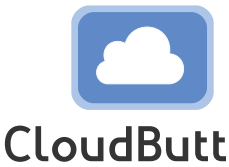

HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D5.1 CloudButton Initial API Definition

Due date of deliverable: 30-06-2019
Actual submission date: 28-06-2019

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	24
WP/Task related to this document	WP5 / T5.1
WP/Task responsible	Imperial
Leader	Peter Pietzuch (Imperial)
Technical Manager	Pedro García (URV)
Quality Manager	Pierre Sutra (IMT)
Author(s)	Simon Shillaker (Imperial), Daniel Barcelona (URV), Gerard París (URV)
Partner(s) Contributing	All
Document ID	CloudButton_D5.1_Public.pdf
Abstract	The CloudButton API covers the entirety of interaction with the system, including deployment time actions such as uploading functions and defining workflows, as well as runtime actions such as interacting with state. We outline our planned approach to all relevant areas and put this in the context of existing APIs and frameworks. In addition, we report on an initial implementation of parts of the API in WebAssembly, called Faasm, that makes it easy to port existing applications to CloudButton.
Keywords	function-as-a-service, API, WebAssembly

History of changes

Version	Date	Author	Summary of changes
0.1	23-05-2019	Simon Shillaker	Skeleton
0.2	31-05-2019	Simon Shillaker	Rough draft
0.3	11-06-2019	Daniel Barcelona	Extended section 4.4
0.4	18-06-2019	Daniel Barcelona	Extended section 4.3
0.5	18-06-2019	Peter Pietzuch	Added new material on Faasm
0.6	20-06-2019	Gerard París	Added new subsection on workflow orchestration with Airflow
1.0	24-06-2019	Simon Shillaker	Reworked Faasm section, general tidy-up and review comments

Table of Contents

List of Abbreviations and Acronyms

API	Application programming interface
AWS	Amazon Web Services
CLI	Command-line interface
GUI	Graphical User Interface
JSON	JavaScript Object Notation
YAML	YAML Ain't Markup Language

Executive summary

The CloudButton project extends serverless beyond traditional use-cases, particularly focusing on large data-intensive applications and big datasets. A major goal of CloudButton is to facilitate the migration of existing workloads, e.g. in the HPC space, to serverless cloud environments. A key enabler for the transparent migration of existing workloads is the *CloudButton API* that we propose. The CloudButton API must be expressive and flexible, yet it must align with prior paradigms and be compatible with emerging standards in the serverless space.

This deliverable describes our initial design for the CloudButton API. It explains how the CloudButton API defines the complete interaction of users and their serverless applications with the underlying CloudButton platform. The API covers both *deployment* time actions such as uploading servereless functions and defining events, as well as *runtime* actions such as the chaining of functions and the interaction of functions with state.

In addition, this deliverable presents our current thinking regarding a general implementation of the CloudButton API, which takes advantage of WebAssembly as a language independent execution substrate. WebAssembly allows us to support a large number of languages using the CloudButton API and execute functions written in these languages transparently on the CloudButton platform. Such an approach facilitates the porting of existing HPC application to the CloudButton platform without having to rewrite an application in a specific language.

To support the CloudButton API with WebAssembly, we describe *Faasm*, a multi-tenant serverless runtime with lightweight isolation, low-latency shared state and efficient resource reuse. Faasm's isolation unit, the *Faaslet*, uses WebAssembly to isolate users' functions in a single thread. According to the CloudButton API, Faaslets allow functions to share data via shared memory regions and reuse much of a function's resources across invocations. Faaslets also support snapshot and restore, allowing users to capture and reuse a function's execution state, thus reducing start-up overheads. Since Faasm functions are compiled to WebAssembly intermediate representation (IR), they offer strong memory safety guarantees and multi-language support out-of-the-box.

1 Introduction

When talking about the API to a serverless system there are two types of interaction: deployment time and runtime. Deployment interactions includes the uploading functions and the definition of events and are usually performed via a commandline interface or GUI; runtime actions include interacting with state and inter-function communication and are provided via language-specific programming abstractions.

Many serverless systems offer extensive deployment APIs. KNative [1], AWS Lambda [2] and OpenWhisk [3] are the most mature and relevant in this context. All three support the definition of functions, their dependencies and triggers, as well as more complex workflows and data processing pipelines. Runtime APIs in these systems are simple and concerned with providing context and access to other systems. AWS Lambda [2] is typical in this regard, providing functions with data about their environment (e.g. resource limits and version number) and the ability to interact with other AWS services on the fly (e.g. reading from an S3 bucket or invoking another Lambda function). A different approach to the runtime API is achieved in PyWren [4], which uses standard Python code and annotations to define parallelizable sections of a program. The code and data for each of these sections is serialized and submitted to AWS Lambda for execution, then the results returned to the client.

Although effective in simple use-cases, these APIs lack the features required to build larger data-intensive applications, which we focus on in CloudButton. APIs better suited to this task can be found in non-serverless systems such as Spark [5] and Tensorflow [6], both offering rich libraries of abstractions for big data and machine learning. Academic work has covered this sort of API in detail, both in the context of managing consistency in systems such as Bloom [7] and complex data flows as in Ray [8].

The CloudButton API will therefore adopt and extend appropriate concepts from both serverless and non-serverless systems. The deployment API will closely resemble those of the major cloud providers in its more basic operations, while introducing more novel concepts for defining workflows; the runtime API will be more similar to those found in a non-serverless context, resembling existing big data and concurrent programming environments.

Our goal with this document is not to provide an exhaustive definition of the final API, rather to outline the key features that it will cover. In Section 2, we provide an overview of the mean features of the deployment API for CloudButton; Section 3 then describes the main features of the runtime API. We finished the deliverable with Section 4, which describes an initial proof-of-concept implementation of parts of the CloudButton API based on the WebAssembly language.

2 CloudButton Deployment API

The deployment API deals with any system interactions that are not part of the actual execution of code. This includes defining the functions and their triggers, as well as dependency and data management. This part of the API will broadly be language-agnostic and accessed via a command line interface.

2.1 Function definition

Along with the source code we must specify a number of other artifacts to make up a complete serverless function. These include the function's library dependencies, the desired language runtime and its resource requirements. The format of such a definition varies across platforms, but has been somewhat standardised by higher-level frameworks such as the Serverless Framework [9]. The Serverless Framework employs a generic YAML file format to define aspects of functions common to all platforms, as well as some platform-specific features. The use of a YAML file in this instance makes sense and is something we will adopt for CloudButton. The format is easy to understand and makes it simple to manage a function's definition in source control.

An example of this format may be as follows:

```
name: my-function

environment:
  runtime: python
  env:
    MY_VAR: 123

functions:
  my_func:
    entrypoint: my_func_entrypoint
    events:
      - http:
          type: GET
          url: /my_funcs/
```

Listing 1: Simple YAML definition of a function

2.2 Dependencies

The most basic deployment API interactions centre around creating, updating and deleting functions. Function definitions in CloudButton will consist of a YAML file as outlined above, along with an archive containing the source code and any relevant resources. This format is common but the specifics vary across systems. In some cases the archive must contain the function plus its complete dependency tree and ABI, as with the C++ runtime on AWS Lambda [10]. SOCK [11] outlines a language-aware mechanism for sharing dependencies between functions, while IBM CloudFunctions provides a whilelist of pre-installed libraries [12]. AWS Lambda offers Layers [13], which lets users compose a function's environment from filesystem layers that are stored independently and duplicated across functions.

In CloudButton, we will support both language-specific package managers such as Pip for Python and npm for NodeJS, as well as a concept similar to Lambda's Layers. This will minimize the size of the deployment bundle by maximising sharing of dependencies across functions. The mechanism for defining both will be covered in the YAML file format mentioned above.

```
name: my-function

environment:
```

```
  runtime: python

dependencies:
  libraries:
    - numpy
    - requests
  layers:
    - my_custom_filesystem_layer
```

Listing 2: YAML definition of function dependencies

2.3 Event sources and triggers

Defining events and their associated triggers is another key part of the deployment API. The universe of possible event sources is large and varies widely across platforms, mainly being determined by the provider's other cloud-based offerings (e.g. S3 events in AWS Lambda). Definition of events and triggers is usually bundled with the function definition itself as in AWS Lambda and OpenWhisk. KNative Eventing [14] defines several high-level primitives from which event sources and consumers can be constructed. These include brokers, triggers and subscribers, and give a good foundation on which to build more complex event streams.

To remain flexible and support a range of events the CloudButton events API will adapt the KNative Eventing framework, using composable primitves from which to construct event sources and triggers. These definitions will take the form of YAML files very similar to those outlined in the KNative documentation.

```
kind: Trigger
metadata:
  name: my-trigger
source:
  kind: PubSub
  topic: my-topic
spec:
  subscriber:
    kind: Function
    name: my-function
```

Listing 3: YAML definition of a trigger and subscriber

2.4 Workflow definition

Support for workflows in current platforms exists in the context of data processing pipelines and sequential downstream applications. This is exemplified by AWS Step Functions [15] which lets users define a state machine composed of Lambda functions and other AWS services in a static JSON file. KNative Eventing can also be used to define static workflows which connect to different event sources ahead of time. Openwhisk Composer [16] takes a different approach, defining workflows in an imperative style in Javascript. This Javascript is then used to generate a "composition", which can be static but also contain dynamic elements via "conductor actions".

Defining workflows ahead of time has its benefits including simpler optimization and ease of understanding. Deployment workflows expressed as YAML files fit well with the rest of the CloudButton API, allowing users to express their whole system in a single format. The major downside to defining deployment workflows is their rigidity and simplicity, particularly when dealing with existing applications written for more dynamic platforms like Spark. The CloudButton API will support deployment workflow definitions, as well as providing a set of programming abstractions for dynamic workflow definition in code. It is likely these two approaches will not be mixed and different

use-cases will call for one or the other.

2.5 Data management

Managing data ahead of time is straightforward but worth discussing for completeness. This task is generally handled outside of the serverless-specific API, instead depending on the given provider's other cloud offerings (e.g. AWS S3).

The CloudButton API will define a simple interface for uploading and querying existing datasets, plus defining access restrictions for specific functions. Additional scope for defining dependencies between data and functions will be included, offering flexibility in scheduling and caching further down the line.

3 CloudButton Runtime API

The runtime API is for functions to interact with the system at runtime via programming abstractions. These are language-specific and provided as part of a client-side library, although the underlying concepts are not tied to any given language. As with the deployment API, our goal here is not an exhaustive definition of the final API, rather to outline some key concepts that the API will include.

3.1 Function input and output

All serverless platforms support some kind of function input and output data, usually in the form of serializable dictionaries or structs. Examples of this are found in all of the major serverless platforms [2, 3, 1]. This is a simple interface aimed at handling small amounts of data, with the contents being passed over the wire in the queuing/messaging infrastructure.

Such an interface is useful in simple functions, hence will be included in the CloudButton runtime API, however, it is unsuitable for the many of the big data applications we will be targeting. To enable passing larger inputs and outputs, the runtime API will define an interface backed by object storage, as well as functions for interacting with arbitrary storage identifiers.

A simple example of using the runtime API in Java is shown in Listing 4.

```java
import cloudbutton.function.Call;
import cloudbutton.function.Context;
import cloudbutton.function.Handler;

import cloudbutton.data.Object;
import cloudbutton.data.ObjectStorage;

public class MyFunction implements Handler {

    @Override
   public int handleCall(Call c, Context ctx) {
      // Load input from object storage
      Object largeInput = c.inputFromStorage();

      // Load some arbitrary storage ID
      Object someOtherData = ObjectStorage.loadByIdentifier("my-data");

        // Perform some work and write large output
      Object largeOutput = myTransformation(largeInput, someOtherData);
      c.outputToStorage(largeOutput);

      return Call.SUCCESS;
    }

}
```

Listing 4: A function performing simple input and output via object storage. The `MyFunction` class encapsulates all behaviour of this specific user-defined function. `handleCall` is the entrypoint that is called on each request to this function. The lifecycle of function classes is handled by the CloudButton runtime.

3.2 Chaining functions

Chaining functions is the process by which one function calls another dynamically. Although some form of chaining is common in many platforms, it is often a "call-and-forget" mechanism, where functions run sequentially without acting on the results of the chained calls. This is true in AWS Lambda where chaining is done by making a call to the standard AWS client library from within the calling function [2]. Existing platforms that support more complex chaining arrangements do so in an deployment context, as in AWS Step Functions [15], KNative [1], and Openwhisk Composer [16].

Table 1: State API

CATEGORY	API	DESCRIPTION
Object	`@Shared`	*Atomic shared object*
	`@Partitioned`	*Partitioned object's state*
Method	`@ReadOnly`	*Locally cached object*
Data structure	`AtomicInteger`, `Counter`, `HashMap`, `Cluster`, ...	*Generic objects annotated with `@Shared` and `@Partitioned`*
Synchronization	`Barrier`, `Future`, `Lock`	*Objects for distributed worker-to-worker coordination*

Simple sequential chaining is insufficient for the CloudButton API where we would like to build more complex workflows at runtime. For handling control flow, the API will define a set of primitives similar to those found in traditional concurrent and asynchronous programming environments such as Java's concurrent package [17] or Python's asyncio [18]. These will include locks, barriers and futures, as well as a more generic fork/join approach to spawning new function calls.

3.3 State

Most serverless platforms employ an explicitly stateless model, hence support for stateful functions and handling of distributed data is uncommon. Some simple examples of state management do exist, for instance PyWren [4] provides convenience methods on top of AWS S3 to connect functions with the data they are processing, while SAND [19] outlines a simple caching layer for shared data. Non-serverless platforms are more relevant in this case, such as Bloom's high-level flexible consistency abstractions [7] and Ray's domain-specific annotations for distributed machine learning [8].

The CloudButton runtime API will provide language-specific libraries for handling large-scale distributed data, necessary for many of the use-cases we are targeting. These libraries will consist of data structures and annotations to support transparent access to cluster-wide shared data from multiple concurrent functions. These libraries will include a set of annotations and objects allowing users to specify their own consistency guarantees on a per-object basis.

We are already working on a preliminary library for the Java language. Table 1 summarizes the most remarkable elements.

The library already includes a set of base shared objects to support mutable shared data across serverless functions. This group consists of common objects such as integers, counters, maps, lists and arrays. The `@Shared` annotation also gives programmers the ability to craft their own custom shared objects. The library refers to an object with a key crafted from the field's name of the encompassing object. The programmer can override this definition by explicitly writing `@Shared(key=k)`. Shared objects can be differentiated between ephemeral and persistent. Unless otherwise stated, shared objects are ephemeral and they only exist during the application lifetime. Once the application finishes, they are discarded. Nonetheless, it is also possible to make them persistent with the annotation `@Shared(persistent=true)`. In such a case, the annotated object outlives the application lifetime and is only removed from storage by an explicit call. The programmer can also decide the consistency policy through the `@Shared` annotation (see Listing 5).

As an optimization, we can apply the `@ReadOnly` annotation to methods of a `@Shared` object. A call to a `@ReadOnly` method triggers the execution of the method on a local copy of the object, instead of invoking it remotely (the default behavior). The primary intend of this annotation is to provide fast access to objects whose state will no longer change by caching them at workers.

```
import cloudbutton.function.Call;
import cloudbutton.function.Context;
import cloudbutton.function.Handler;

import cloudbutton.data.Array;
import cloudbutton.data.Consistency;
import cloudbutton.data.Matrix;

public class MyFunction implements Handler {

   @Shared(key="some-matrix", consistency=Consistency.STRONG)
   private Matrix mat;

   @Shared(key="some-output", consistency=Consistency.EVENTUAL)
   private Array output;

   @Override
   public int handleCall(Call c, Context ctx) {
      // Load input array
      Array inputArray = c.input().toArray();

      // Perform the multiplication
      Array result = inputArray.dot(this.mat);

      // Persist the result
      this.output.update(result);

      return Call.SUCCESS;
    }
}
```

Listing 5: Multiplying a source matrix by an input array, persisting the result to a shared eventually consistent array

The `@Partitioned` annotation improves the scalability of a `@Shared` object at the cost of consistency. In detail, the library replaces at compile time a `@Partitioned` object with a collection of distinct fragments, or *shards*, of the same type. For instance, annotating a `Map` object distributes its content between smaller maps, using the key of each relation for distribution. Each shard behaves as a `@Shared` object. The number of shards is passed as a parameter of the annotation and is thus configurable by the programmer. Methods that read the full state of the object (e.g. `size`) are replaced with appropriate implementations that access all the shards. An example of using `@Partitioned` to implement word count is shown in Listing 6.

To provide fine-grained coordination of serverless functions, the library offers a number of primitives such as cyclic barriers and semaphores (see Table 1). These coordination primitives are semantically equivalent to those in the standard `java.util.concurrent` library. They allow a coherent and flexible model of concurrency for serverless functions that is, as of today, nonexistent. An example of using `@Shared` to implement k-means is shown in Listing 7.

The design of this library is detailed in depth in deliverable D4.1.

3.4 Higher-level abstractions

Higher-level abstractions for operating on state are non-existent in a serverless context, but common in non-serverless platforms. These abstractions allow users to define sequences of operations on distributed data through an imperative programming model. Spark's API centered around operations on RDDs is a good example of this type of high-level abstraction [20], and allows the user to easily

```
public class Wordcount implements Runnable {
    @Paritioned(key="words", persistence=True)
    MergeableMap<String, Long> globalWords = new MergeableHashMap<>();

    public void run (){
        List<String> lines = getLines (workerId, nWorkers);
        Map<String, Long> data = countWords(lines);
        globalWords.mergeAll(data, Long::sum);
    }

    private Map<String, Long> countWords(List <String> lines) {...}
    private List<String> getLines(int worker, int nWorkers) {...}
}
```

Listing 6: A word count implementation uses partitioned shared state. Using persistence we ensure that the data will be available for further processing (e.g., searching the most used words).

```
public class KMeans implements Runnable{
    private CyclicBarrier barrier = new cloudbutton.synch.CyclicBarrier();
    @Shared(key="delta")
    private GlobalDelta globalDelta = new GlobalDelta();
    // Wraps a list of @Shared centroids
    private GlobalCentroids centroids = new GlobalCentroids();

    public void run(){
        loadDataset();
        int iterCount = 0;
        do {
            correctCentroids = globalCentroids.getCorrectCoordinates();
            resetLocalStructures();
            localDelta = computeClusters();
            globalDelta.update(localDelta);
            centroids.update(localCentroids, localSizes);
            barrier.await();
            iterCount++;
        } while (iterCount < maxIterations && !endCondition());
    }
}
```

Listing 7: *k*-means is an iterative algorithm that needs worker synchronization at each iteration.

```
ServerlessExecutor taskExecutor = ServerlessExecutor(nLambdas);
IntStream.range(0, nLambdas).forEachOrdered(n -> {
    taskExecutor.execute(new MyRunnable());
});
taskExecutor.awaitTermination(timeout, TimeUnit.NANOSECONDS);
```

Listing 8: Performing the fork/join pattern using a serverless `ExecutorService`.

```
ServerlessExecutor se = new ServerlessExecutor();
se.invokeParallelFor(MCPi.class, nWorkers, fromInclusive, toExclusive);

public static class MCPi extends ParallelFor{
    @Shared(key = "piCount")
    Counter count = new Counter();
    Random prng = new Random();
    long partial = 0;

    public void loop(long index){
        double x = prng.nextDouble();
        double y = prng.nextDouble();
        if (x*x + y*y <= 1.0) partial++;
    }
    public void finalize(){
        count.add(partial);
    }
}
```

Listing 9: Implementing a Monte Carlo simulation to approximate π using a parallel for abstraction to distribute a loop.

express complex operations on distributed data.

The CloudButton runtime API will provide its own set of high-level abstractions for the execution and coordination of serverless functions that may share state using the data structures outlined in the previous section. These will act much like operators on Spark's RDDs, dispatching a set of serverless functions to perform tasks in parallel which update the shared data (e.g. in model training in ML algorithms). By expressing these through standard imperative constructs, the process should be transparent to the user.

Coordinating such operations require synchronization primitives to manage concurrent execution (such as barriers, locks and futures). Moreover, concurrent applications usually follow common patterns to distribute tasks (such as parallel for) and sometimes with recurrent synchronization points (such as the fork/join pattern and iterative tasks).

This approach works very closely with the idea of programming serverless functions like threads. With this abstraction in mind, we envision a programming framework where function calling is like running a local thread. Using Java jargon, our toolkit includes a serverless `ExecutorService`, through which the user can submit tasks and synchronize their completion from a controller. Listing 8 shows a simple example of this, while Listing 9 shows a more complex example implementing a Monte Carlo simulation.

3.4.1 ServerlessExecutor API

One core principle behind CloudButton toolkit is programming simplicity. Our focus is to make this tool as usable as possible, irrespective of whether the programmer is a cloud expert or not.

We will devote extra efforts to integrate the CloudButton toolkit with other tools (e.g. Python notebooks such as Jupyter [21]), which are very popular environments for the scientific community.

Python notebooks are interactive computational environments, in which you can combine code execution, rich text, mathematics, plots and rich media. To this aim, IBM Cloud contains a service called IBM Watson Studio [22], that, among other things, allows users to create and execute notebooks in the cloud, where CloudButton toolkit can be very easily imported to run big data analytics or embarrassingly parallel jobs.

A ServerlessExecutor instance will provide access to all the available methods in the API. The API will be comprehensive enough for novice users to execute algorithms out of the box, but simple enough for experts to easily tune the system by adjusting important knobs and switches (parameters). At the beginning there will be three main methods to execute the user's code trough a serverless platform: `call_async()`, `map()` and `map_reduce()`. Also, there will be two different methods to monitor the executions or get the results: `wait()` and `get_result()`.

To cater for average users, the ServerlessExecutor will be utterly simplified, so that the typical behavior of calling a computing method (e.g., `map()`) followed by a call to `get_result()` to retrieve the results does not require dealing with complex artifacts. Indeed, the readiness of the results will be internally managed. To deliver a finer control, all the three computing methods, however, will also return *future*[1] objects to track the status of the executors and get the results when available.

Let us review succinctly the main proposed methods of the ServerlessExecutor API.

▷ `call_async()`. The first proposed method will be used to run asynchronously just one function in the cloud. This method will be non-blocking, i.e., the sequential execution of the local code continues without waiting for the results. The parameters of this method will be the `function_code` and the input `data` that the function executor receives.

Parameter	Description
`function`	callable method
`op_args`	Function arguments, as a dictionary (key = parameter name, value = parameter value)

▷ `map()`. The second proposed method is called `map()`. This method will be used to run multiple function executors. This method will also be non-blocking and will take as main input the `map_function_code` and the `data` that the map function executors receive. Unlike the prior method, this one will receive as input `data` a list the number of parallel functions to spawn, alongside with the input parameters that should be sent to the functions.

Parameter	Description
`map_function`	callable method
`op_args`	Function arguments as a dictionary. Compulsory key: `'iterdata'`, where the value is the iterable parallelizable data (list, dictionary, bucket names, etc.).
`chunk_size`	Size (in Bytes) of the data chunks. Default: `None` (map per file).

▷ `map_reduce()`. The third proposed method will be used to execute MapReduce flows, i.e., multiple map function executors (map phase), and one or multiple reduce function executors (reduce phase). This method will also be non-blocking. It will take as input the `map_function_code`, the input `data` as a list of values, and the `reduce_function_code`. As in the prior method, it can spawn the desired number of mappers and reducers.

[1] We will mimic, for example, the Python 3.x futures interface (`https://pythonhosted.org/futures/`).

Parameter	Description
`map_function`	callable method.
`reduce_function`	callable method.
`op_args`	Function arguments as a dictionary. Compulsory key: 'iterdata', where the value is the iterable parallelizable data (list, dictionary, bucket names, etc.).
`chunk_size`	Size (in Bytes) of the data chunks. Default: None (map per file).
`reducer_one_per_object`	Invoke a reducer for every object after partitioning. Default: False
`reducer_wait_local`	Wait for results locally. Default: False

▷ `wait()`. On the client side, the ServerlessExecutor will offer a method to monitor the executions. This method will be called `wait()`. It will be synchronous, i.e., the local user code is blocked until the call to `wait()` ends. It will provide a configurable parameter to decide when to release the call and continue the execution. Moreover, a user will be able decide to unlock the method in three different circumstances: 1) '`Always`': it will check whether or not some result is available on the invocation of `wait()`. If so, it returns them. Otherwise, it will resume the local execution; 2) '`Any completed`': it will resume the local execution upon termination of any function invocation; and 3) '`All completed`': it will wait until all the functions have finished they execution and the results are available. In these three cases, the `wait()` method will return a 2-tuple of lists: the first list with the futures that completed and the second with the uncompleted ones.

Parameter	Description
`timeout`	Time to wait for functions completion
`unlock`	When to unlock the method: always, any_completed, all_completed

▷ `get_result()`. This method will be used to collect the results from the functions when a parallel task has finished (e.g., `map()`, `map_reduce()`, etc.). It will add some functionality such as timeout support, keyboard interruption to cancel the retrieval of results, and a progress bar to inform users about the % of task completion. Last but not least, this method will be *composition-aware*: it will transparently wait for an on-going function composition to complete, just returning the final result to users.

3.4.2 Data discovery and partitioning in `map_reduce()`

To provide good MapReduce support, an important key ingredient will be to provide a built-in data partitioner in the platform to abstract users from this arduous, prone to error task. Of course, this support will be given to the `map()` and `map_reduce()` methods, along with a useful data discovery mechanism.

In particular, the only action that a user will have to do is to supply the list of object keys that compose the dataset. However, as a dataset may contain hundreds, or even thousands of files, it will be possible to specify the name an object storage bucket(s) that will contain all the objects in the dataset instead. In the latter case, the ServerlessExecutor will be responsible for discovering all the objects in the bucket(s), and partition them.

The data discovery process will be automatically started when a bucket is specified instead of a list of objects. It will consist of a HEAD request over each bucket to obtain the necessary information to create the required data for the execution.

Once the data discovery process has ended, the data partitioner will initiate to produce the data partitions based on a configurable chunk size parameter. Each data partition will be automatically assigned to a function executor, which applies the map function to the data partition, and finally writes the output to the object storage service. The partitioner then will execute the reduce function. The reduce function will wait for all the partial results before processing them.

Additionally, it will allow to have results computed by multiple reducers. By default, the `map_reduce()` method will use a single reducer for all the partitions of the dataset. Moreover, it will be possible to increase the number of reducers and make `map_reduce()` behave as a kind of Spark's `reduceByKey()` operator by setting the parameter `reducer_one_per_object=True`. So when this parameter is enabled, all the values for the same object key are combined in a separate reducer. This feature is very useful, since very often, it is necessary to produce a different result for every object in the dataset.

3.4.3 Workflow orchestration with Airflow

It's also possible to use a declarative progamming model to define sequences of operations. We've performed some exploratory work with Apache Airflow [23], 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 combination of Airflow's DAGs definition that follows the declarative programming paradigm, with serverless function's capability of executing massive parallel tasks, raises the opportunity of creating declarative DAGs with tasks that take advantage of serverless function's ease of computation for parallel tasks while taking away the computational load from the host machine that executes Airflow. Our current implementation of the IBM Cloud Functions Airflow Plugin[2] includes 3 operators:

`IbmCloudFunctionsBasicOperator`
Invokes a single function bvy using the serverlessexeutor.call_async() API method.

`IbmCloudFunctionsMapOperator`
Invokes multiple parallel tasks by using the serverlessexeutor.map() API method.

`IbmCloudFunctionsMapReduceOperator`
Invokes multiple parallel tasks with the desired number of reducers by using the by using the serverlessexeutor.map_reduce() API method.

Listing 10 shows a simple example of an Airflow workflow that uses `IbmCloudFunctionsBasicOperator` (invokes a single serverless function) and `IbmCloudFunctionsMapOperator` (invokes multiple parallel functions).

3.5 Porting legacy code

Porting legacy parallel code to a serverless context is challenging, especially when targeting multiple languages. For example, FaaS-ifying a standard fork/join operation in legacy code must translate to spawning a new serverless function that executes in its own sandbox potentially on another host, then returning control back to the caller. This requires not only packaging code and dependencies, but also transmitting global variables and arguments from the context of the parent process. Certain languages are more amenable to this than others—e.g. a serialisable Java `Callable` can wrap a new serverless function call, and Python functions can be serialised and executed remotely as in PyWren [4].

With the CloudButton API, we aim for minimal intervention when porting existing code. For this, we following a two-pronged approach:

1. We will approach the problem in the context of legacy Java applications, adapting existing code to use some of our new stateful data structures, as well as creating an interface for submitting serialized Callables on the fly.

[2] `https://github.com/aitorarjona/ibm_cloud_functions_airflow_plugin`

```
import airflow

from airflow.operators.ibm_cloud_functions_plugin import IbmCloudFunctionsBasicOperator
from airflow.operators.ibm_cloud_functions_plugin import IbmCloudFunctionsMapOperator

from functions import example_functions

args = {
    'owner': 'airflow',
    'start_date': airflow.utils.dates.days_ago(2),
    'provide_context' : True
}

dag = airflow.models.DAG(
    dag_id='ibm_cf_simple_example',
    default_args=args,
    schedule_interval=None,
)

# Generates a list of 10 random numbers between 1 and 100
generate_list = IbmCloudFunctionsBasicOperator(
    task_id='generate_list',
    function=example_functions.generate_list,
    dag=dag,
)

# Generates a random number between 1 and 100
generate_random = IbmCloudFunctionsBasicOperator(
    task_id='generate_random',
    function=example_functions.generate_random,
    op_args={'max' : 100},
    dag=dag,
)

# Adds the random number to each element of the list
map_add = IbmCloudFunctionsMapOperator(
    task_id='map_add',
    map_function=example_functions.add,
    op_args={'iterdata' : 'FROM_TASK:generate_list', 'x' : 'iterdata', 'y' :
        'FROM_TASK:generate_random'},
    dag=dag,
)

# Adds all the numbers of the list
add_list = IbmCloudFunctionsBasicOperator(
    task_id='add_list',
    function=example_functions.add_list,
    op_args={'results' : 'FROM_TASK:map_add'},
    dag=dag,
)

# DAG declaration
[generate_list, generate_random] >> map_add >> add_list
```

Listing 10: Example of an Airflow workflow

2. We will also investigate an implementation of the CloudButton API with WebAssembly. A large number of modern programming languages can be automatically translated to WebAssembly, which would therefore give the CloudButton platform the ability to execute existing applications written in different languages natively according to a FaaS model. In the next section, we report on the proof-of-concept implementation of our WebAssembly-based runtime for CloudButton, which is called Faasm.

4 Faasm: Supporting the CloudButton API using WebAssembly

In this section we outline *Faasm*, a serverless runtime with a new approach to isolation, shared state and resource reuse that includes a partial implementation of the CloudButton API in WebAssembly. Faasm combines *software fault isolation (SFI)* as offered by WebAssembly, with OS support and a virtualized host interface to achieve strong isolation guarantees with low overheads while implementing a function-as-a-service model. Based on the CloudButton API, Faasm can offer performance, cost and efficiency gains over other existing FaaS runtimes that follow a purely container-based approach. As a specific technical innovation, Faasm leverages shared memory regions to achieve low latency state sharing, coupled with flexible consistency guarantees.

4.1 WebAssembly

WebAssembly is a portable low-level bytecode built primarily as a successor to Native Client [24] and asm.js [25]. WebAssembly initially focused on the execution of high performance, multi-tenant code on the web, but its usefulness extends far beyond the browser. This is evidenced by Mozilla's WebAssembly System Interface (WASI) [26], which aims to standardise host interaction in server-side WebAssembly.

Although WebAssembly is a compilation target that is not written directly, it has a readable text representation that gives a good intuition for the format. Listing 11 shows the text representation of a WebAssembly module that exposes a function `addFun`, which adds two to its first argument.

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export 'memory' (memory $0))
  (export 'addFun' (func $addFun))
  (func $addFun (; 0 ;) \
     (param $0 i32) (result i32)
    (i32.add
      (get_local $0)
      (i32.const 2)
    )
  )
)
```

Listing 11: Sample WebAssembly function

We take advantage of several important properties of WebAssembly to achieve our goals:

Security. The WebAssembly specification has undergone formal verification from the start and has strict memory safety guarantees [27]. WebAssembly modules can be validated in a single pass, and runtime traps cover any aspects not caught in static analysis. This allows us to safely execute multi-tenant code side-by-side in a single runtime process.

Host interaction. WebAssembly is agnostic to the underlying target, making it suitable for everything from browsers to standard OSs. Each runtime must provide its own virtualized system call interface and optional client libraries that are best suited to the task. This makes it easy to define and restrict a secure serverless-specific host interface.

Memory. The memory accessible by a WebAssembly module consists of regions of linear byte arrays. At runtime the execution environment must provision these arrays and make them available to the executing modules. Not only is this very simple and easy to reason about, but it gives the runtime fine-grained control over a module's memory before, during and after execution.

Multi-language support. Being an IR that works well with LLVM, WebAssembly can easily support any language with a readily available LLVM back-end. C and C++ are particularly well

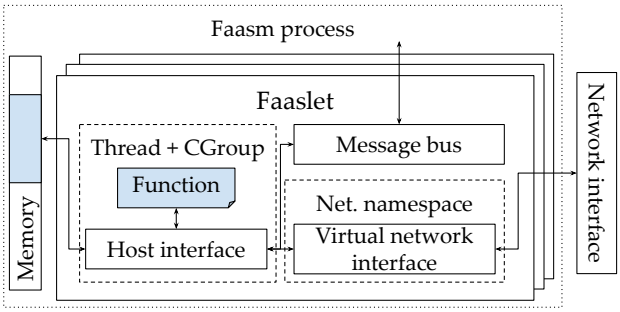

Figure 1: Functions in Faasm execute inside Faaslets. The Faaslet allows the function to interact with the underlying host exclusively through the host interface, which controls access to memory and other system resources. Faaslets operate as threads of a single runtime process, with each provisioning its own disjoint region of the shared process memory.

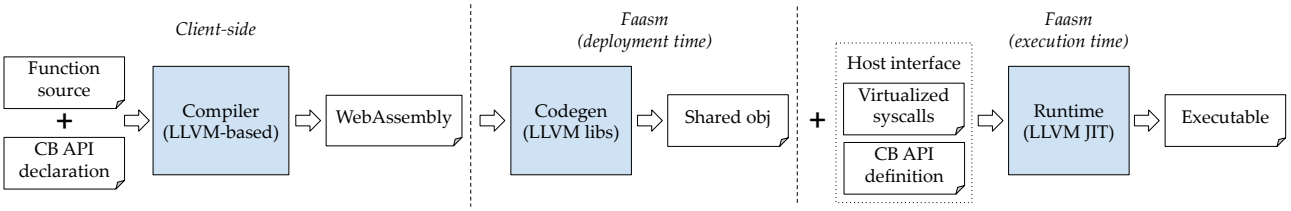

Figure 2: The Faasm code generation and execution pipeline. Source code is compiled on the client-side along with the language-specific CloudButton API declaration (e.g. a header file in C/C++) to produce a WebAssembly binary. On upload to Faasm, a shared object is created and stored. At execution time this shared object is linked with the host interface (including the CloudButton API definition) to produce the final executable.

supported with Typescript and Rust also gaining in popularity. The list of well supported languages will expand over time and makes multi-language support straightforward for existing runtimes.

Dynamic linking. WebAssembly modules can share both functions and data, with a well defined runtime linking interface. This means applications can be split up and shared across multiple distributable modules. This is important for replicating dynamic linking in existing applications as well as improving efficiency in sharing serverless dependencies.

4.2 Faasm and the CloudButton API

Faasm executes all functions inside its own isolation abstraction called a *Faaslet*. Faaslets are reused for repeat invocations of the same function, then destroyed when no longer needed. The architecture is shown in Figure 1. The most important part when considering the CloudButton API is the *Host Interface*, this defines both a virtualized syscall layer and any other runtime-related functions. This is where the CloudButton API is implemented in Faasm.

4.2.1 Code generation

To produce an executable, the user's source code must first be compiled to a WebAssembly binary which is then uploaded using the CloudButton deployment API. Once uploaded, Faasm will generate and store a shared object file from this source. At runtime this object file is loaded and linked with the host interface to produce an executable. This process is outlined in Figure 2.

Faasm relies heavily on WAVM [28] for handling code generation and execution. WAVM is an

open-source WebAssembly virtual machine (VM), which conforms to the WebAssembly specification tests [29]. Part of the ahead-of-time code generation includes injecting runtime traps that catch invalid operations, the nature of these traps is also part of the WebAssembly specification.

At runtime WAVM uses LLVM just-in-time (JIT) libraries to dynamically load the object file and produce an executable module. The host interface functions and CloudButton API definitions are wrapped in thunks and passed along with the required memory and global variable definitions to the JIT, which is then used to execute the function. This JITing only happens once per Faaslet, after which the Faaslet is bound to that function. Once a Faaslet is bound to a function it can be reused for repeat executions.

4.2.2 Memory safety

Secure execution within Faaslets relies heavily on the memory safety guarantees afforded by the WebAssembly spec, coupled with a strictly defined host interface.

Functions compiled to WebAssembly can be verified according to the WebAssembly specification [29], thus giving us strong guarantees that they will remain within their own region of linear memory. The bounds of this memory are also checked at runtime with traps as defined in the spec.

Any interaction with the underlying host is performed through the host interface. Along with the CloudButton API, the host interface provides a whitelist of syscalls and other functions through which the code can perform tasks such as memory management, basic networking and virtualised file I/O.

The security of the host interface is down to the specific implementation of each function, and not covered by the WebAssembly specification. As a result, keeping this interface minimal and simple is crucial to ensuring secure execution.

4.2.3 CPU and network

Each Faaslet uses its own OS thread to execute function code. To ensure each function has fair access to CPU and network resource, this thread is assigned to the relevant `cgroup` and network namespace, through which it communicates with a virtual network interface. We apply further traffic shaping to this interface to ensure no function consumes too much bandwidth. This is similar to the mechanisms used in standard Docker containers.

4.3 State API

The Faaslet host interface exposes a key-value-like API through which functions can share state. All functions belonging to the same tenant share the same key space. These state API functions support both synchronous and asynchronous access, with colocated functions able to access state held in shared memory. A remote key-value store and background synchronization process enables cluster-wide sharing.

To ensure a generic interface and ease integration, all state values are stored as byte arrays. Utilities to make handling these arrays easier are provided by default in the Faasm WebAssembly library. The API is simple yet flexible and outlined below. It is low-level enough to give fine-grained access to those that need it, while others can abstract away details in language-specific libraries.

Synchronous state API. When interacting with state synchronously, calls are made directly to the remote key-value store, with results copied into the Faaslet's memory. This means two functions on the same host can read the same state variable and each get their own copy. Likewise, writes from one function will only be seen by another if they perform another read operation. As well as reading and writing whole values, functions can address subsections of state values too, allowing them to interact with smaller parts of larger pieces of data. Acquiring and releasing global locks on specific keys is supported using the Redis Redlock mechanism [30], although any equivalent distributed lock will suffice.

An outline of the synchronous API functions is shown in Listing 12.

Asynchronous state API. To support weaker consistency and more efficient state sharing, Faasm also provides an asynchronous API. This involves a single copy of data on each host, with colocated

```
// Read a value or a subsection of it synchronously
vector readState(string key)
vector readState(string key, int offset, int len)

// Write to a value or subsection of it synchronously
void writeState(string key, vector val)
void writeState(string key, int offset, vector val)

// Locks
void lockStateGlobal(string key)
void unlockStateGlobal(string key)
```

Listing 12: Faasm synchronous state access

```
// Get a pointer to the given value or subset of it
// int shared memory
byte* readState(string key)
byte* readStateOffset(string key, int offset, int len)

// Tell the runtime the state or a subsection of it
// has been modified
void flagDirty(string key)
void flagDirtyOffset(string key, int offset, int len)

// Force push the whole value or subsections
void pushState(string key)
void pushStatePartial(string key)
```

Listing 13: Faasm asynchronous state access

functions accessing it directly via shared memory. A read call to the asynchronous API returns a pointer to the shared memory region, thus allowing direct access. The Faaslet will perform a remote read if this is the first interaction with the given state on this host, otherwise it will map the existing value.

An outline of the asynchronous API functions is shown in Listing 13.

4.3.1 Shared memory management

Functions executing in Faaslets have their own continuous, disjoint region of shared process memory. When state is read synchronously the remote value is copied into this memory region, and when written synchronously, the relevant call is made to the remote key-value store.

Asynchronous state reads and writes are handled differently. An asynchronous read will cause a read from the remote key-value store if the value is not already present on that host. If the value is present, it is only refreshed if it is deemed "stale". Writes are made directly to this shared memory region, and the background sync process will periodically push these to the remote key-value store. The same is true of partial reads from and writes to this shared memory region.

Pages of shared memory are mapped into a function's existing linear memory region using the underlying OS. This means that the function code still "sees" a linear address space, but writes to the shared regions are mapped accordingly. In this way we still take advantage of WebAssembly memory safety guarantees, while also sharing between functions.

4.4 Snapshot and restore

Another experimental feature of the Faasm API that may prove useful is a snapshot and restore mechanism. This is made feasible by the simple linear memory model of WebAssembly, and hence

```
int main(Faasm *f) {
  if (f->snapExists("mysnap")) {
    f->restoreSnap("mysnap");
  } else {
    initialize();
    f->createSnap("mysnap");
  }

  // Do work

  return 0;
}
```

Listing 14: Faasm snapshotting mechanism. If the function runs and the snapshot `"mysnap"` already exists, we can restore it, otherwise we must call the `initialize()` function and create the snapshot. If the `initialize()` call is expensive, this can noticeably reduce start-up overheads.

only applicable in the context of WebAssembly functions.

Because a WebAssembly module's memory is held in a single contiguous byte array, it is possible to snapshot a function's memory by copying this array. The resulting snapshot can be kept and used to restore the same memory in a different function execution.

This can prove useful in avoiding repetitive initialization on more complex functions, thereby reducing start-up times. The cost of doing this restore can be reduced further by sharing the snapshot via copy-on-write memory rather than creating an actual duplicate every time. Listing 14 shows a simple example of using this API to avoid repeat initialization.

Because we are copying the stack, heap and data for the function itself (and not any OS-specific memory related to the underlying thread), these snapshots can be used to restore across hosts. The mechanism for sharing snapshots across hosts depends on the underlying serverless platform, for example, S3 can be used when running on AWS.

5 Conclusion

In this document we give an overview of the complete API for CloudButton. This is the first unified vision for the complete system, forming an important step in the development of a coherent, usable end product. The API covers both the *deployment* and *runtime* aspects of the system, ranging from creating simple functions through to complex workflows interacting with distributed state.

This is an early specification and many of the underlying components will undergo significant changes over the lifetime of the project. The API will evolve to reflect these changes, but its goal of providing a unified intuitive experience will remain constant.

References

[1] Google, "KNative." https://cloud.google.com/knative/.

[2] Amazon Web Services, "AWS Lambda." https://aws.amazon.com/lambda/.

[3] Apache Project, "Openwhisk." https://openwhisk.apache.org/.

[4] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pp. 445–451, 2017.

[5] Apache Project, "Spark." https://spark.apache.org/.

[6] Google, "Tensorflow." https://www.tensorflow.org/.

[7] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak, "Consistency Analysis in Bloom: a CALM and Collected Approach.," in *CIDR*, pp. 249–260, Citeseer, 2011.

[8] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging {AI} Applications," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 561–577, {USENIX} Association, 2018.

[9] Serverless, Inc., "Serverless Framework." https://serverless.com/.

[10] Amazon Web Services, "CPP Lambda Runtime." https://github.com/awslabs/aws-lambda-cpp.

[11] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), USENIX Association, 2018.

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

[13] Amazon Web Services, "AWS Layers." https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html.

[14] Google, "KNative Eventing." https://knative.dev/docs/eventing/.

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

[16] Apache Project, "Openwhisk Composer." https://github.com/apache/incubator-openwhisk-composer.

[17] Oracle, "Java Concurrent Library." https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html.

[18] Python Software Foundation, "Python asyncio librari." https://docs.python.org/3/library/asyncio.html.

[19] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), USENIX Association, 2018.

[20] Apache Project, "Spark RDD Programming Guide." https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[21] jupyter, "Python notebooks." https://jupyter.org/.

[22] IBM, "Watson studio." `https://dataplatform.ibm.com`.

[23] Apache Software Foundation, "Apache Airflow documentation." `http://airflow.apache.org/`.

[24] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *2009 30th IEEE Symposium on Security and Privacy*, pp. 79–93, may 2009.

[25] Asmjs.org, "asm.js."

[26] Mozilla, "WASI: WebAssembly System Interface."

[27] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, pp. 185–200, 2017.

[28] A. Scheidecker, "WAVM."

[29] WebAssembly, "WebAssembly Specification."

[30] Redis Project, "Redis Redlock."