is shared by all of them, but there are specific mechanisms from point (6).

Simple sequence life cycle: in this case, when the controller determines that action is needed and requests a scheduling plan, the Scheduler directly returns the instruction to execute the next action in the sequence, usually within the Fission platform. The number of blocking calls identified in this case is 4 ((3),(4),(5),(9)).

Branching execution life cycle: branching steps entail a bit more of complexity as the system needs to determine what action to execute. In this case, the Scheduling plan includes the invocation of the *if* internal function within the Internal environment. When the *if* function is executed, it returns the branch to be taken as a new *Task*. The new *Task* is stored in the system as a new workflow that needs to be executed and controlled, involving a new iteration in the life cycle. This type of step involves 3 blocking calls when determining the action to take (*if* function execution)((3),(4),(6)) and 4 calls for the execution of the specific action ((3),(4),(5),(9)), that makes a total of 7 possible blocking calls per branching step.

Parallel execution life cycle: the mechanism for parallel steps is similar to the branching steps. First of all, the Scheduling plan determines that *foreach* internal function should be executed. The result of its execution is a new workflow that includes the N parallel tasks. The next step is to store the workflow in the system and iterate in the life cycle. In this case, the number of blocking calls is 3 (*foreach* function execution), plus an other 3 ((3),(4),(6)) to schedule the execution of the parallel tasks and N points in (9), one per each task N that should be run in parallel. The total number of potential blocking calls is $N + 6$.

3.2.3 Limitations

The orchestration systems offered by Amazon, IBM and Azure were evaluated in terms of their runtime overhead by García-López et al. [50]. They considered as overhead the difference between the total execution time of the composition and the effective execution time of the functions being composed. First of all, they evaluated the overheads of composing sequences of actions with compositions of lengths

$$n \in 5, 10, 20, 40, 80$$

However, compositions of length *80* are not allowed in IBM Sequences and IBM Composer, as the limit of actions in a composition is 50 actions, so there are no results for this case. The complete results for this type of composition are shown in Figure 11. As can be seen, the lowest overhead corresponds to IBM Sequences, while IBM Composer and Amazon Step Functions have similar overheads and Azure Durable Functions presents the highest ones. In general, overheads tend to grow linearly when the number of actions composed increments. Furthermore, the study states that Composer is not prepared for the execution of long-running workflows due to their limit of 50 actions in compositions and the fact that the orchestration can not be suspended to be resumed later.

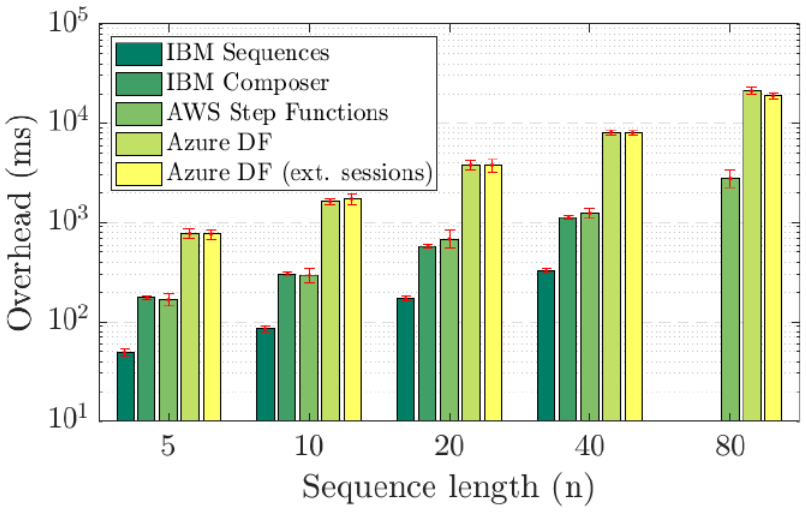

Figure 11: Function sequences overhead. *Source: [50]*

Secondly, the study evaluated the overhead of running parallel executions by experimenting with

$$n \in 5, 10, 20, 40, 80$$

actions running in parallel. Also, the action had a fixed time duration of 20 seconds. In this case, the overhead was calculated as the difference between the total execution time of the composition and the fixed time duration of the action. At the time of their analysis IBM Composer did not support parallel executions, so the results correspond to ASF and ADF. As shown in Figure 12 the overheads grow exponentially with the increase in the number of parallel functions. Starting at barely 300 millisecond overhead in both systems for 5 parallel functions, it increases to over 18 seconds in ASF and 32 seconds in ADF for 80 functions. What is more, the overhead of ADF presents high variability.

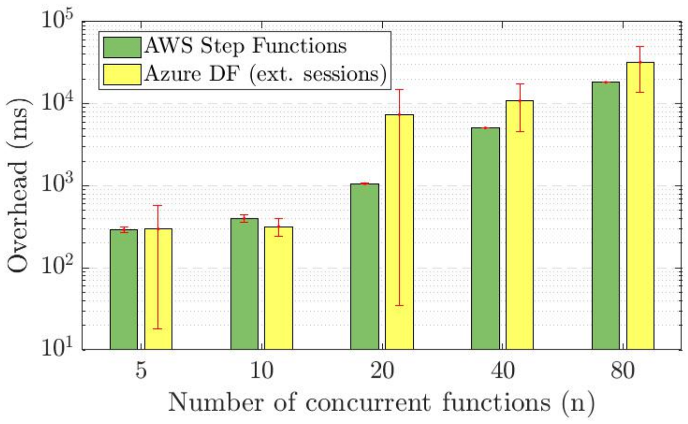

Figure 12: Parallelism overhead. *Source: [50]*

Support for parallel executions was added to Apache OpenWhisk Composer in January 2019, so we studied the overheads of these parallel compositions and compared them to ASF and ADF orchestration overheads. Figure 13 depicts the observed overheads. The main insight obtained is that Composer presents the lowest overheads with the highest degree of parallelism. However, it presented great variability in our experiments, with the overheads ranging from 2 to 12 seconds with 80 functions in parallel. While other vendor orchestration systems overheads grow exponentially, Composer is able to perform the execution with lower overheads but is also very variable.

On the other hand, Composer mechanism in charge of orchestrating parallel compositions is not integrated within conductor actions and the underlying platform. In order to orchestrate this type of composition, Composer needs a Redis database to store intermediate results and also spawns a function that blocks against Redis to determine if the parallel execution has finished.

To sum up, limitations of current vendor orchestrating systems include the overhead in parallel executions and the blocking mechanisms in order to control the orchestration.

The following list presents current limitations of the systems studied, including the vendor orchestration systems and Fission Workflows:

- Overheads in parallel executions: ADF and ASF present overheads that grow exponentially with the level of parallelism. At the moment, these systems are not prepared for parallel programming.
- Blocking mechanisms: the mechanism underneath Composer's parallel compositions relies on blocking patterns and is not embedded in the underlying architecture. On the other hand, ASF relies on a blocking client scheduler to orchestrate compositions. Regarding Fission Workflows, we have identified several blocking calls when orchestrating workflows.
- Long running jobs: due to the fact that some systems rely on blocking mechanisms, long-running compositions require the use of resources during prolonged periods of time. Also, tasks can not be suspended and later resumed on a certain event or occurrence.

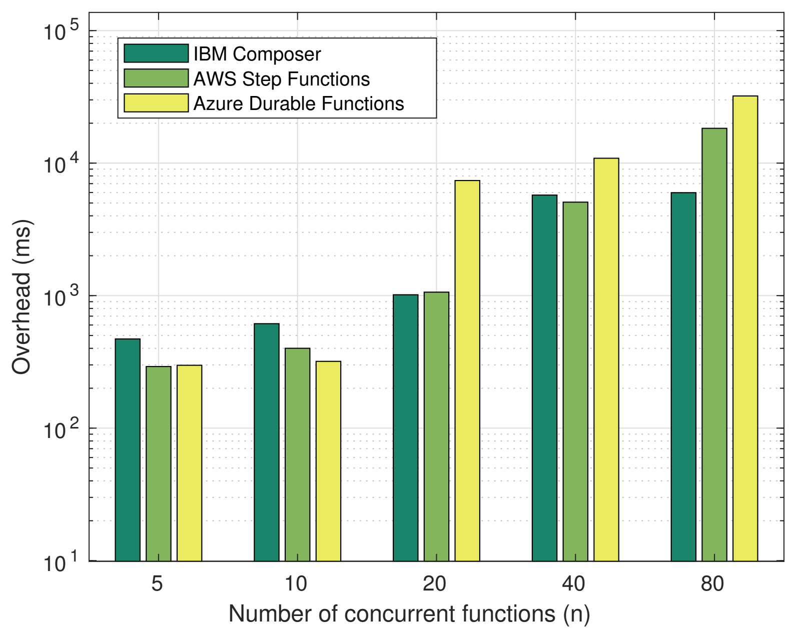

Figure 13: Parallelism overhead

- Scalable core: in the case of Fission Workflows, the orchestration engine relies in the execution of flow control functions within the Internal environment, with no sandboxing nor autoscaling options. For this reason, the orchestration engine might be slowed down if a lot of computation is performed.
- Limitations in compositions: IBM Composer limits the number of actions within a composition, so it is not designed for orchestrations with a long number of stages.
- Portability to other Cloud platforms: the vendor orchestration systems studied are developed for specific Cloud platforms and entangled in their architectures. As a result, they do not follow a model portable between the different vendors' platforms.
- Billing model: with regards to the three orchestration systems studied, García-López et al. concluded that neither ADF nor IBM Composer billing models are clear. Charges could include function execution times and storage costs, but they are not predictable.

3.2.4 Kubeflow Pipelines

Kubeflow Pipelines is a platform for building and deploying portable, scalable machine learning (ML) workflows based on Docker containers. The Kubeflow Pipelines framework comprises:

- A user interface (UI) for managing and tracking experiments, jobs, and runs;
- An engine for scheduling multi-step ML workflows (represented as a DAG of tasks);
- A software development kit (SDK) for defining and manipulating pipelines and components;
- Notebooks for interacting with the system using the SDK.

The following are the goals of Kubeflow Pipelines:

- End-to-end orchestration: enabling and simplifying the orchestration of machine learning pipelines;
- Easy experimentation: making it easy for a data scientist to try numerous ideas and techniques and manage various trials and experiments;

- Easy re-use: enabling a data scientist to re-use components and pipelines to quickly create end-to-end solutions without having to rebuild each time.

In Kubeflow v0.1.3 and later, Kubeflow Pipelines is one of the Kubeflow core components. It's automatically deployed during Kubeflow deployment.

```python

@dsl.pipeline(
  name='XGBoost Trainer',
  description='A trainer that does end-to-end distributed training for XGBoost models.'
)
def xgb_train_pipeline(
    output,
    project,
    region='us-central1',
    train_data='gs://ml-pipeline-playground/sfpd/train.csv',
    eval_data='gs://ml-pipeline-playground/sfpd/eval.csv',
    schema='gs://ml-pipeline-playground/sfpd/schema.json',
    target='resolution',
    rounds=200,
    workers=2,
    true_label='ACTION',
):
  delete_cluster_op = DeleteClusterOp('delete-cluster', project, region).apply(gcp.use_gcp_secret('user-gcp-sa'))
  with dsl.ExitHandler(exit_op=delete_cluster_op):
    create_cluster_op = CreateClusterOp('create-cluster', project, region, output).apply(gcp.use_gcp_secret('user-gcp-sa'))

    analyze_op = AnalyzeOp('analyze', project, region, create_cluster_op.output, schema,
                           train_data, '%s/{{workflow.name}}/analysis' % output).apply(gcp.use_gcp_secret('user-gcp-sa'))

    transform_op = TransformOp('transform', project, region, create_cluster_op.output,
                               train_data, eval_data, target, analyze_op.output,
                               '%s/{{workflow.name}}/transform' % output).apply(gcp.use_gcp_secret('user-gcp-sa'))

    train_op = TrainerOp('train', project, region, create_cluster_op.output, transform_op.outputs['train'],
                         transform_op.outputs['eval'], target, analyze_op.output, workers,
                         rounds, '%s/{{workflow.name}}/model' % output).apply(gcp.use_gcp_secret('user-gcp-sa'))

    predict_op = PredictOp('predict', project, region, create_cluster_op.output, transform_op.outputs['eval'],
                           train_op.output, target, analyze_op.output, '%s/{{workflow.name}}/predict' % output).apply(gcp.
                                use_gcp_secret('user-gcp-sa'))

    confusion_matrix_op = ConfusionMatrixOp('confusion-matrix', predict_op.output,
                                            '%s/{{workflow.name}}/confusionmatrix' % output).apply(gcp.use_gcp_secret('user-gcp-sa'))

    roc_op = RocOp('roc', predict_op.output, true_label, '%s/{{workflow.name}}/roc' % output).apply(gcp.use_gcp_secret('user-gcp-sa'))
```

Listing 6: Kubeflow Pipeline: XGBoost Model Training on GCP with the CSV Data (*Source [31]*)

A pipeline component is a self-contained set of user code, packaged as a Docker image, that performs one step in the pipeline. For example, a component can be responsible for data pre-processing, data transformation, model training, and so on. Kubeflow Pipelines are integrated with Nuclio: it is possible to call Nuclio functions seamlessly from within the pipeline. A conceptually similar design was tested by us in the context of integrating Airflow [10] with PyWren as described in Subsection 5.3.1.

Kubeflow Pipelines allows a data scientist to write code in Python without worrying neither about containers configuration, nor their deployment nor orchestration. The code in Python is automatically translated into the DAG of tasks. Listing 6 shows the Python code for a sample ML pipeline that automates training of an XGBoost model in Python [57] on Google Cloud Platform (GCP) using CSV data. Figure 14 shows how the GUI of Kubeflow Pipelines represents the same Python code as the DAG of tasks. The DAG is extracted automatically from the source code without bothering the data scientist with the gory details of function decomposition.

Kubeflow causes a lot of interest in the K8s oriented ML community and rapidly becomes a tool of choice for the data scientists working in a hybrid cloud. The Kubeflow ambition is to become on par with the more established Airflow, but offer an efficient K8s native implementation and a focus on the ML workflows[5].

[5]It should be noted that even though at the moment Kubeflow is focusing on ML, it is a general tool that can be used in

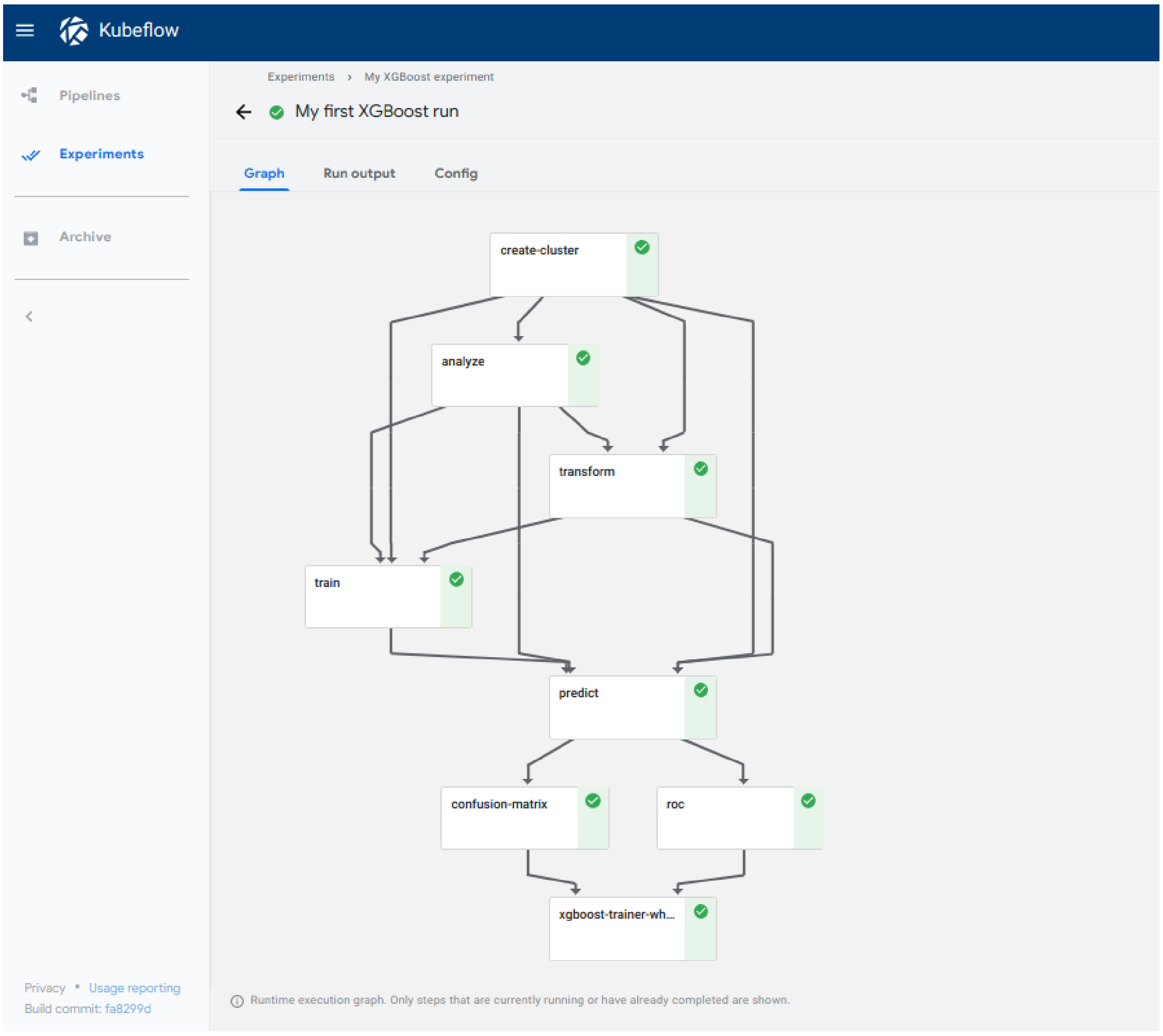

Figure 14: Kubeflow Pipelines GUI Representation of Listing 6 (*Source [31]*)

Under the hood, Kubeflow is powered by CNCF Argo Workflows & Pipelines [] project [12]. In fact, the ML pipelines of Kubeflow Pipelines are translated into Argo templates, which are YAML definitions rendered at the K8s level of abstraction in terms of K8s resources, such as pods. The actual orchestration of the flow is then performed by Argo. Argo is important to our proposed reference and software architectures discussed in Section 4. Because of Argo's importance, we discuss it separately in Subsection 3.2.5.

3.2.5 Argo Workflows & Pipelines

Argoproj comprises Argo Workflows [12], Argo CD [58], and Argo Events [59].

Argo Workflows is an open source container-native workflow engine for orchestrating parallel jobs on K8s. Argo Workflows is implemented as a K8s CRD (Custom Resource Definition). Argo Workflows allows to:

- Define workflows where each step in the workflow is a container;

the future for all kinds of general tasks and not necessarily as the Kubeflow developers envision it – i.e., it can be expected that the Kubeflow's focus will be expanded to encompass any tasks orchestration.

- Model multi-step workflows as a sequence of tasks or capture the dependencies between tasks using a directed acyclic graph (DAG);
- Easily run compute intensive jobs for machine learning or data processing in a fraction of the time using Argo Workflows on Kubernetes.
- Run CI/CD pipelines natively on Kubernetes without configuring complex software development products.

Argo Workflows is designed from the ground up for containers in K8s. Its primary goal is to orchestrate highly parallel jobs on K8s. Argo is cloud-agnostic and can run on any K8s cluster making the workflows defined with Argo workflows portable from Day 1, an important feature for the hybrid cloud deployment. Although not available as an out of the box feature, Argo has been demonstrated recently to run efficiently across multiple K8s clusters [60]. Argo Workflows support a rich set of features: DAG or Steps based declaration of workflows, Artifact support (S3, Artifactory, HTTP, Git, raw), Step level input and outputs (artifacts/parameters), Loops, Conditional Branching, Parameterization, Timeouts (step and workflow level), Retry (step and workflow level), Resubmit from the last memorized state, Suspend and Resume, Cancellation, K8s resource orchestration, Exit hooks (on notifications with cleanup), Garbage Collection, Scheduling (affinity, node selectors, etc.), Volumes attachment (ephemeral or persistent), Daemoned steps (e.g., starting a service that will run throughout the whole workflow duration), Script steps, Sidecars, CI/CD, Parallelism limits enforcement, and Docker in Docker (DinD).

As Listing 7 shows, Argo Workflows defines a new K8s resource, *Workflow*, and the definition is at the YAML level, aligned with the K8s resource definition style. It is not reasonable to expect that a data scientist will use Argo Workflows. Rather, it can be be used under the hood by the high level frameworks, such as PyWren or Kubeflow. In a forward reference to our proposed architecture discussed in Section 4, Argo is proposed as a candidate workflow orchestrator that will execute and orchestrate data intensive serverless workflows created automatically from the high level programming language code (e.g., Python).

```yaml
# The following workflow executes a diamond workflow
#
#   A
#  / \
# B   C #B and C execute in parallel
#  \ /
#   D
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: dag-diamond-
spec:
  entrypoint: diamond
  templates:
  - name: diamond
    dag:
      tasks:
      - name: A
        template: echo
        arguments:
          parameters: [{name: message, value: A}]
      - name: B
        dependencies: [A]
        template: echo
        arguments:
          parameters: [{name: message, value: B}]
      - name: C
        dependencies: [A]
        template: echo
        arguments:
          parameters: [{name: message, value: C}]
      - name: D
        dependencies: [B, C]
        template: echo
        arguments:
          parameters: [{name: message, value: D}]

  - name: echo
    inputs:
      parameters:
      - name: message
    container:
      image: alpine:3.7
command: [echo, "{{inputs.parameters.message}}"]
```

Listing 7: Simple DAG Argo Template (*Source [61]*)

3.3 Massively Parallel Serverless Computational Frameworks

Until recently, serverless computing was not used for data-intensive computations. In fact, some in the serverless community considered this as an anti-pattern [62]. In 2017, Eric Jonas et al. built a prototype called PyWren [15], and showed how the serverless execution model with stateless functions can substantially lower the barrier for non- expert users to leverage the cloud for massively parallel workloads. By skipping the need for complex cluster management and using a simple Pythonic futures-based interface, PyWren allows users' non- optimized code to run on thousands of cores using cloud FaaS.

Several other attempts recent attempts have been made to implement MapReduce with serverless technology, such as Qubole's Spark on AWS Lambda [63] and Flint [64] (another implementation of Spark on AWS Lambda). Qubole creates an execution DAG on the client side and runs each task as a serverless function. In addition to the users' difficulty to learn Spark API, these approaches present performance issues. For instance, Qubole reports executor startup times to be around two minutes in the cold start case. Also, the unmitigated pure data shipping model of AWS Lambda causes network traffic and overhead related to the data shuffling. Flint reports similar issues.

Other works have tried to implement a Map/Reduce serverless framework from scratch such as Map/Reduce Lambda [65] , which is still under active development. Many open issues remain in these systems, where data shuffling remains one of the bigger challenges in running Map/Reduce jobs over serverless architectures. Several proposals have been made on how to make shuffle more efficient by using AWS ElastiCache [66].

As we go to press, we are not aware about any implementations of a serverless Map/Reduce (or Spark) that make use of K8s and K8s native serverless technologies. The CloudButton project intends to fill in this void.

3.4 Cost Efficiency

It has been widely discussed in the art (both industrial and academic) that serverless economic benefits largely depend on the workload behavior [67, 68, 69, 70]. Namely, depending on the time utilization of a serverless function it can be either cheaper than a VM/server or more costly if time utilization crosses some break-even point depending on the pricing scheme for serverless functions and virtual machines. This is a manifestation of a general problem related to consuming cloud compute resources using different resource flavors and billing contracts as discussed in [71]. In that paper, serverless is not considered *per se*, but serverless functions can be easily incorporated into the same modeling approach. This is indeed part of our future plans in improving cost-efficiency of the CloudButton platform. However, this approach addresses cost-efficiency of serverless only partially.

Other important points include:

- Rightsizing: serverless functions require specification of a *flavor* in terms of CPU and memory (or just memory if CPU is allocated proportionally to main memory as in AWS Lambda). Obviously, this problem is not new. In fact, at its core, it is similar to rightsizing of VMs with an important difference that serverless functions are short lived while VMs stay put for months potentially. In fact, rightsizing of serverless functions is simpler in this respect: a representative statistical population of serverless requests is quickly accumulated allowing to make rightsizing decisions. However, serverless functions can be immutable (e.g., pods in K8s), which adds complexity to rightsizing of serverless functions;

- Over-commit subject to SLAs: overcommit subject to SLA is key to cost-efficiency. It has been studied previously in the context of VM overcommit in a cloud [72, 73]. However, with serverless functions, some assumptions, e.g., assumption on a small size of VM groups dependent through auto-scaling, are not valid in the serverless environment. For example, a single user can issue many thousands of short lived functions in parallel. Hence, overcommit might be more challenging in serverless environments.

- Resource Reservation: resource reservations are generally considered as an antithesis to serverless. However, with the servermix based workflows, reserving resources for highly utilized components while using serverless for high peaked irregular workflows might be a useful approach. Again, extending findings of [71] can be helpful to achieve this goal.

- Serverlessness of the workflow orchestrators: in conventional workflow orchestration, the workflow controllers continue to consume resources even if there are no events requiring their attention. Consider long period of silence between the ETL steps in a multi-hour (or even a multi-day) workflow. Obviously, solutions such as Apache Composer are not suitable to remedy this inefficiency, because of their short-lived ephemeral execution. A new solution is required to achieve serverless orchestrators and it will be pursued in the CloudButton project.

3.5 SLA Monitoring and Management

Service Level Agreements (SLAs) in Cloud computing environments allow cloud users and providers to reach a common understanding on Cloud services' characteristics and allow to express requirements in terms of quality of service (QoS).

As introduced in *D2.1 Experiments and Initial Specifications* the area of QoS and SLA management has been broadly researched initially in the scope of Grid computing. These initial activities were evolved afterwards in context of traditional VM-based Cloud Computing. Within the latter area some exemplary works are: OPTIMIS project [74], focusing on the applicability of SLAs in multi-cloud and hybrid cloud models. In order to do so, diverse execution and deployment configurations were considered for the SLA negotiation and management processes. These included different deployment and runtime configuration scenarios such as private and public cloud, as well as cloud bursting and multi-cloud (optionally including the cloud broker role). The QoS terms considered in OPTIMIS focussed around operational capabilities of providers in terms of risk of failure, trust in the provider to fulfil the agreed SLAs, runtime eco-efficiency of the infrastructure and adherence to legal requirements, in relation to personal data management. Interestingly for CloudButton for cluster federation approaches, OPTIMIS approached the question as a negotiation among a private and public cloud provider to punctually purchase the necessary additional resources. Focussing at the Cloud PaaS level, Cloud4SOA [75] framework offered dynamic SLA negotiation for Multi-Cloud PaaS scenarios. CLOUD4SOA instead of addressing the infrastructure level QoS metrics, took into account business dynamics. These were considered in order to establish the dynamic relation among business performance metrics, Service Level Objectives, and correlated SLA performance metrics.

In modern Cloud Native Application environments based on Kubernetes, there have been significantly lesser amount of research activities so far. Natively, Kubernetes offers two mechanisms in relation to QoS[76]: QoS classes, as well as, requests and limits, which work tightly united in order to offer the inherent QoS support in Kubernetes. Requests represent the capacity in terms of resources demanded by a container in pod - which defines the deployment unit as a collection of containers. Limits define maximum amount of resources a container can consume. At level of scheduler, a pod in Kubernetes — can be assigned with different QoS classes. The considered QoS classes are: Guaranteed, Burstable and BestEffort. Guaranteed ensures that both resources and limits are ensured by the Kubernetes scheduler. Burstable pods get resources defined by resources and can grow-up to the limit in case of resource availability. BestEffort, considers the workload as not critical and requests and limits are not established for the containers execution.

In [77] Heidari presents a survey on the QoS mechanisms available in diverse Container Orchestration technologies including Kubernetes, Mesos and Aurora. In this, available mechanisms in Kubernetes are highlighted while recognising room for improvement, for example with regards to support to tenant quotas and guarantees. In a different direction, [78] defines a performance and management model for Kubernetes that aims at supporting porting applications to Kubernetes environments and with capacity planning according to application needs. Findings of this work, offering a characterisation of workloads and modelling its behaviour in this environment are of interest for CloudButton QoS metrics selection.

4 Overview of the Proposed Architecture

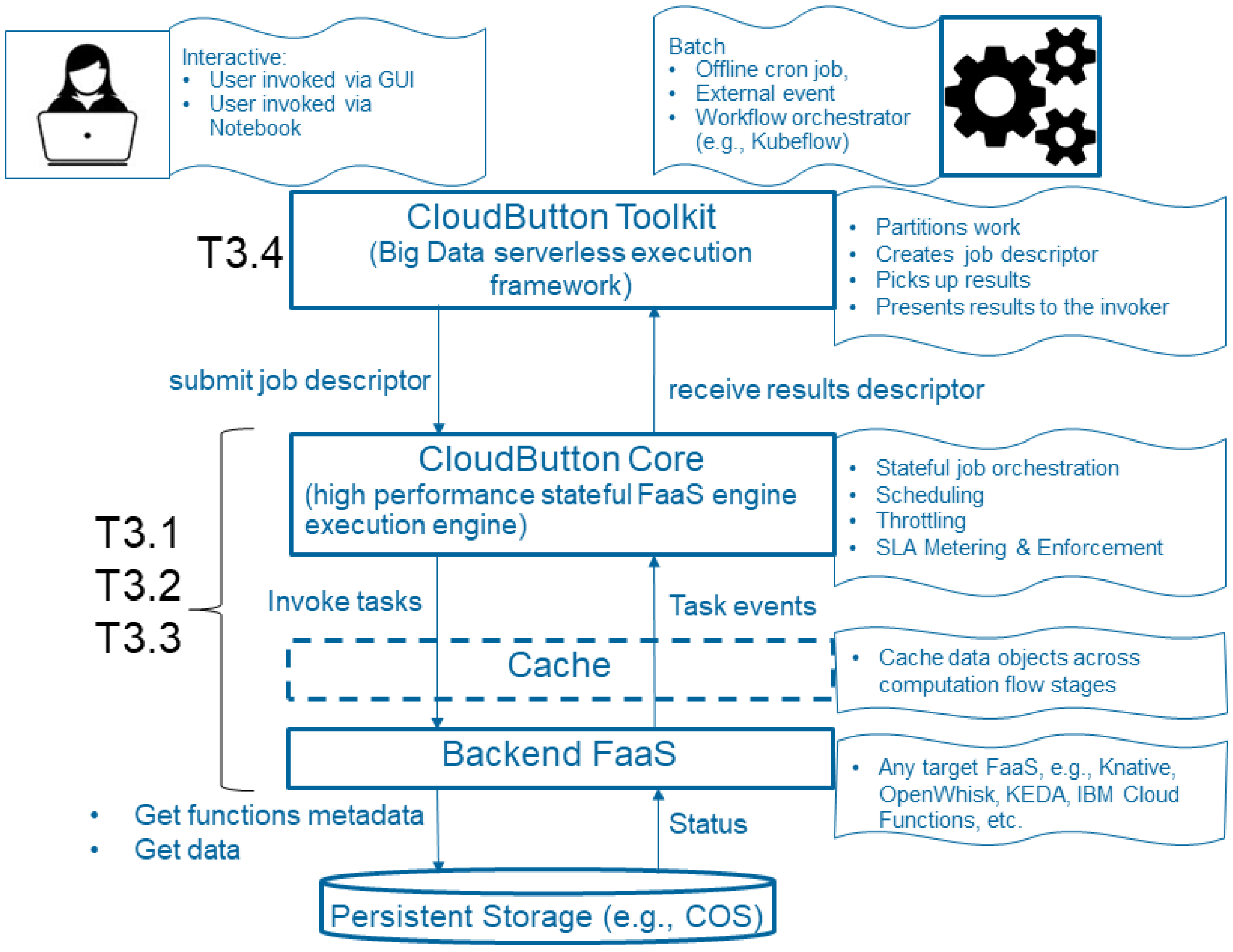
Interactive:
• User invoked via GUI
• User invoked via Notebook
Batch
• Offline cron job,
• External event
• Workflow orchestrator (e.g., Kubeflow)
T3.4
CloudButton Toolkit
(Big Data serverless execution framework)
• Partitions work
• Creates job descriptor
• Picks up results
• Presents results to the 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 15: High Level Architecture

Figure 15 outlines the high-level architecture of the CloudButton framework. The architecture design follows a cloud-native approach. Essentially, it is a mesh of micro-services where each micro-service can be developed, maintained, and deployed independently. We identify three logical groups of these micro-services that we term (a) CloudButton Toolkit, (b) CloudButton Core, and (c) FaaS Backend. These large functional "blocks" interact by means of the well-defined interfaces, which allows for a variety of reference implementations. The constituents of each "functional block" are described in detail in the reminder of this section.

4.1 Design Principles

First, we lay out the guiding principles of the design. The goal is to create a robust yet agile meta-structure of a serverless data intensive computational engine that would ensure a lasting impact of the CloudButton project by keeping its artifacts relevant throughout the project duration and beyond. This is extremely challenging because of the fluidity of the cloud native open source projects and commercial offerings landscape.

To exemplify this, consider, one of the main challenges that CloudButton faces: proliferation of the open source projects of relevance that might be essential to leverage in the future. For example, there are currently more than ten FaaS frameworks at different levels of maturity claiming to be native to the K8s environment. It is reasonable to expect that more such FaaS frameworks will appear throughout the duration of the project, not to mention multiple public FaaS offerings, which also can serve as the execution substrate for the CloudButton Toolkit. Hence, for example, the client execution framework, CloudButton toolkit and a specific FaaS environment should be decoupled by placing and intermediate CloudButton Core as a mediator and adapter between a specific FaaS Backend and a specific client execution framework.

- **Zero Learning Curve:** a user should not be required to master yet another framework to exploit CloudButton. Rather, a data scientist or an application developer will be able to use a high level programming language, e.g., python and simply use library functions to request a specific function execution in a massively parallel way, i.e., to push the "cloud button" to run this function in the pluggable FaaS Backend with Cloud Button Toolkit creating a job descriptor and picking up the results transparently, CloudButton Core orchestrating the execution and persistent store, such as COS storing the executable functions and th eexecution results. In this sense, the CloudButton Toolkit will offer the level of abstraction similar to PyWren, Kubeflow, and Airflow.

- **Native integration** Serverless computing, as is today, requires users to be familiar with REST API of the underlying FaaS platform. Users requires to design their code to be invoked as an action according to well defined specification of the FaaS platform. For example, an action to be invoked in Apache OpenWhisk will always return a JSON response, that may also include result of the code running inside invocation. Input parameters to the invocation, usually provided in JSON also. Using JSON for input and output has limitations, as it not possible to write arbitrary code that accept any class as an input and return any class at the output. To overcome this limitation, CloudButton Toolkit Client will accept any user function that may have any input type and return any class. This will allow users to use serverless natively with their code, without modifying existing code or design their code according to FaaS specifications.

- **Dependencies management** User's code or applications usually dependents on various additional packages or libraries. Deploying user's code as serverless action requires not only to deploy the code but also to deploy all dependencies to make cloud runtime identical to the one being used by the user. The process which required to deploy and identify dependencies is not straightforward and usually force users to spent many efforts. CloudButton toolkit client will be designed to automatically scan dependencies packages and versions and then automatically deploy not only the user written code, but also required dependencies. We also allow users to include large size dependencies inside Docker images.

- **Support for Complexity:** big data analytic applications can be complex comprising multiple interrelated steps with intermediate state in between. CloudButton design should support stateful flows efficiently to cater for the application complexity.

- **Cost-Efficiency:** The cost/performance tradeoff should be controlled by the users in a simple manner, so that they will be able to provide their preferences to the CloudButton Toolkit and monitor compliance with the specified cost/performance goals

- **Extreme Scale:** Cloud Button should scale to the very large data volumes without incurring excessive load or interference between different workloads from the different tenants.
- **Agility:** The CloudButton framework should be amenable to deployment in various setting. For example, we should be able to deploy it in a single K8s cluster, in a multi-K8s cluster setting, in a public cloud, in a hybrid cloud, and a light weight version of it should be deployable on a data scientist's laptop.
- **Extensibility:** It should be possible add new functionality, such as new scheduling logic, new work partitioning logic, etc. without the need to stop the system or re-engineer it.
- **Portability:** The CloudButton framework should be portable across public cloud vendors and private cloud platforms.
- **Multitenancy:** The CloudButton framework should support multi-tenancy by providing effective isolation between the users' workloads. In particular, the CloudButton framework should allow to explicitly specify, monitor, and enforce performance SLOs.

4.2 CloudButton Toolkit

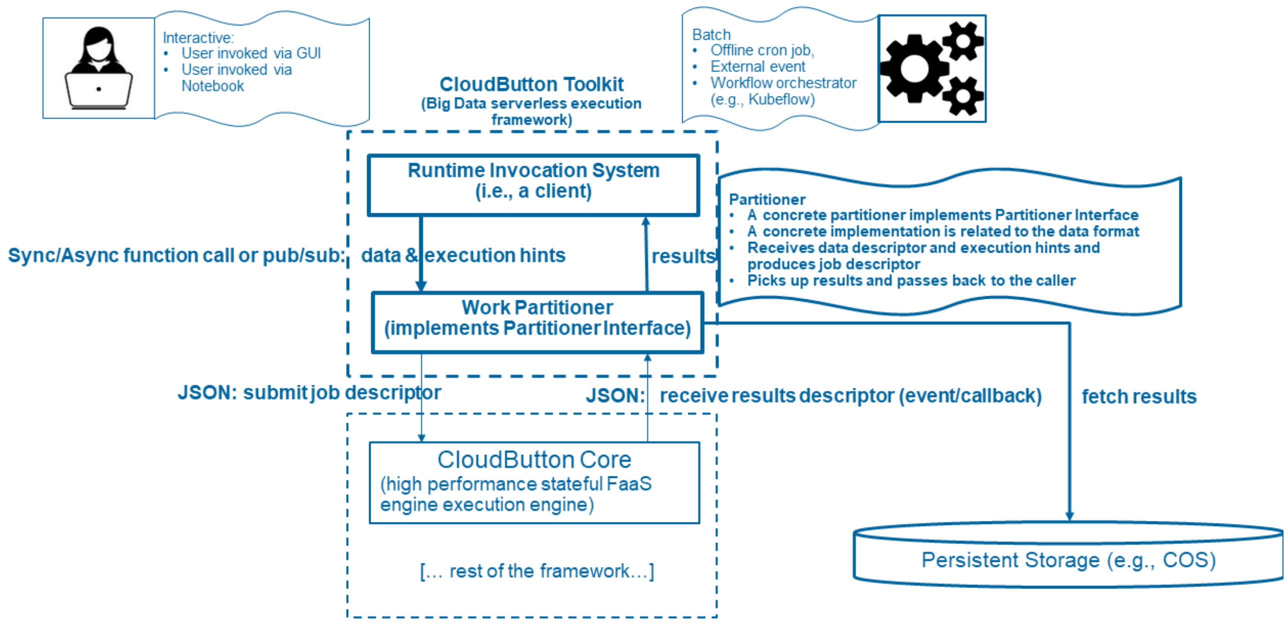
Interactive:
User invoked via GUI
User invoked via Notebook
CloudButton Toolkit
(Big Data serverless execution framework)
Batch
Offline cron job,
External event
Workflow orchestrator (e.g., Kubeflow)
Runtime Invocation System (i.e., a client)
Partitioner
A concrete partitioner implements Partitioner Interface
A concrete implementation is related to the data format
Receives data descriptor and execution hints and produces job descriptor
Picks up results and passes back to the caller
Sync/Async function call or pub/sub:
data & execution hints
results
Work Partitioner (implements Partitioner Interface)
JSON: submit job descriptor
JSON: receive results descriptor (event/callback)
fetch results
CloudButton Core
(high performance stateful FaaS engine execution engine)
[… rest of the framework…]
Persistent Storage (e.g., COS)

Figure 16: High Level Architecture: CloudButton Toolkit

Figure 16 highlights the structure of the CloudButton Toolkit. The toolkit comprises two principal components:

- **Runtime Invocation System:** the Runtime Invocation system is a user-facing client comprising library functions to interface with the Work Partitioner. The library functions for the languages popular with the data scientists can be provided. Given the popularity of Python, CloudButton will focus on the Python language first. It is presumed that we should be able to deploy the Runtime Invocation System should in a variety of scenarios as stipulated by the Agility requirement of the previous section. In one typical use case the Runtime Invocation System can be collocated with a Jupiter notebook, a local program or some type of GUI. Other examples include enterprise automation scenarios. In these scenarios, the Runtime Execution System can be invoked by a cron job, a handler of some external event, a cloud-deployed workflow orchestrator, etc.

- **Work Partitioner:** Parallel data processing is a core functionality of any Big Data engine. For example, Apache Spark examines input data and apply algorithms to logically partition the data. Having partitions, Spark generates DAG of tasks, where each task assigned to process a single partition. While serverless platforms is a good candidate to run stateless parallel computations, it still remains complicated to process large volumes of data. If user need to process large amounts of data, it's his responsibility to partition datasets, making the entire process too complicated. For instance, if user need to process files from the object storage he will need to write specific code to partition the input data and then manually invoke actions for every partition. The complexity of this process is clearly a major blocker for the wider adoption of FaaS for Big Data processing.

  The Work Partitioner is the critical component of the proposed architecture. The Work Partitioner was bundled with the client (i.e., Runtime Invocation System). This puts limits to extensibility and agility. Hence, a design decision was made to separate between the client and the Work Partitioner. The Work Partitioner can be deployed either locally on a user's computer or in a cloud.

  At it's core, the Work Partitioner obtains a description of the data objects to be processed from a caller (or from a message queue it subscribes to) along with the desired SLOs for the processing (e.g., a deadline). The data objects description includes the data format, which would dictate specific partitioning function to be dynamically loaded. For example, partitioning of the CSV the data for the parallel processing would be different from partitioning an MPEG data (e.g., for a per-frame processing). Work Partitioner outputs a ***job descriptor***. The job descriptor is a JSON object that contains both mandatory and optional key/value pairs describing the computation required by the client that helps the CloudButton Core to create actual computation tasks to carry it out and orchestrate their execution.

  It is envisioned that a library of different implementations for the Work Partitioner will be implemented and additional implementations can be further added by the third parties desiring to join the CloudButton ecosystem. The Work Partitioner is represented by a generic facade function that is used by the Runtime Execution System. Two basic implementation options of the actual Work Partitioner will be supported. In the first option, the Work Partitioner is implemented as a local function. In the second option, the Work Partitioner function proxies the actual remote implementation that can be deployed remotely, e.g., in the cloud. In this case, Work Partitioner is a microservice that can be accessed either via REST or messaging API (this implies different proxies that will be dynamically loaded as a facade for the remote Work Partitioner).

4.3 CloudButton Core

Figure 17 depicts the structure of the CloudButton Core.

The underpinning design principle of the CloudButton Core is that it follows a cloud-native microservices-oriented architecture. The integration point between the CloudButton Toolkit and the CloudButton Core is the job descriptor JSON.

The CloudButton Core comprises a number of microservices and their basic inter-operation as follows[6].

- **A Flow State Service:** The purpose of this microservice is to provide an intermediate state and event notifications pertaining to a specific execution flow. In the CloudButton approach, every job execution is a flow. Even if the client framework does not explicitly work with flows, the flows are implicitly created by the Flow Service of the CloudButton Core, which uses the Flow State Service to orchestrate a flow execution.

[6]The dashed rectangles around the microservices groups signify that in some implementations, these microservices functionality might be bundled together in a single microservice.

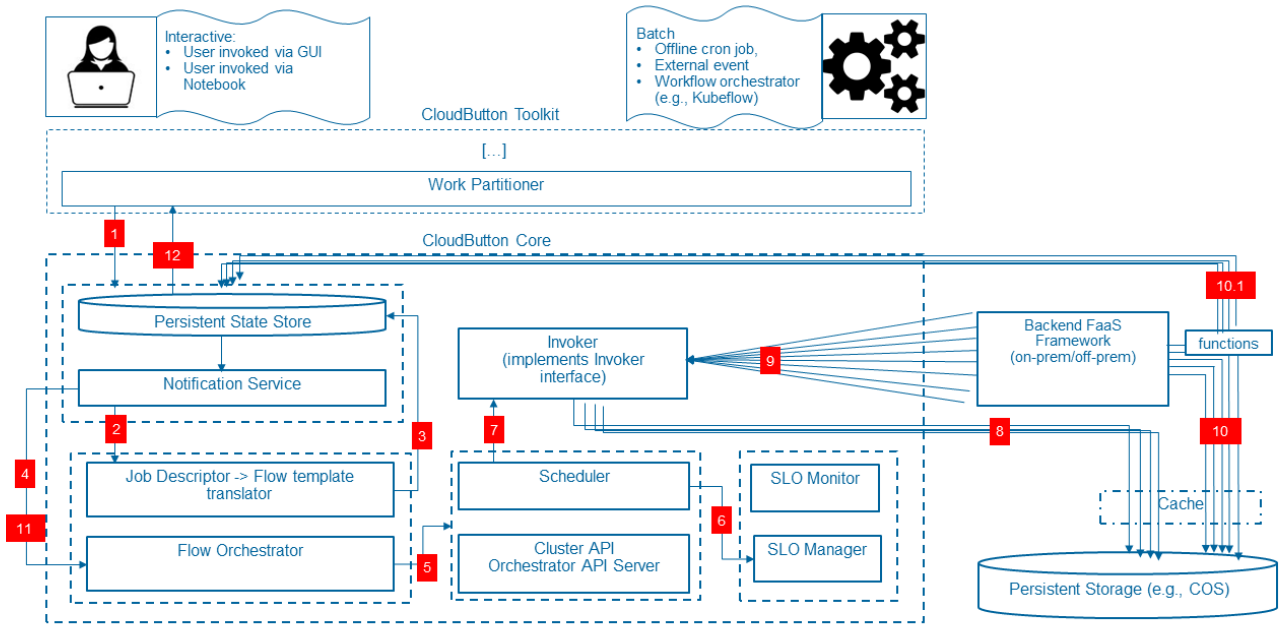
Interactive:
User invoked via GUI
User invoked via Notebook
Batch
Offline cron job,
External event
Workflow orchestrator (e.g., Kubeflow)
CloudButton Toolkit
[...]
Work Partitioner
CloudButton Core
1
12
Persistent State Store
Notification Service
2
3
4
11
Job Descriptor -> Flow template translator
Flow Orchestrator
5
Invoker (implements Invoker interface)
7
9
Scheduler
Cluster API Orchestrator API Server
6
SLO Monitor
SLO Manager
8
Backend FaaS Framework (on-prem/off-prem)
functions
10.1
10
Cache
Persistent Storage (e.g., COS)

Figure 17: High Level Architecture: CloudButton Core

- **Persistent Data Store:** this component can be implemented and deployed in a variety of ways. For example, this component can be implemented using a pub/sub system with retention policies (e.g., Kafka or Rabbit MQ). In another implementations, this component either can be a document database, such as Cloudant or CouchDB or a cache service, such as Redis or InfiniSpan. Requirements for this microservice include persistency for the document objects (JSON) describing the data pertaining to the flow execution.
- **Notification Service:** this service produces all the events of interest to the Flow Service and the CloudButton Toolkit. For example, this service notifies the Flow Service upon a submission of a new job descriptor by the CloudButton Toolkit (Step 1 in Figure 17). Likewise, it notifies the CloudButton Toolkit about completion of the job (Step 10 in Figure 17).

- **A Flow Service:**
  - **Job Descriptor to Flow Translator:** this microservice produces a flow descriptor (e.g., Argo flow template or and Airflow DAG – depending on the configuration of the Flow Orchestrator) upon a notification from the Flow Data State service about a new job descriptor submission by the CloudCore Toolkit (Step 2 in Figure 17). As shown in Step 3 in Figure 17, the flow descriptor is handed back to the Flow State Service. This causes a notification to the Flow Orchestrator in Step 4.
  - **Flow Orchestrator:** this microservice orchestrates the execution of a computational job. It takes care of counting tasks successful completions, restarting failed tasks of the job and satisfying precedence order of the tasks Flow Orchestrator can be implemented based on a mature open source project such as CNCF Argo or Apache Airflow. In Step 5, the flow template, e.g., Argo flow template, is handed off to the K8s API Server that will forward it to the custom FaaS Scheduler.

- **FaaS Scheduling Service:** This service is responsible for the high level scheduling decisions w.r.t. serverless functions executing as part of the overall computation (i.e., a flow). It should be noted that because of the functional separation decision that we made earlier in the interest of extensibility and agility, FaaS Scheduling Service does not exercise a direct control over the backend FaaS Framework in all cases. For example, in case the backend FaaS framework is provided by a public cloud, it is being treated as a black box. In this case, the FaaS Scheduling Service reduces to traffic shaping and capacity pooling across multiple regions or multiple clouds. For instance, the FaaS Scheduling Service is aware of the per-account per-region system limits on the number of function executions per time unit and simultaneous executions allowed and dispatches requests so as to minimize errors due to violating of these limits.

However, in another scenario, when the backend FaaS System runs on K8s and is implemented via Knative, KEDA, or OpenWhisk, FaaS Scheduling Service assumes direct control of scheduling serverless functions on K8s via K8s custom scheduler extension implementation [79]. In the latter case, the FaaS Scheduling Service can enforce SLOs related to a contracted performance of a specific flow.

  – **Cluster API:** this microservice is either K8s API Server (when deployment of the CloudButton Core) is on K8s or a facade API server (that can itself be implemented as a serverless Web service via, e.g., Knative) that proxies the High Level FaaS Scheduler.
  – **High Level FaaS Scheduler:** this microservice is either a custom K8s scheduler that operates on the pods labeled for its attention by the Flow Orchestrator Service or a scheduling service that operates on the external backend FaaS Framework using Custom Resource Definitions (CRD) for this framework. In Step 6, the FaaS Scheduler consults SLO Manager to obtain scheduling hints that would help enforcing the target SLOs. Finally, in Step 7, the FaaS Scheduler invokes the serverless functions through the Invoker.

- **SLA/SLO Compliance Service:** This service is responsible for monitoring SLO compliance and helping the FaaS Scheduling Service to make scheduling decisions to ensure compliance with the SLO clauses of the SLA.
  – **SLO Monitor:** monitors per flow metrics related to performance of the computation as specified in the SLO definition passed in the job descriptor.
  – **SLO Manager:** this microservice checks compliance of the measured performance metrics to the ones that were put as a goal for the execution and makes this information available to the FaaS Scheduling Service.
- **Invoker:** Invoker serves as a facade to the backend FaaS Framework. On the southbound interface, Invoker is specific to a FaaS beackend framework. On the northbound interface, Invoker obtains a command for execution by the scheduler that should be run in a massively parallel manner against the backend FaaS framework. In fact, Invoker can be regarded as a Scheduler's plugin. In an actual implementation, Invoker can be a serverless function in its own right. Obviously, integrating a new backend FaaS framework, requires developing a plugin for this service. In Step 8 Invoker obtains the function to run and in Step 9 invokes it in parallel against the backend FaaS Framework and the input data in Persistent Storage.
- **Backend FaaS Framework:** as we already discussed above, multiple backend FaaS frameworks will be considered with each system being accessed via its specific Invoker plugin. A common denominator is that the backend FaaS framework appears as a multi-tenant service to the outside world. Accessing it via an Invoker instance on behalf of a specific tenant would require the Invoker to have an access to the API key and possibly other secrets pertaining to the tenant executing this specific flow. The per-tenant secrets will propagate to the CloudButton Core from the CloudButton Toolkit. In Step 10, the serverless functions write the results into Persistent Storage (possibly caching them for the next round of computation) and in the same Step (we denote it as Step 10.1 they put results descriptors on the Persistent Store of the Flow State Service, which causes notifications to the Flow Service in Step 11. The Flow Service manages these and other events related to the flow and it the Flow Service that publishes the result of the computation on the Flow State Service (not shown in the figure), which – in turn – alerts the CloudButton Toolkit in Step 12.

- **Persistent Storage:** this service is required to store the serverless function serialized metadata and the computation results.
- **Cache:** caching is required to improve performance of the massively parallel serverless computations. One of the more important problems with the serverless architecture is its data shipping architecture. Caching of the intermediate results or often accessed objects (e.g., for model serving or repeating part of the computation avoiding recalculations), dramatically improves performance.

4.4 Hybrid Cloud Deployment

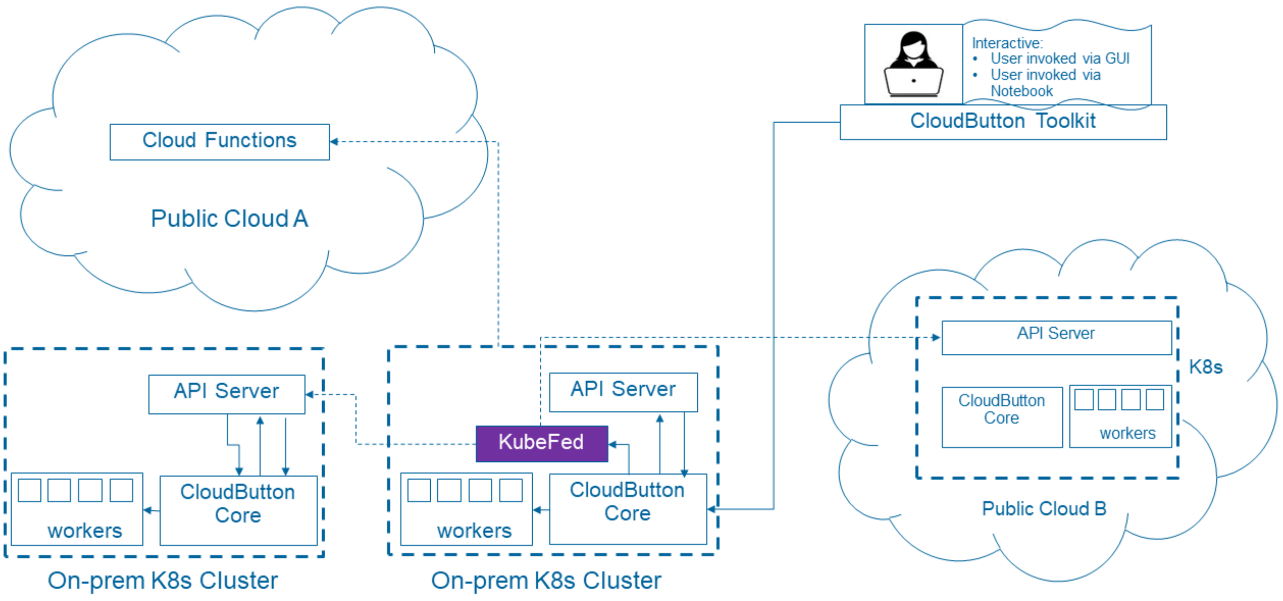

Figure 18: High Level Architecture: Hybrid Cloud Deployment

Figure 18 highlights deployment of CloudButton in a hybrid cloud setting. In this example, the hybrid cloud comprises two on-prem K8s clusters and one off-prem K8s cluster deployed in a public cloud. In addition, there is another public cloud provider that offers Cloud Functions (i.e., FaaS). The CloudButton Core is deployed in both on-prem and off-prem K8s clusters. It is not deployed at Public Cloud A. The Cloud Functions of that provider is treated as a black box and serves as a backend FaaS Framework to the CloudButton Core deployed in the on-prem K8s where KubeFed [80] is located.

A different deployment configuration could be considered. For example, the CloudButton Core could have been deployed in only one of the K8s clusters with other clusters providing their FaaS frameworks as FaaS backend only.

However, the deployment option is a preferred one. It provides for autonomy of the clusters and clean separation of concerns with only the local FaaS Scheduler managing local resources for CloudButton.

We now take a closer look at Figure 18. The main idea behind the proposed deployment architecture is enabling capacity pooling across K8s clusters and external FaaS resources to overcome the elasticity limits, which otherwise will be too restrictive thus preventing massive parallelism. One of the K8s clusters is designated as a federation master. This cluster runs KubeFed that allows to define federated K8s resources, which can be both K8s native resources and arbitrary CRDs.

Figure 18 shows K8s API Server outside of the CloudButton Core for clarity. The FaaS Scheduler (an custom scheduler extension for CloudButton) is assumed to be inside the CloudButton Core. When a request for a new job execution is produced by the CloudButton Toolkit, the CloudButton Core submits applies the new flow resource (e.g., Argo flow YAML definition) to K8s API server. The latter forwards scheduling requests to the FaaS scheduler (i.e., the component that is logically inside

CloudButton Core). The FaaS Scheduler then uses local capacity to schedule serverless functions. In case local capacity is insufficient, CloudButton Core turns to KubeFed for to access federated capacity via CloudButton Core in other K8s clusters or even outside K8s federation. Each such request is forwarded by KubeFed to the local API Server and there the process repeats itself. Each FaaS Scheduler is only responsible for the local capacity scheduling and turns to other clusters only when needed. The master CloudButton Core (i.e., collocated with KubeFed) can limit the followers in the federation not to propagate requests further (i.e., to enforce a star topology useful for Map/Reduce type of computations).

To be able to execute as explained above, the master CloudButton Core should be able to partition the flow DAG into sub-DAGs and the federation should be pre-configured. The same flow descriptor, job descriptor, and result descriptor JSON will be used across the federation.

4.5 Software Architecture

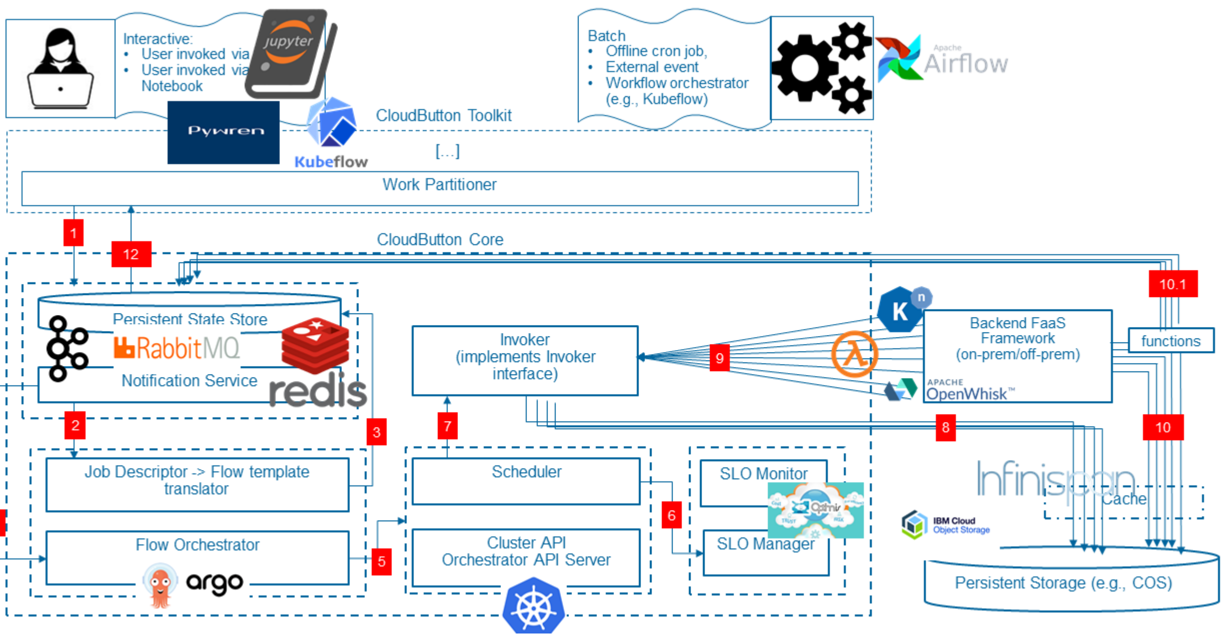

Figure 19: Cloud Button Toolkit and Core Architecture

Figure 19 shows our initial software architecture. The components without a logo superimposed on them will be developed from scratch (or almost from scratch), because there is no open source code that can be readily reused to implement them[7]. For other components, the multiple logos superimposed on them, signify possible choices. For the SLA/SLO management component, we evaluate whether software from Optimis [74] SLA/SLO management component can be reused by CloudButton.

5 Initial Prototype Design and Implementation

5.1 EMBL Use Case

In this subsection we recap the motivational use case of EMBL. The Metabolomics use case comprises practical use-cases:

1. Interactive exploration of the raw data in both spectral and imaging
2. Metabolite-images-guided analytics on the raw data by executing simple machine learning tasks such as finding similar images in the raw

[7]In case of the Job Descriptor to Flow Template Translator component, some code from Kubeflow that generates Argo templates can be reused.

3. Search for novel biologically and medically relevant small molecules in the METASPACE [81] knowledge base
4. Providing a flexible and at the same time efficient public access to the raw data to enable data-intensive third-party applications including novel engines for metabolite annotation.

A typical use case involving METASPACE includes a user interaction with the UI of the Metabolomics application that drives the underlying metabolomics engine.

The user experience in the typical use comprises the following steps:

- A researcher uploads medical images to the object storage;
- The researcher chooses a molecular database from the existing public data sets. The researcher may also choose to use her own database;
- The researcher executes metabolomics engine that will find potentially interesting molecules and indicate where they were detected in the uploaded sample. In certain cases, the researcher knows what specific molecules to look for. In other cases, the search is for the unknown molecules.

The Metabolomics engine comprises a complex Python application that performs binary data pre-processing, data extractions, pattern matching, and images generation. The Metabolomics engine uses Apache Spark [82] that is deployed on a cluster comprising several cloud based virtual machines.

A typical flow can be summarized as follows:

- Segmentation and data pre-processing of the user provided images and specific molecular databases chosen by the user;
- Annotation (including re-segmentation of the centroids file);
- FDR calculation;
- Downloading of the results data frame;
- Downloading images of all target ions.

To start with, we identified a particular representative Metabolomics use case involving METASPACE. We performed thorough comparisons between the existing implementation based on Apache Spark vs an implementation based on the CloudButton Toolkit.

Our first step was to to leverage CloudButton Toolkit and modify the representative use case by replacing calls to Apache Spark with calls to a FaaS engine. To achieve this, we imported CloudButton Toolkit as a package and only few additional lines of code were required to replace the calls to Apache Spark with the calls to the CloudButton toolkit. Using CloudButton toolkit we demonstrated that the same metabolomics engine can now get elastic serverless resources and run serverless invocations on demand and just in time for the actual workload, instead of the current approach where engine was executed over static Spark cluster deployed on VMs procured irrespectively of the actual load. With CloudButton Toolkit we managed to scale Python functions at a massive scale (hundreds of simultaneously executing computations) against the metabolomics datasets stored in the IBM Cloud Object Storage.

Running our initial tests, we got impressive parallel data processing and speed of deployment. On average it took less then 25 s to ramp up to our current target capacity of 256 parallel invocations. The backend FaaS engine keeps the functions warm for about 10 minutes after the computation terminates. Thus a new computation (if triggered within this period) has even lower start-up latency.

This initial implementation was open sourced and made public at [83]. In the next phases of the project this initial prototype will be iteratively improved. In particular, we will modularize the toolkit and integrate it with the CloudButton Core

5.2 CloudButton Toolkit - Initial Prototype

In this section, we summarize the initial CloudButton Toolkit prototype

The initial prototype of CloudButton Toolkit is developed in Python and uses 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 data (including intermediate results (e.g., output of a `map` task) on the server side). 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. The initial CloudButton toolkit includes a client, which can run on a laptop or in the cloud. The client is responsible for taking user's Python code and the relevant data, and saving them into IBM COS, among other things.

The default Python runtime used in the CloudButton Toolkit is python-jessie:3. It includes the most common Python packages, so it supports most computations out of the box. IBM Cloud Functions runtime for the CloudButton Toolkit is based on Docker images. In other words, while the researcher only writes Python code, this Python code is injected into a pre-baked Docker image containing python-jessie:3. An important feature supported by CloudButton is that a user can build a custom Docker image with any non-standard required packages, either Python libraries or system libraries and binaries, and then upload the image to the Docker hub registry. IBM Cloud Functions service will get the image from the registry and will use it to run the serverless functions. IBM Cloud functions gets the self-built Docker container image from the Docker registry the first time a user invokes a function with the specific runtime. Then, the Docker container is cached in an internal registry and no additional overhead of pulling the container image from the remote registry in Docker hub is being paid..

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.

The standard way to configure CloudButton to work with the Jupiter notebook is to import the CloudButton Toolkit module that will also contain an executor for IBM Cloud Functions.

Then an executor instance should be created. Upon creation, a unique executor ID is generated in order to track function invocations related to this executor and the results of the execution (carried out by the individual functions) will be stored in IBM COS.

Obviously, access to both IBM services (i.e., IBM COS and IBM Cloud Functions) is required for the executor to operate. The executor configuration includes the secrets pertaining to the user account that will be billed for using IBM COS and IBM CloudFunctions.

We explain the principles of CloudButton toolkit by a very simple example. Assume a developer who uses local Jupyter notebook and codes a function which receives an integer type input and returns a received value plus 7

```
def sum_function(x):
    return x + 7
```

Now assume user wishes to run this function as a serverless action in the cloud with one function instance per input data item (horizontal scaling). To make it more interesting, we assume that the user also wants to summarize the results of all invocations to obtain a grand total. Thus, in this example we have two steps: map and reduce. Consider an array of 4 entries that will serve us as input. The listing below shows sequential Python code that will be scaled out by CloudButton.

```
data = [1, 2, 3, 4]

def sum_function(x):
    return x + 7

def combiner(results):
    total = 0
    for map_result in results:
        total = total + map_result
    return total
```

Using CloudButton Toolkit requires to add only 3 lines of code to scale the user function as a serverless actions deployed in the cloud. CloudButton toolkit will generate an action against each entry from the array and automatically invoke summary function, once all invocations have completed. The CloudButton Toolkit based code would is shown in the next listing:

```
import cloudbutton as cbclient

data = [1, 2, 3, 4]

def sum_function(x):
    return x + 7

def combiner(results):
    total = 0
    for map_result in results:
        total = total + map_result
    return total

client = cbclient.ibm_cf_executor()
client.map_reduce(sum_function, data, combiner)
result = client.get_result()
```

Listing 8: Sample CloudButton Toolkit Example

While this is a very simple example, it already demonstrates the power of the toolkit. Imagine that the input might be an array with millions of entries. Alternatively, the input could be a location of the data in object storage. CloudButton will handle all the complexity of execution at scale and summarizing the result totally insulating the data scientists from the technical details involved. Thus indeed created a prototype of a "push to cloud" button, which was our initial motivation.

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.

5.3 Improving Performance via a Message Broker

In the initial prototype implementation, we use only a rudimentary CloudButton core based on the RabbitMQ Message Broker [84]. CloudButton Tookit generates unique job id for all invocations per user submit execution. This job id is a random number, uniquely identifying all invocations belong to particular execution. Consider our example in

When user submits an execution

```
client.map_reduce(sum_function, data, combiner)
```

CloudButton Toolkit automatically generates a unique job identifier number that will identify all 4 invocations that belongs to the particular execution. CloudButton Toolkit will automatically generate status of each invocation and persist this status in RabbitMQ, as a JSON object having job id as a name prefix. Once all invocation completed, the reduce step will be run and the user will get the result.

```
result = client.get_result()
```

Persisting statuses in RabbitMQ instead of object storage results in considerable speed up due to in-memory access compared the original PyWren implementation that persisted the statuses of tasks directly on Object Storage.

5.3.1 A Prototypical Integration between PyWren and Airflow

Apache Airflow is an open source platform that provides authoring, scheduling and monitoring of workflows represented as directed acyclic graphs (DAGs). Every node of the DAG represents a task, and the edges represent the order of execution and dependencies between tasks. The Airflow scheduler executes the tasks on an array of workers, while following the specified order and fulfilling the dependencies [10]. In addition, Airflow allows easily integration with other systems like SQL databases or S3, among others, allowing to apply map reduce tasks to data obtained from a variety of sources. Airflow is a very mature project with a very significant traction.

Airflow executes every task as a thread. Airflow is designed to be scalable, scaling up the number of threads for parallel task execution. Serverless can be helpful for running Airflow in a slow machine while being able to execute map and map reduce tasks or a single function in a matter of seconds.

Combining Airflow's DAGs definition that follows the declarative programming paradigm with PyWren's capabilities of executing massively parallel tasks facilitates creating "declarative DAGs" with tasks that take advantage of PyWren's parallelism in the cloud while taking away the computational load from the host machine that executes Airflow.

A use case example The following example consists on a possible use case of PyWren's map-reduce operator in an Airflow workflow.[8]

The workflow is about an imaginary air pollution central station that controls multiple air sensors all over a city and determines if any of the city districts inhabitants are in danger due to the air's quality. The sensors take an air sample every minute, analyze its components and upload the results to IBM Cloud Object Storage to the corresponding bucket, where there is a bucket for every district.

To begin with, Airflow launches as many map-reduce operations as districts in the city. The map phase applies a function to every sample that analyzes the quantity of harmful components of the air

[8]The source code for the DAG definition can be found at https://git.io/fjzOE

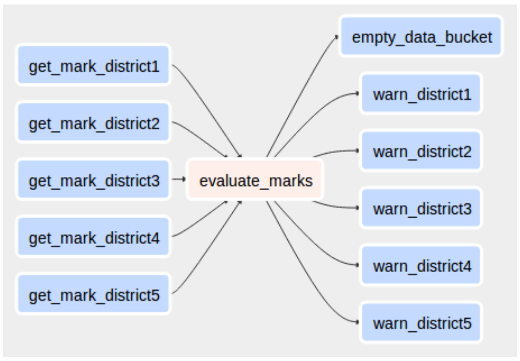

Figure 20: Graph visualization of the example.

and then assigns a mark from 0 to 100 to it, depending on the air quality of the sample. The reduce function calculates the arithmetic mean of the marks provided for every district.

The results go to a branching task. The branching task checks what districts surpass the mark limit from which a district's air is considered harmful.

Finally, a map task is launched for every district marked as harmful that sends a warning notification to every citizen phone that lives in that district. The mobile phone numbers are stored in a separate bucket, one for every district. The tasks that correspond to a district that wasn't considered harmful are skipped by the branching operator. It is also executed a single task that empties the buckets after the samples have been analyzed.

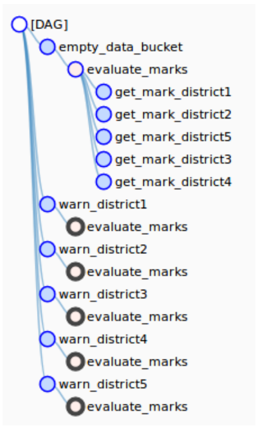

Figure 21: Tree visualization of the example.

The 'get mark' tasks are executed by the map-reduce operator, the 'warn district' tasks are executed by the map operator and the 'empty bucket' task is executed by the basic operator.

6 Next Steps

Our current initial prototype focuses on the EMBL use case and is oriented towards the public cloud FaaS backend and COS. However, a number of the building blocks essential to building a full platform are being developed in parallel by the partners. These blocks are currently at the different levels of maturity and are not integrated. We first briefly outline the status of these components and then provide an outlook into their integration.

6.1 PyWren on Knative Initial Prototype

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service
```

```
metadata:
  name: pywren-action
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: docker.io/sadek/kpywren
          containerConcurrency: 1
```

Listing 9: Prototypical Knative Serving resource definition for PyWren

In this subsection we briefly overview our initial prototype of PyWren/Knative integration. It should be stressed that the scope of this prototype is a simple feasibility check. No optimization or integration with the rest of the proposed architecture has been performed yet. Following up on these issues (as well as strengthening the initial prototype) will be our immediate next step.

Listing 9 shows Knative Serving definition for PyWren on Knative. In Listing 10 we show a simple example of a PyWren program that uses this definition as its FaaS backend. Listing 11 depicts the PyWren invoker (*app.py*) running inside this service definition (see Listing 12). It should be noted that in this prototype we use one request per container. This causes relatively large cold times. Going further, we will optimize the concurrency and cold starts aspects. In this example, *my_function* will be called 4 times (once per each item in the input array – using default work partitioning) and the results will be placed in IBM COS as the backend persistent storage.

```python
"""
Simple PyWren example using one map function invocation
"""
import pywren_ibm_cloud as pywren


iterdata = [1, 2, 3, 4]

def my_function(x):
    return x + 7

config = {'pywren': {'storage_bucket': 'pywren-sadekj', 'storage_prefix': 'pywren.
    jobs'},
          'ibm_cf': {'endpoint': <ISTIO_URL>,
                     'namespace': '',
                     'api_key': '',
                    },
          'storage_backend': 'ibm_cos',
          'ibm_cos': {'endpoint': 'http://s3-api.us-geo.objectstorage.softlayer.net',
                      'access_key': <>,
                      'secret_key': <>}}

if __name__ == '__main__':
    pw = pywren.ibm_cf_executor(config=config)
    pw.map(my_function, iterdata)
    print (pw.get_result())
```

Listing 10: Simple PyWren-on-Knative Example

```python
import logging
import json
from pywren_ibm_cloud import wrenlogging
from pywren_ibm_cloud.action.wrenhandler import ibm_cloud_function_handler

logger = logging.getLogger('__main__')
wrenlogging.ow_config(logging.DEBUG)

```

```python
import os

from flask import Flask, request

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def pywren_task():
    json_dict = json.loads(request.data)
    ibm_cloud_function_handler(json_dict)
    return 'Data: {0}!\n'.format(request.data)

if __name__ == "__main__":
    app.run(debug=True,host='0.0.0.0',port=int(os.environ.get('PORT', 8080)))
```

Listing 11: PyWren Invoker Running inside Knative Serving

```dockerfile
FROM python

RUN apt-get update && apt-get install -y \
        gcc \
        libc-dev \
        libxslt-dev \
        libxml2-dev \
        libffi-dev \
        libssl-dev \
        zip \
        unzip \
        vim \
        net-tools \
        iputils-ping \
        && rm -rf /var/lib/apt/lists/*

RUN apt-cache search linux-headers-generic

COPY requirements.txt requirements.txt

RUN pip install --upgrade pip setuptools six && pip install --no-cache-dir -r
    requirements.txt

# Copy local code to the container image.
ENV APP_HOME /app
WORKDIR $APP_HOME
COPY app.py .
COPY pywren_ibm_cloud ./pywren_ibm_cloud

# Service must listen to $PORT environment variable.
ENV PORT 8080

# Run the web service on container startup.
CMD ["python", "app.py"]
```

Listing 12: Dockerfile for PyWren on Knative

6.2 Argo Based Orchestration Prototype

Argo Pipelines is the generic orchestrator engine that we are planning to use for the CloudButton Core. As was explained in Section 3.2.5, Argo uses YAML static templates for orchestrating workflows. We plan to reuse Kubeflow Pipelines SDK [85] to generate Argo templates from the Python code automatically. In particular, *kfp.compiler.Compiler.compile* can be extended as needed and reused for generating Argo templates for PyWren, Jupiter Notebook multiple cells calculations (where the cells are inter-related with output of one cell being an input to another one), etc.

An important part related to cost-efficiency is serverlessness of Argo workflow execution. To achieve a higher statistical multiplexing gain, the workflow orchestration will be made serverless in the CloudButton Core. This will be achieved at a later stage in the project by combining Argo Pipelines and Argo Events [59] and executing Argo as a Knative serving to scale it to zero when no activity in the workflow occurs. This will be achieved at a later stage in the project.

6.3 SLA Manager

The SLA management component allows to define QoS execution parameters in service level objectives (SLO), to monitor its fulfilment at execution time, as well as, to generate notifications in case of not satisfying the expected SLO. Overall, the SLA Management component offers a solution that handles the complete lifecycle of an SLA considering the following steps: template based SLA definition, continuous monitoring and runtime control including metrics assessment, actions to enforce agreements, accounting and notification of breaches.

The SLA agreement is a JSON structure which details the desired service execution characteristics. An example is provided below.

```
{
    "id": "2018-000234",
    "name": "an-agreement-name",
    "details":{
        "id": "2018-000234",
        "type": "agreement",
        "name": "an-agreement-name",
        "provider": { "id": "a-provider", "name": "A provider" },
        "client": { "id": "a-client", "name": "A client" },
        "creation": "2018-01-16T17:09:45Z",
        "expiration": "2019-01-17T17:09:45Z",
        "guarantees": [
            {
                "name": "TestGuarantee",
                "constraint": "[execution_time] < 100"
            }
        ]
    }
}
```

The architecture details the three main components of the SLA Management framework implementing the features mentioned above:

- Dashboard, Graphical interface to help to define the SLO terms that are part of an agreement and supports to observe the run time behavior of defined agreements.
- REST API, enabling the same functionalities than the Dashboard in a non graphical interface as a REST API.
- Evaluation modules: These perform the runtime logic to perform the monitoring of SLA breaches.
  - Adaptor,it provides a common interface to access diverse supported monitoring systems. Prometheus is the native monitoring system to be used in a Kubernetes execution environment such as CloudButton. The existing wide baseline of metrics in Prometheus will be complemented with specific exporters if necessary i.e. in order to collect information for the status of GPUs supporting NVIDIA GPU or new Xeon Phi exporters.
  - Evaluator,This component entails the steps to assess whether an defined agreement is fulfilled. More concretely, it implements the process of interfacing with the monitoring system by means of the adaptor component and validate if these are in accordance with the requested values in constraints. This assessment process can be executed with a certain periodicity or can be configured for punctual assessment. The simpler form of assessment is in the form of
    "value comparison threshold
    ". These simple rules can be combined using logical operators. In addition, it offers support for indicating the number of repetitions necessary to raise a breach.

– Notifier. In case of SLA breach, the SLA management framework currently supports two different mechanisms for notification of the violation: to invoke an REST interface of another component in the architecture and to push a message to a publish/subscribe queuing system

The internal architecture of the SLA management is represented in Figure 22

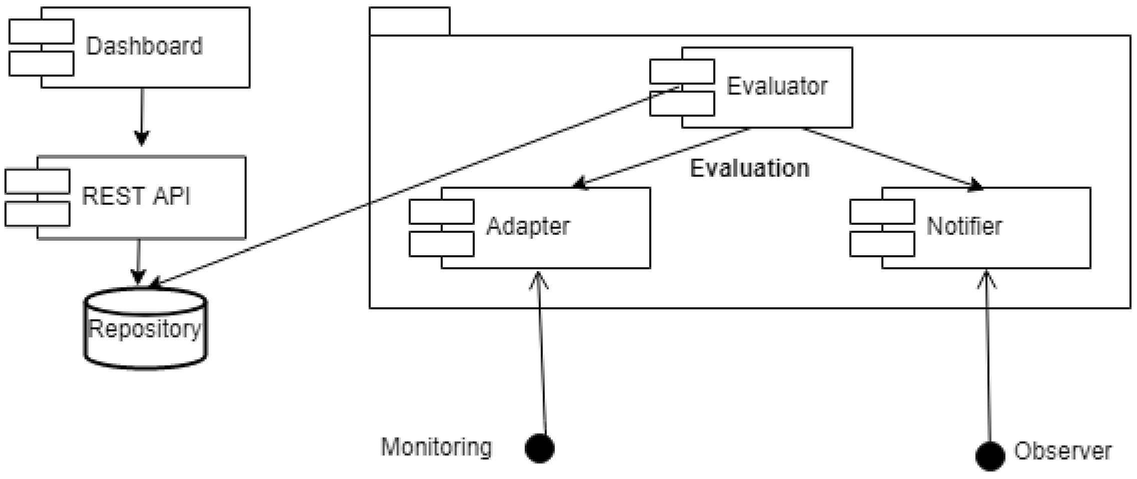

Figure 22: SLA Management detailed architecture.

6.4 Integration Approach

Our proposed architecture is highly modular and loosely coupled. There are a few integration points that should be implemented to arrive at our integrated platform vision:

- Job Descriptor: JSON structure sufficient to create the Argo YAML template and scheduling and SLA management hints;
- Information flow between the SLA Manager and Scheduler should be defined;
- Facades for Persistent Notification Service, Scheduler, Invoker, and Cache services should be defined and implemented.

7 Conclusions

We presented the initial design and specifications for the CloudButton Toolkit and CloudButton Core, the relevant SOTA for the components comprising their functionality, design and implementation of our initial prototypes and outlined our next steps.

Throughout this reporting period, we gained a significant understanding of the motivating use cases as well as a hands on experience with the key enabling technologies. Our next steps will be focusing on an iterative improvement and detalization of the proposed architecture and evolving of the initial disintegrated prototypes into a fully capable system.

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] Google, “Knative.” https://cloud.google.com/knative/.

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

[7] AWS, “Amazon Step Functions.” https://aws.amazon.com/step-functions/.

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

[9] OpenWhisk, “Apache OpenWhisk Composer.” https://github.com/apache/incubator-openwhisk-composer.

[10] A. Airflow, “Apache Airflow documentation.” http://airflow.apache.org/. Accessed on June 2019.

[11] Kubeflow, “Kubeflow: The Machine Learning Toolkit for Kubernetes.” https://www.kubeflow.org/.

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

[13] Fission, “Fission WorkFlows.” https://github.com/fission/fission-workflows.

[14] “Apache OpenWhisk.” https://openwhisk.apache.org/.

[15] S. V. I. S. B. R. Eric Jonas, Qifan Pu, “Occupy the cloud: distributed computing for the 99pp. 445–451, 2017.

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

[17] 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/.

[18] 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/.

[19] 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.

[20] “Apache Mesos.” http://mesos.apache.org/.

[21] Docker, “Docker, Swarm Mode.” https://docs.docker.com/engine/swarm/.

[22] Kubernetes, “Kubernetes: Production-Grade Container Orchestration.” https://kubernetes.io/.

[23] “Plasma Object Storage.”

[24] Paul Castro and Vatche Ishakian and Vinod Muthusamy and Aleksander Slominski, “The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry,” 2019.

[25] H. Lee, K. Satyam, and G. C. Fox, ““Evaluation of Production Serverless Computing",” in WOSC3, Jul 2018.

[26] “Serverless Benchmark.” https://serverless-benchmark.com/.

[27] Winder Research, “A Comparison of Serverless Frameworks for Kubernetes: OpenFaas, OpenWhisk, Fission, Kubeless and more.”

[28] Abraham Ingersoll, “The Leading Open Source Serverless Solutions for Kubernetes.” https://gravitational.com/blog/serverless-on-kubernetes/, Feb 2019.

[29] KEDA, “Kubernetes-based Event Driven Autoscaling (KEDA).” https://github.com/kedacore/keda.

[30] Yaron Haviv, “Nuclio.” https://hackernoon.com/nuclio-the-new-serverless-superhero-3aefe1854e9a

[31] Kubeflow, “Understanding Pipelines: Overview if Kubeflow Pipelines.” https://www.kubeflow.org/docs/pipelines/overview/pipelines-overview/.

[32] CloudFlare, “Serverless Computing with Cloudflare Workers.” https://www.cloudflare.com/products/cloudflare-workers/.

[33] WebAsssembly, “WebAssembly 1.0.” https://webassembly.org/.

[34] Breitgand, David, “Lean OpenWhisk: Open Source FaaS for Edge Computing.” https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b.

[35] Breitgand, David, “apache openwhisk meets raspberry pi.” https://medium.com/openwhisk/apache-openwhisk-meets-raspberry-pi-e346e555b56a.

[36] “5G-MEDIA: An Edge-to-Cloud Virtualized Multimedia Service Platform for 5G Networks.” http://www.5gmedia.eu/outcomes/.

[37] Knative, “Knative Eventing Architecture.” https://knative.dev/docs/eventing/.

[38] “Istio: Connect, secure, control, and observe services.” https://istio.io/.

[39] , “Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach.” https://arxiv.org/abs/1903.12221, 2019.

[40] Google, “Cloud Run.” https://cloud.google.com/run/.

[41] IBM, “Announcing Managed Knative on IBM Cloud Kubernetes Service (Experimental).” https://cloud.google.com/run/, Feb 2019.

[42] TRIGGERMESH, Serverless management Platform, “TRIGGERMESH KLR, Knative Lambda Runtime.” https://github.com/triggermesh/knative-lambda-runtime.

[43] Pivotal, “riff is for functions.” https://projectriff.io/.

[44] Fission, “Fission Framework.” https://docs.fission.io/.

[45] Bitnami, "Kubeless." https://kubeless.io/.

[46] OpenFaaS, "OpenFaas Design and Architecture." https://docs.openfaas.com/architecture/.

[47] Alex Ellis, "Scale to Zero and Back Again with OpenFaaS." https://www.openfaas.com/blog/zero-scale/.

[48] Red Hat, "Red Hat OpenShift 4." https://www.openshift.com/.

[49] MDN Web Docs, "Service Worker API." https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[50] P. García-López, M. Sánchez-Artigas, G. París, D. Barcelona-Pons, Á. Ruiz-Ollobarren, and D. Pinto-Arroyo, "Comparison of faas orchestration systems," in 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018, Zurich, Switzerland, December 17-20, 2018, pp. 148–153, 2018.

[51] A. S. Workshops, "Image Processing." https://github.com/aws-samples/aws-serverless-workshops/tree/master/ImageProcessing. Accessed on June 2019.

[52] Microsoft, "Azure Durable Functions." Accessed on 07.04.2019.

[53] NATS, "NATS documentation." https://nats.io/documentation/.

[54] Fission, "Fission Architecture." https://github.com/fission/fission-workflows/blob/master/Docs/architecture.md.

[55] Fission, "Fission Controller Architecture." https://github.com/fission/fission-workflows/blob/master/Docs/controller-design.md.

[56] Fission, "Fission Function Environments." https://github.com/fission/fission-workflows/blob/master/Docs/functions.md.

[57] Manish Pathak, "Using XGBoost in Python." https://www.datacamp.com/community/tutorials/xgboost-in-python.

[58] Argo, "Argo CD - Declarative Continuous Delivery for Kubernetes." https://argoproj.github.io/projects/argo-cd.

[59] Argo, "Argo Events - The Event-Based Dependency Manager for Kubernetes." https://argoproj.github.io/projects/argo-events.

[60] Adrien Trouillaud, "Running Argo Workflows Across Multiple Kubernetes Clusters." https://admiralty.io/blog/running-argo-workflows-across-multiple-kubernetes-clusters/.

[61] Jesse Suen, "Argo Workflows Examples: dag-diamond.yaml." https://github.com/argoproj/argo/blob/master/examples/dag-diamond.yaml.

[62] Rohit Akiwatkar, "Serverless Antipatterns: What Not to Do With AWS Lambda?." https://www.simform.com/serverless-antipatterns/.

[63] Bharath Bhushan and Mayank Ahuja , "Qubole Announces Apache Spark on AWS Lambda." https://www.qubole.com/blog/spark-on-aws-lambda/.

[64] Y. Kim and J. Lin, "Serverless Data Analytics with Flint," CoRR, vol. abs/1803.06354, 2018.

[65] AWS, "Serverless Reference Architecture: MapReduce." https://github.com/awslabs/lambda-refarch-mapreduce.

[66] AWS, " Amazon ElastiCache: Managed, Redis or Memcached-compatible in-memory data store." https://aws.amazon.com/elasticache/.

[67] D. Goyal, "Lambda vs EC2 Cost Comparison." http://www.theabcofcloud.com/lambda-vs-ec2-cost/.

[68] Alvaro, Alda Rodriguez, Fernando, Alvarez, Gabriel, Diaz Lopez, Martin, Evgeniev, and Pancho, Horrillo, "Economics of Serverless." https://www.bbva.com/en/economics-of-serverless/.

[69] A. Eivy, "Be Wary of the Economics of "Serverless" Cloud Computing," IEEE Cloud Computing, vol. 4, pp. 6–12, March 2017.

[70] Adzic, Gojko and Chatley, Robert, "Serverless Computing: Economic and Architectural Impact," in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, (New York, NY, USA), pp. 884–889, ACM, 2017.

[71] O. Biran, D. Breitgand, D. Lorenz, M. Masin, E. Raichstein, A. Weit, and I. Iyoob, "Heterogeneous resource reservation," in 2018 IEEE International Conference on Cloud Engineering (IC2E), pp. 141–147, April 2018.

[72] D. Breitgand, Z. Dubitzky, A. Epstein, A. Glikson, and I. Shapira, "SLA-aware resource over-commit in an IaaS cloud," in 2012 8th International Conference on Network and Service Management (CNSM) and 2012 Workshop on Sy pp. 73–81, Oct 2012.

[73] D. Breitgand and A. Epstein, "Sla-aware placement of multi-virtual machine elastic services in compute clouds," in 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops, pp. 161–168, May 2011.

[74] A. Juan Ferrer, F. HernáNdez, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, et al., "Optimis: A holistic approach to cloud service provisioning," Future Generation Computer Systems, vol. 28, no. 1, pp. 66–77, 2012.

[75] E. Kamateri, N. Loutas, D. Zeginis, J. Ahtes, F. D'Andria, S. Bocconi, P. Gouvas, G. Ledakis, F. Ravagli, O. Lobunets, and K. A. Tarabanis, "Cloud4soa: A semantic-interoperability paas solution for multi-cloud platform management and portability," in Service-Oriented and Cloud Computing (K.-K. Lau, W. Lamersdorf, and E. Pimentel, eds.), (Berlin, Heidelberg), pp. 64–78, Springer Berlin Heidelberg, 2013.

[76] Kubernetes, " Configure Quality of Service for Pods." https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/#qos-classes. Accessed on June 2019.

[77] P. Heidari, Y. Lemieux, and A. Shami, "Qos assurance with light virtualization - a survey," in 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 558–563, Dec 2016.

[78] V. Medel, O. Rana, J. A. Banares, and U. Arronategui, "Modelling performance resource management in kubernetes," in 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), pp. 257–262, Dec 2016.

[79] Kubernetes, "Configure Multiple Schedulers." https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/.

[80] Kubernetes, "Kubernetes Federation Evolution." https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution/.

[81] METASPACE, “METASPACE: Platform for metabolite annotation of imaging mass spectrometry data.” `https://metaspace2020.eu/`.

[82] “Apache Spark: a unified analytics engine for large-scale data processing.” `https://spark.apache.org/`.

[83] U. EMBL, IBM, “Metaspace annotation pipeline on IBM Cloud.”

[84] Pivotal, “RabbitMQ.” `https://www.rabbitmq.com/`.

[85] Kubeflow, “Build Pipelines with the SDK.” `https://www.kubeflow.org/docs/pipelines/sdk/sdk-overview/`.