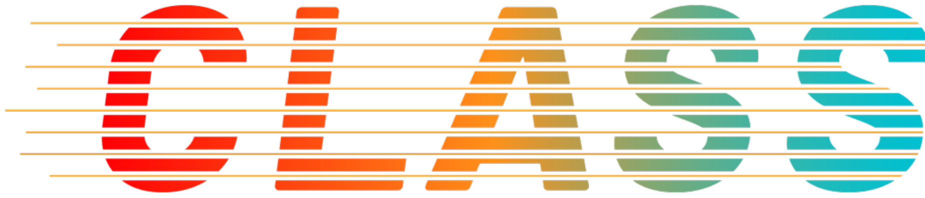
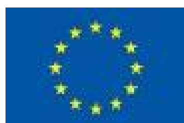


D5.4 Final Release of CLASS Big-Data Analytics Layer

Document Information

Contract Number	780622
Project Website	https://class-project.eu/
Contractual Deadline	M29, May 2020 (Due to COVID situation this deliverable has been submitted on M31, July 2020)
Dissemination Level	PU
Nature	Demonstrator
Author(s)	Erez Hadad (IBM)
Contributor(s)	Jorge Montero Gomez (ATOS)
Reviewer(s)	Roberto Cavicchioli (UNIMORE)
Keywords	Analytics, Serverless, Map, Reduce



Notices: *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No "780622".*

© 2018 CLASS Consortium Partners. All rights reserved.

Change Log

Version	Author	Description of Change
V0.1	Erez Hadad (IBM)	Initial Draft
V0.2	Erez Hadad (IBM)	• Added PyWren optimizations • Finished full-text, pending partner contributions
V0.3	Erez Hadad (IBM)	• Merged ATOS contribution • Added EXPRESS positioning • Full draft ready for review
V0.4	Roberto Cavicchioli (UNIMORE)	Review of the document
V0.5	Erez Hadad (IBM)	Final document
V1.0	Maria A. Serrano (BSC)	Final version, ready to EC revision

CLASS

Table of contents

1 Executive Summary 4
2 Overview 4
3 Technical Specification 6
3.1 EXPRESS – EXtended PREdictability ServerlesS 6
3.1.1 Technical Description 6
3.1.2 Usage 11
3.2 Runtime support for COMPSs 12
3.3 Optimized PyWren with dataClay Support 12
3.3.1 Technical Description 12
3.3.2 Usage 13
3.4 COMPSs 14
3.5 pCEP (ATOS) 14
3.6 DNN at the Edge 14
4 Delivery 14
4.1 Bundling and Installation 14
4.1.1 EXPRESS – Extended PREdictability ServerlesS 14
4.1.2 Runtime support for COMPSs 14
4.1.3 Optimized PyWren with dataClay Support 14
4.1.4 Trajectory Prediction Application 14
4.1.5 Collision Detection Application 14
4.1.6 COMPSs 14
4.1.7 pCEP (ATOS) 14
4.1.8 DNN at the Edge 15
4.2 Demonstration and Impact 15
4.2.1 EXPRESS 15
4.2.2 Trajectory Prediction 15
4.2.3 Collision Detection 16
5 Product Glossary 16
5.1 Apache OpenWhisk 16
5.2 PyWren 17
5.3 COMPSs 18
6 References 19

1 Executive Summary

This document describes deliverable D5.4 “Final release of an augmented platform for WP5 analytics workloads”, which is the final release of the analytics layer of the CLASS software stack, in accordance with Task 5.4 of the CLASS DoA [1], deliverables D5.1 [2], D5.3 [3] and other related CLASS documentation. It marks another successful delivery of the CLASS project as part of milestone MS3, executed in M16-M30.

This milestone has been subject to constraints that have slowed progress behind expectations, have already lead to a 2-month delay, and have been reported separately to the PO. To avoid further delay in delivery, we deliver all planned content under some mitigations that are expected to complete shortly after the milestone, as following:

- The EXPRESS prototype delivered is near completion, and is therefore not integrated yet with CLASS applications. A small application demonstrating its basic capabilities is included, as discussed below.
- The Warning Area / Collision Detection integration with PyWren is close to completion, similar to the already-completed Trajectory Prediction + PyWren integration (which is included with demonstration). So, we deliver a stand-alone demonstration of the core Collision Detection logic as proof of progress.

The document lays out as follows: Section 2 consists of overview and feature description of this deliverable, in alignment with other deliverables and project goals. Section 3 provides a technical specification of each delivered component. Section 4 provides delivery details: per-component bundling/installation, and demonstration/impact where available. Last, Section 5 provides a glossary of key technology products involved in CLASS, to assist in overall understanding of the document.

2 Overview

The analytics layer of the CLASS software stack allows multiple types of big-data analytics back-ends, such as map/reduce, task-based, CEP or DNN, to integrate in a uniform mesh where workloads and components can interact or be invoked via REST, CLI or in response to events. The key to this novel approach is the use of a serverless platform based on Apache OpenWhisk [4] as the foundation, as shown in Figure 1 below.

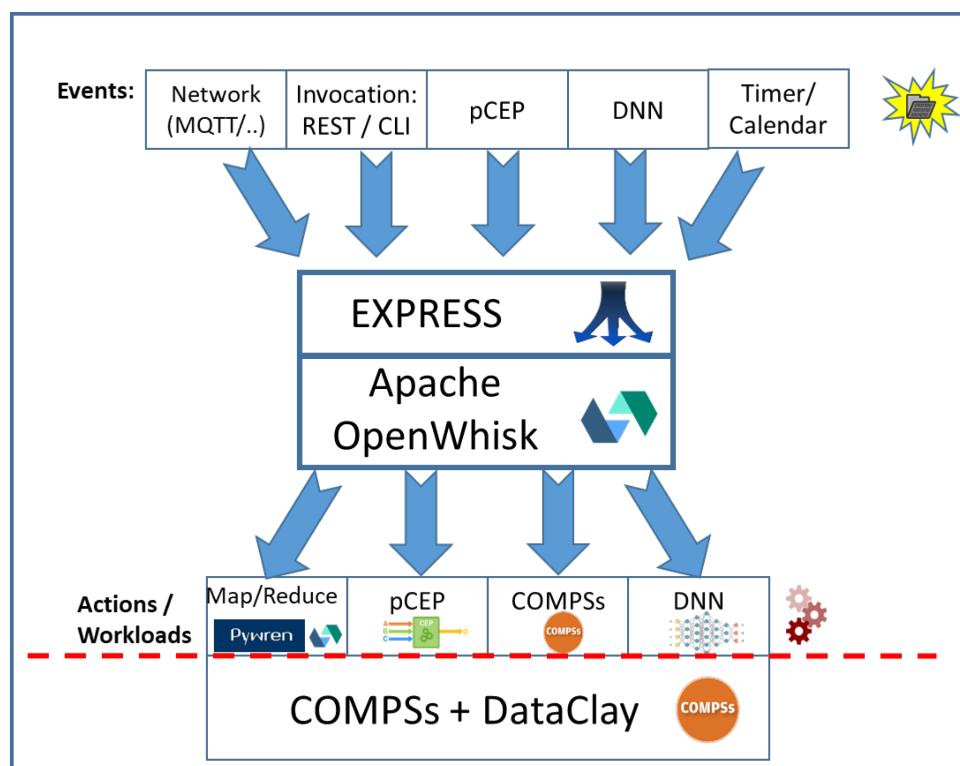


Figure 1. CLASS Analytics Layer Architecture.

In this final release of the analytics layer, in accordance with the high-level design provided in D5.1 [2] and initial release in D5.3 [3], the following features are included:

1. **Core platform:** *EXPRESS* – Extended PREdictability ServerlesS. Unlike the initial release, EXPRESS has been completely redesigned as a portable solution for predictable execution of serverless functions. Being portable, EXPRESS can be deployed on top of an existing serverless platform (which is Apache OpenWhisk in CLASS), at all components of the compute continuum of CLASS (cloud, street node and car node), on top of either Kubernetes [5] or Docker [6].
2. **Runtime support for COMPSs:** same as in D5.3 [3]. We built a container-based runtime for running COMPSs [7] or COMPSs-based workloads as OpenWhisk actions. This also allows relaying the runtime parameters and the *rtp* policy into COMPSs.
3. **Set of analytics back-ends:** comprising of the following back-ends:
    a. **PyWren – serverless Map/Reduce engine**: PyWren [8] is a Map/Reduce engine on top of serverless execution, originally from UC Berkeley and ported to OpenWhisk and extended by our team in IBM Research. In the initial release of D5.3 [3], PyWren has been augmented with support for dataClay [9] from BSC, which is designated as the shared storage backbone for CLASS. In this final release, PyWren has been extensively optimized to reduce its computation latency, as explained further below. Also, PyWren has been integrated into the CLASS application of Trajectory Prediction, in the context of the Obstacle Detection use-case, as discussed in Section 3.3.1.1.
    b. **COMPSs:** a programming framework (programming model and runtime), developed at BSC, that allows to distribute parallel code (written in Java, C/C++ or Python programming languages) in a distributed and heterogeneous computing environment. A more detailed description of COMPSs and its features designed for this project can be found in Deliverable 2.5 [10].

c. **pCEP –** In this final release, the algorithms for processing events in the context of the CLASS applications, such as Trajectory Computation and Collision Detection, have been extracted by ATOS from pCEP and converted to run on PyWren. However, pCEP is still available as an analytics back-end in its own.
d. **DNN (Edge):** a suited and personalized version of YOLOv3, previously implemented in C by the darknet framework [11] then retrained and enhanced by use of TensorRT [12] exploiting tkDNN, a library developed by UNIMORE students. A more complete description of this DNN can be found in Deliverable 1.2 [13]

3 Technical Specification

This section provides a more detailed description of the specific components that are included in this release. Each component's section should include the technical details of the component along with a usage description and example, if such is available and relevant.

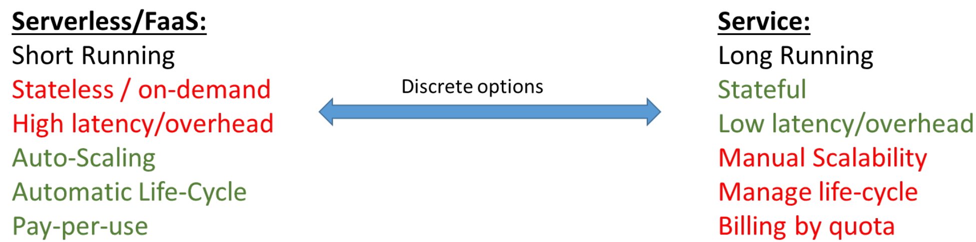
3.1 EXPRESS – EXtended PREdictability ServerlesS

3.1.1 Technical Description

3.1.1.1 Motivation for change

EXPRESS was originally (D5.3 [3]) based on Apache OpenWhisk with some adaptations for CLASS. However, since then, IBM has been considering revising its serverless offerings, moving away from OpenWhisk in favor of alternatives such as Knative [14], which reduced the value of our prototype. In addition, unlike the initial prototype, our team looked for a predictable solution that more significantly reduces the overhead of executing serverless function than just affecting scheduling. At the same time, the global serverless market is also experiencing a rising need for predictable execution, with the leader AWS introducing AWS Lambda Provisioned Concurrency [15], which improves predictable execution of its serverless Lambda offering, and other commercial offerings such as Iguazzio's Nuclio [16] real-time serverless is gaining traction.

Thus, our IBM Research team has conducted deeper research into the requirements of predictable computation on one hand, and into the general mechanisms underlying serverless function execution. A first significant observation is that in a modern commercial cloud environment, there are two discrete options for handling an event or a service request. One is a classic long-running service, and the other is via short running serverless function that is invoked in response to the request / event. Services, being long-running, are typically stateful. Specifically, they can prepare for a request and thus handle it with low latency. Serverless functions, on the other hand, are stateless and initialized on-demand. Thus, they naturally incur both high overhead of runtime initialization (commonly referred to as Cold Start [17]) and of application / state initialization, such as opening a database connection (can go up to 30 seconds [18]). Even after computation, a result cannot be returned before application-level cleanup (finalization) which can also be significant – and deferred in long-running services to the service shutdown phase. On a separate aspect, services often require additional effort of life-cycle and scaling, and are billed at coarse quotas (typical for IaaS implementations). In contrary, serverless functions enjoy automatic life-cycle management, auto-scaling, and pay-per-use. Our research question then became to create a **third option**, a controllable trade-off between services and serverless, which exhibits low-latency response similar to services, but is serverless in nature, enjoying automatic life-cycle and scaling, and is expected to cost more than net use but less than long-term quotas. The considerations are shown in Figure 2 below.



- Can we combine the *best* of both options?
  - Controllable trade-off
- Can the solution be *portable*?
  - Extend existing FaaS

Figure 2: Motivation for EXPRESS - Request Latency

3.1.1.2 EXPRESS Design

The result of the above research effort is a completely new incarnation of EXPRESS, as a portable framework on top of an existing FaaS / serverless platform. The basic approach in EXPRESS is re-modelling the serverless function, from a classic monolithic batch of code that is executed with parameters and returns a result, to a 3 sub-function sequence that is executed in the same memory space (and thus allows to maintain state), as shown in Figure 3(a) below.

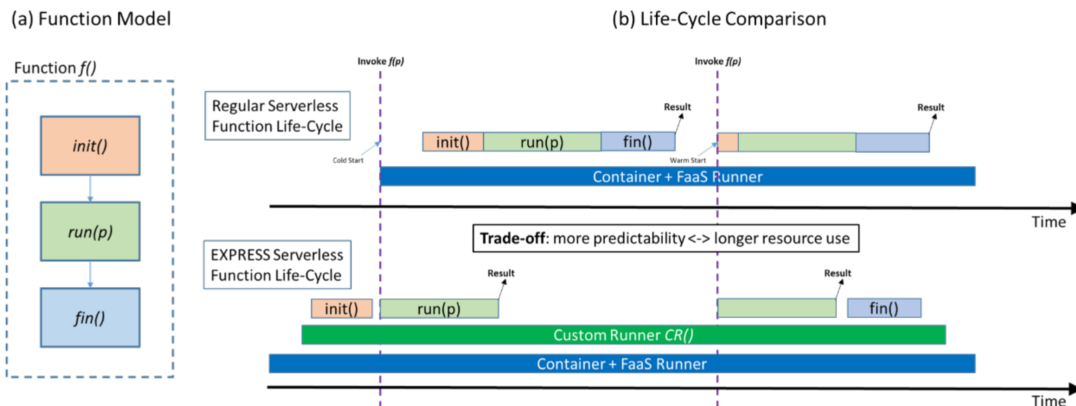


Figure 3: EXPRESS Design Concepts: (a) Function Model (b) Life-Cycle Comparison

The first sub-function is *init()* which encapsulates the initialization code, which is independent of the invocation parameters, such as opening a database connection, loading a machine-learning model for online classification, etc. Thus, *init()* can be executed when the function is pre-loaded, *ahead* of actual invocations, and its effect is preserved in-memory. Next sub-function is *run()*, which is the core computation and generates the result. *run()* is thus invoked when the entire function is invoked, using the invocation parameters. Last is *fin()* which encapsulates the finalization code - cleanup, log generation, closing connections, etc. fin() is independent of the main computation, and can thus be executed *after* an invocation result is returned. Furthermore, *fin()* may be *deferred*, so that the same memory space can be re-used to execute *run()* for multiple consequent invocations using the same initialized state. Figure 3(b) above shows the difference between classic serverless function execution, executing as a whole, and an EXPRESS function, whose separate sub-functions are executed at different

times, as explained above. The result is clear – much of the initialization and finalization execution overhead of serverless invocations is mitigated.

The key enabler for allowing EXPRESS functions to execute in this unique way is the *Custom Runner (CR)*. Any serverless function, in order to execute in a container, requires a special component called a *runner* to be included in the container as well. The runner provides both an execution runtime matching the function code (e.g., Python, JavaScript, or even specialized, such as PyWren). The runner is also remote-controlled by the scheduler of the serverless platform, telling it when to start executing a function invocation. The EXPRESS CR is similar to the FaaS runner in the sense that it provides a runtime matching the specific EXPRESS function structure. However, it is also capable of executing in itself as a classic serverless function, which means it can execute on top of a classic runner, as is shown in Figure 3 above. Upon execution, the EXPRESS custom runner connects to the EXPRESS *custom scheduler* (running a real-time scheduling algorithm), which tells it when to execute the specific sub-functions. This way, EXPRESS can create a *pool* of pre-initialized functions *ahead* of invocations, and execute only the *run()* sub-function in response to invocations. CRs are terminated (after executing *fin()*) at the serverless function timeout limit or when the EXPRESS scheduler decides to discard them. Each pool is auto-scaled and auto-managed by EXPRESS just like regular serverless functions. However, the actual serverless cost of using EXPRESS is determined by the duration of execution of a CR, which is clearly longer than the total sub-function execution, as shown in Figure 3 above. This is exactly the controlled trade-off of EXPRESS - paying more than net usage but less than for an IaaS-based service, to receive a service-level latency.

Integration with FaaS / serverless platform: as implied in Figure 3 above, EXPRESS functions are invoked through regular FaaS invocations, although they are not FaaS functions themselves, but are executed *inside* CRs, which are the actual FaaS functions. This is made possible through *wrapper functions*. Generally, speaking, to invoke an EXPRESS function *f()*, a wrapper function *ef()* is created. This function has the same signature as *f()* itself, but its implementation is calling the EXPRESS function pool API (see below) to invoke *f()*. *ef()* is deployed as a regular FaaS function and can thus relay invocations and events to *f()*. The function *ef()* is natively compiled and has a very small footprint, so it can be quickly scheduled by the FaaS platform and pose little overhead in the overall invocation.

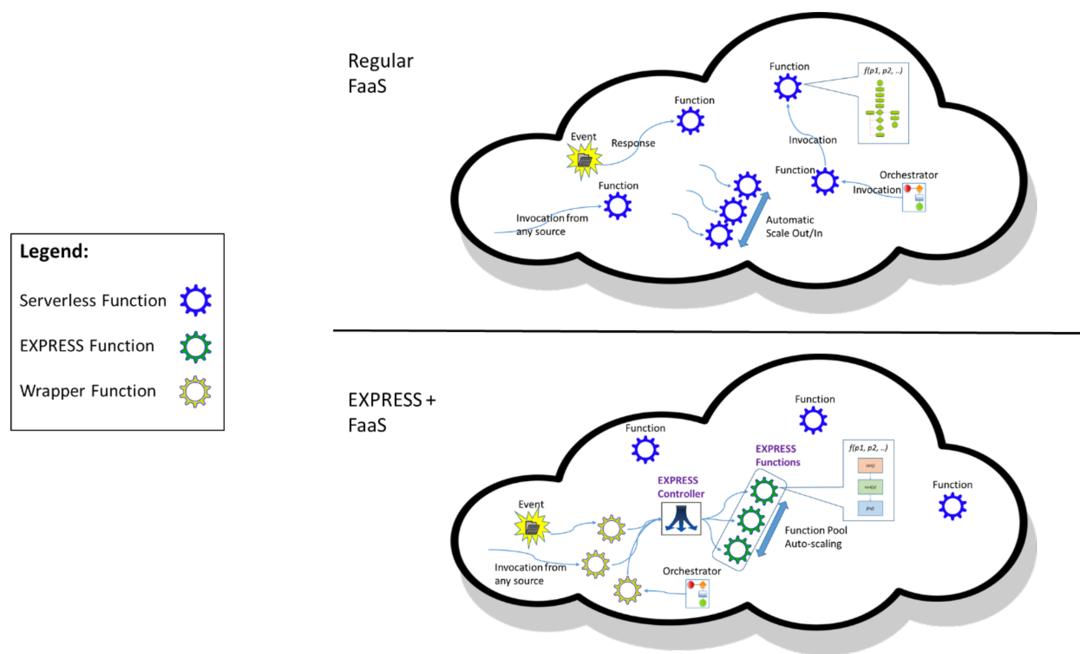


Figure 4: EXPRESS Architecture on top of FaaS Compared with just FaaS

Figure 4 above shows the overall architecture of EXPRESS and contrasts it with that of a regular FaaS / serverless platform. The top part shows a regular FaaS platform such as Apache OpenWhisk. In it, functions can be invoked either directly, in response to events, or from a workflow orchestrator. In the bottom part it shows EXPRESS running on top of a FaaS platform. Here, wrapper functions allow EXPRESS functions to be invoked just like regular FaaS functions, but enjoy custom real-time or demand-predicted scheduling (by the EXPRESS controller) and custom execution provided by EXPRESS CRs in the pool.

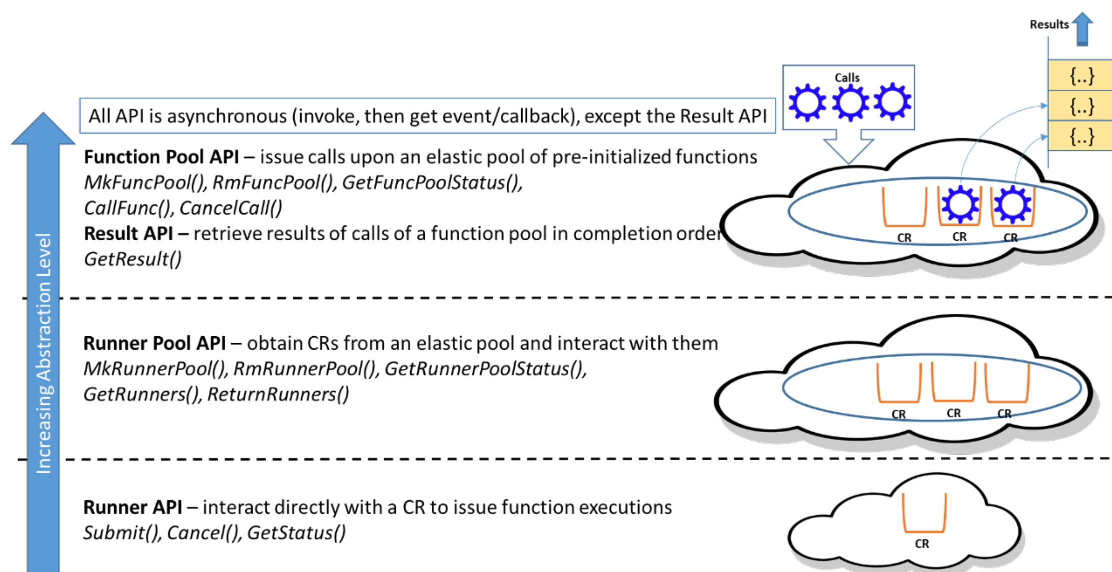


Figure 5: EXPRESS API

The API of EXPRESS is structured at 3 levels, with rising level of abstraction as shown in Figure 5 above. All API is asynchronous/non-blocking, except for result API. At the basic level of *runner API*, it allows to manually start a CR as a serverless function, and then send EXPRESS sub-functions to execute in it. At the next level of *runner pool API*, it allows to create an autonomous elastic pool of CRs and interact with each independently. Finally, at the highest

level of *Function pool API*, it allows to create an elastic pool of pre-initialized functions (with application function code) and perform direct invocations on that pool, taking advantage of predictable execution at its fullest. The API further allows to retrieve results of function invocations.

3.1.1.3 Related Works & Positioning

A common mitigation for cold start in FaaS platforms is that of *warm containers*. Generally speaking, a warm container is a container that finished executing a function. The container is kept alive after the function returns for a certain additional grace period. If another invocation for the same function arrives in that period, the container is reused by the FaaS platform to respond to the new invocation, without any cold start overhead. In some providers' platforms, such as AWS Lambda [19] or Azure Functions [20] , warm containers also share application objects across consequent function executions. Obviously, if the grace period expires, or if more concurrent invocations arrive than there are warm containers, the cold start issue returns, so warm containers alone is more of an opportunistic solution for predictable execution. Some platforms such as Apache OpenWhisk [4] feature *pre-warming* which creates a small number of extra containers without functions, in order to mitigate cold start from the first invocation.

A simple and popular technique of *periodical warming* [21] leverages warm containers proactively by invoking functions periodically to keep a designated number of containers warm indefinitely. However, this technique does not easily scale to many concurrent functions without respective adjustment of the refresh event processing time in the function code, which makes it even harder to scale dynamically.

In the academic and technical literature, there are plenty of solutions to improving predictability of serverless function execution. Nuclio [16] and Archipelago [22] couple real-time scheduling with explicit support for early initialization of both container and application. Cloudburst [23] provides efficient stateful serverless computation. SAND [24] improves function execution overheads by collocating functions based on scope – whether they belong to the same application or not. AWS Lambda Provisioned Concurrency [15] is a recent feature of AWS Lambda that allows early provisioning of a fixed-size pool of function containers and application initialization. Combined with AWS Auto-Scaling, the pool size can be dynamically adjusted to match load changes and adapt to bursts. Similarly, Knative [14] can be integrated with a custom scheduler that may allow for a dynamically-sized cold-start pool. PCPM [25] is an effort to improve container provisioning in FaaS platforms by modifying them to use pause containers.

Comparing EXPRESS with the above solutions shows that it is capable of supporting most of the key predictable serverless function execution features offered by them: mitigation of initialization and finalization, custom scheduling with demand prediction and/or real-time scheduling, dynamic pool scaling and even predictable state services (by properly extending its custom runners). However, EXPRESS embodies one other uncommon feature – it is *portable*, in the sense that it can be implemented on top of any basic FaaS system and maintain a similar development experience for both EXPRESS functions and for regular functions. Only two other reviewed solutions can be considered portable, namely warm containers and periodical warming, but they lack all other features except initialization mitigation. Adding predictable execution in a portable fashion can help a cloud consumer better control the cost of adding real-time support, by leveraging the existing FaaS investment rather than changing it, and arriving at a solution that is market-competitive in terms of features. It can also help a consumer and a provider to mitigate costs of maintaining or

switching a FaaS solution with added features, such as predictability, maintained independently.

3.1.1.4 An Intermittent Conclusion of EXPRESS

EXPRESS holds an enormous potential for innovation, well beyond serverless predictability. The powerful combination of custom runtime and custom scheduling is a clear indication of its ability to implement additional Non-Function Requirements for serverless platforms, and we expect to gain impact both inside IBM, as a strategic building block, and outside, in the open-source and academic communities. Also, we are pushing it to future EU project proposals. See Section 4.1.1 for details.

3.1.2 Usage

The instructions in this section explain how to install the current prototype of EXPRESS and launch a demo application of it, as following:

1. The current prototype of EXPRESS is available as a git repository[1]. It can be installed by simply cloning the git repository in your build machine (Linux) using the command below. The root folder of the cloned repository shall be referred to as EXPRESS_ROOT.
   ```
   git clone https://github.com/class-euproject/express.git
   ```
2. Install pre-requisites for building the prototype:
   - `Golang` version >=1.13
   - `Python 3`, version >= 3.6
3. As explained above, EXPRESS requires a FaaS / serverless platform to execute on top. For the current prototype, an OpenWhisk instance should be available with its command-line client (wsk) installed and configured – have a proper .wskprops file in the user's home folder. To install OpenWhisk, follow the instructions at https://github.com/apache/openwhisk
4. The prototype supports API at the basic runner level. There is a sample Python CR that can be built by following the instructions in `EXPRESS_ROOT/runners/python/README.md` .
5. EXPRESS relies on message queues for fast reliable communication between its components. In the current prototype, it uses RabbitMQ. The easiest way to start RabbitMQ is using Docker as explained in https://hub.docker.com/_/rabbitmq . Note that this requires installing Docker first, so it's easier to install on the same machine where OpenWhisk is installed as well (which requires Docker or K8s+Docker).
6. Now that RabbitMQ is running, it can be used via the URL: `amqp://guest:guest@<rabbitmq_host_ip>:5267` .
   Edit `EXPRESS_ROOT/test/happy0/happy0.go` line 70 and paste this URL with the right RabbitMQ host IP address.
7. Open a terminal at `EXPRESS_ROOT/test/happy0` and run
   ```
   go mod tidy
   go build
   ```
   to build the demo executable of happy0. If the installation process has been properly followed, no errors should be generated.
8. Run the demo in the same folder of the build above: `./happy0`. The demo starts a CR as a function in the configured OpenWhisk instance, sends an execution request to it, gets a result and finally probes for status.

[1] Currently, EXPRESS is hosted in a private git repository until the IBM administrative process of open-source release completes. Access will be granted to reviewers in a per-case basis.

3.2 Runtime support for COMPSs

This deliverable has been provided in full at MS2 and all relevant details are in D5.3 [3].

3.3 Optimized PyWren with dataClay Support

3.3.1 Technical Description

By the previous milestone MS2 we adapted PyWren (see D5.3 [3]) to use dataClay [9] object oriented API and models. dataClay is the CLASS backbone for data propagation across cloud and edge. In this milestone of MS3 we focused on two aspects. The first is integrating PyWren in designated CLASS applications of Trajectory Prediction (completed - see demonstration below) and Collision Detection (near completion – see demonstration of stand-alone code), as part of the Obstacle Detection use-case. The second is optimizing PyWren latency to make it more suitable for low-latency computation, especially in the context of CLASS use-cases.

3.3.1.1 CLASS Application Integration

There are two designated applications involving PyWren in the CLASS use-case: Trajectory Prediction (TP) and Collision Detection (CD). TP involves computation of an updated trajectory prediction for each object detected in a new snapshot of scanning a given traffic arena covered by cameras. CD involves first filtering which objects are included in the Warning Area (WA) of each car that requests alerts, and then computing possible collision alerts between the car and these objects based on their predicted trajectories. Both applications are described in more detail in D1.4 [26].

Both core applications have been originally developed by ATOS as pCEP applications. Then they were converted to Python (to be used with PyWren) and tested using file input. As part of the integration effort, a common data input layer called "data managers" has been provided by IBM to allow smooth transition from file input to dataClay input. Both delivered applications use this layer. This layer essentially provides two different implementations (file-based and dataClay-based) to a common API of retrieving and storing data that is required by each application's logic. For example, a common API is `getVehicleByID()`, which retrieves a full record of an object in the DKB (the term "vehicle" is a chronological misnomer).

Trajectory Prediction PyWren integration: involves taking the core logic function of `traj_pred_v2(v)` from the ported Python application. This function takes an object id and computes its trajectory prediction based on its updated location history. The PyWren TP application is triggered by an event whose payload points to the dataClay alias containing the updated snapshot. This snapshot contains the object ids of street objects (cars, pedestrians and bicycles). The main PyWren application loads the object ids and then runs a PyWren `map()` operation which processes all objects concurrently using OpenWhisk actions (serverless functions). Each action executes the `traj_pred_v2(v)` function on the respective object. A demonstration of the integrated PyWren application is provided and described in Section 4.2.2.

Collision Detection PyWren integration: involves filtering objects belonging in the warning area of a given car, and then detecting possible collisions between the filtered objects and the given car, which yields alert warnings. The core logic function `collision_detection(main_obj,other_obj)` in the ported application checks for a possible collision between two street objects from the DKB and generates an alert object if collision is detected. The PyWren application is triggered by an event pointing to a given DKB object (assumed to be a car). Then PyWren invokes a `map()` operation that concurrently filters all

objects in the same arena based on a distance filter to determine if they fall within the car's warning area. Each object that passes the filter is then checked for collision with the car using the function `collision_detection(main_obj,other_obj)`. This integration is not completed yet, so we provide a stand-alone simulation of running the function `collision_detection(main_obj,other_obj)` between all objects in a given set, and using the data managers. This simulation is provided and described in Section 4.2.3.

3.3.1.2 PyWren Optimizations

PyWren, being a means for large-scale batch map-reduce computation, was not originally designed for low-latency computation. However, its ability to pool concurrent resources in an efficient map/reduce computation is a very good fit for CLASS requirements in the specific applications of TP and CD. It also fits the data generation pattern in CLASS, since data is generated in snapshots, each applying to multiple objects. Our objective in MS3, therefore, is to tune PyWren, both on its own and in combination with the CLASS applications and with dataClay, to reduce the overall end-to-end latency of snapshot computation to a value that is usable by the application.

The PyWren-specific optimizations have been:

1. Tuning specific time-outs in PyWren of polling for result availability: PyWren is typically geared for long computation in its serverless workers. The main client code dispatches work to the workers and then polls the object storage for results. We increased polling frequency in various code locations to accommodate the much shorter worker computation.
2. Skipping PyWren dynamic runtime installation in workers – by creating custom docker images for workers with runtime pre-installed
3. Moving the distribution of the the PyWren map function (which contains the TP or CD logic, respectively) from the dynamic process using object storage to a static process using custom docker images
4. Switching the PyWren workers from spawning a process to do the computation to spawning a thread, in the container runtime. This keeps settings in the same memory space and allows leveraging warm containers – running multiple consequent invocations in the same memory space, which allows skipping initialization. Specifically, this helped with accelerating dataClay usage in warm container – since the session is already set up, it can be skipped, removing overhead of entire seconds.

These optimization are combined with a new, more efficient data model in dataClay by BSC described D1.4 [26].

Our current efforts result in the PyWren-TP application reducing from an original ~12 seconds computation (for a single object) to ~0.6 seconds – x20 decrease. This value seems quite usable, since we deal with the *warning area* of a car, whose objects are reasonably assumed to be 10-50 meters away from a car, which is traveling at the speed limit for urban area of 50 Km/H. At this speed, the car would take 0.7-3.6 seconds to arrive at an object that is stationary or at least not moving in the opposite direction. This covers many possible scenarios including those selected for demonstration in D1.4 [26].

3.3.2 Usage

Usage of PyWren with dataClay has already been provided in detail in D5.3 [3]. The same description still holds since the changes for MS3 do not affect the external aspect of PyWren. See Section 4.2 for installation details of both PyWren and TP and CD integrations.

3.4 COMPSs

COMPSs provides a task-based programming model, as well as map/reduce operations. Details of COMPSs as an analytics backend can be found in D5.3 [3] and D2.5 [10].

3.5 pCEP (ATOS)

Details of the pCEP component can be found in deliverables D5.3 [3], D2.1 [27], D5.1 [2], and D1.2 [28].

3.6 DNN at the Edge

The Deep Neural Network (DNN) analytics backend has been described in D5.3 [3] and D1.4 [26].

4 Delivery

This section contains all the specifics of the delivery, including: bundling, installation instructions, and related demonstrations and impact, in accordance with the DoA and evolution of work. As before, there is a sub-section per component.

4.1 Bundling and Installation

4.1.1 EXPRESS – Extended PREdictability ServerlesS

Installation for EXPRESS and OpenWhisk is explained together with its usage in Section 3.1.2

4.1.2 Runtime support for COMPSs

Installation details are available in the respective Delivery Section of D5.3 [3]

4.1.3 Optimized PyWren with dataClay Support

The code is available at this new git branch: https://github.com/class-euproject/pywren-ibm-cloud/tree/pywren-class

Clone the above repository and follow the installation instructions at the same link.

4.1.4 Trajectory Prediction Application

The code and installation instructions are available at this git repository: https://github.com/class-euproject/trajectory-prediction

4.1.5 Collision Detection Application

The code and installation instructions are available at this git repository: https://github.com/class-euproject/collision-detection

4.1.6 COMPSs

Instructions on how to download, install and use COMPSs are available in D2.6 [29].

4.1.7 pCEP (ATOS)

Installation details are available in the respective Delivery Section of D5.3 [3]

4.1.8 DNN at the Edge

Installation details are available in the respective Delivery Section of D1.4 [26] and online at https://github.com/class-euproject/class-edge

4.2 Demonstration and Impact

The artefacts delivered in MS3 have resulted in a considerable impact. In addition, this report includes two bundled demonstrations in the form of on-line videos available at the CLASS intranet:

https://class-project.eu/user/login

A dedicated user has been created for demonstration purposes, with limited access to deliverables and related videos. The credentials to access this service are the following:

Username: *EC_user*

Password: *@Hz.52qXXF#K23*

After log in, click on “Intranet”, the demonstration videos and files of this deliverable are located in “**PU_D5-4Demo**” directory

4.2.1 EXPRESS

EXPRESS is the key innovation our IBM team builds on for a future serverless platform. The impact in its wake includes:

- IP (patent disclosure) submitted to the IBM patent board, covering the architecture and operational concepts of EXPRESS. It has already passed an internal evaluation board for novelty and business value, and is now undergoing an extensive prior art search by a professional team.
- Two future EU project proposals – CONGENIAL and UBIQUITY – include EXPRESS as a foundation. The proposals are for the future cloud call (ICT-40), are from different consortiums (except IBM), and are focusing on predictability and additional non-functional capabilities delivered via EXPRESS.
- EXPRESS is promoted in an IBM-internal strategic discussion on the future of cloud and serverless
- EXPRESS has already been briefly presented to an IBM stakeholder of serverless offering.

We have avoided publicizing EXPRESS in papers/blogs until the patent process reaches filing, which we hope should happen in the next few months.

4.2.2 Trajectory Prediction

We provide a video demonstration of the PyWren-based implementation of the Trajectory Prediction (TP) application, which is part of the overall workflow of the Obstacle Detection use-case in CLASS. The application operates as explained in Section 3.3.1.1.

In the video we see first the application action *tp-action* and the event trigger that is bound to it, *tp-trigger*. Next, we launch a small demo application that polls the default snapshot alias in dataClay and prints the trajectory prediction values for all objects. Then, the event is fired without parameters, so it points to the default alias. The action is invoked and quickly computes TP values for all objects, which are then shown in the demo application output.

4.2.3 Collision Detection

We provide a video demonstration of a stand-alone implementation of the Collision Detection (CD) application, which is part of the overall workflow of the Obstacle Detection use-case in CLASS. The application operates as explained in Section 3.3.1.1.

The video shows a demonstration of the CD application using an infinite loop (in our case, every 10 seconds) to continuously detect possible collisions and trigger an alert, for now as a print. In each iteration, all the objects stored in dataClay, and their predicted trajectories, are taken into consideration using an alias and the specific function from the dataClay API. Different types of times (global execution, time per collision detection, and more) are analyzed to get an initial estimation of the CD component efficiency when working with different number of objects and communications with dataClay. The output of this demonstration shows the different alerts that should arise to specify the possible collisions between objects with the coordinates and the timestamp associated, and the execution times of different parts of the application.

5 Product Glossary

This auxiliary section provides a brief overview of the products involved in CLASS analytics, to put the above discussion in context.

5.1 Apache OpenWhisk

Apache OpenWhisk [4] (OW for short) is a serverless, open source cloud platform, which was initiated, and is still maintained, by IBM. OpenWhisk executes functions (called *actions*) in response to events, at scale. Both actions and events are high-level abstractions that can be implemented in various ways. Actions, as code, can be written in virtually any programming language (although there are 7+ languages that have official support), and using many platforms and SDKs. Similarly, events can represent any concrete event or signal, such as message arrival, command invocation, device signals, or mark the occurrence of a higher logic result, such as complex events or other decision logic. Once defined, events can be bound to actions using *rules* to create event-driven applications, with simple facilities for relaying event data to invoked actions. Such applications are cloud-native, in the sense that events can arrive and be processed by actions anywhere in the cloud, and actions are elastically auto-scaled to match the event load.

The resulting programming model of OpenWhisk offers several attractive advantages to developers, in addition to polyglot programming and auto-scaling. Developers do not need to manage the location of their code (hence the term “serverless”), its life-cycle or its resource allocation – OpenWhisk uses a default (but customizable) resource allocation for each action. Actions are time-limited to keep consistent with the original serverless model from AWS, but time-limit is configurable.

The architecture of OpenWhisk consists of the components shown in Figure 6.

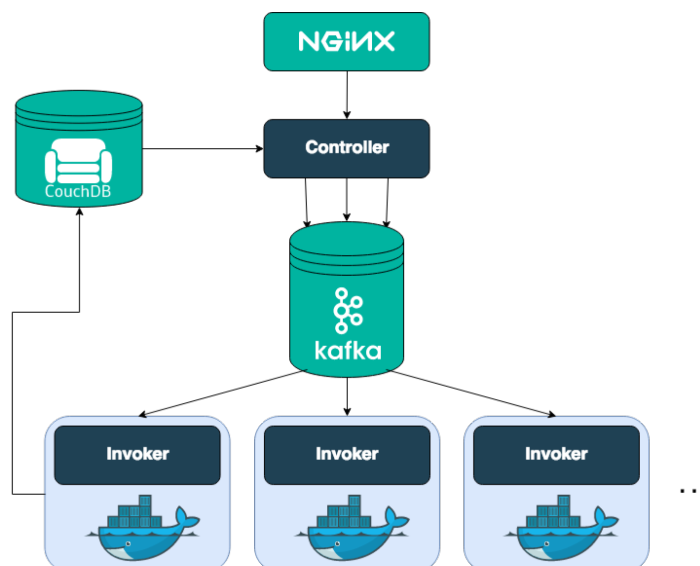


Figure 6: OpenWhisk Architecture.

The components of the OW architecture are introduced as follows:

- **NGINX** is an optional reverse proxy, used for load-balancing controllers and for SSL termination.
- **CouchDB** is a database used for storing the assets created by users – actions, triggers, rules, packages and records of action activations.
- **Controller** is a management logic of OW. It implements the OW REST API, and dispatches actions for execution at invokers in response to events.
- **Kafka** is a message bus used to distribute messages from controllers to invokers in a cloud setting.
- **Invoker** is a "worker" of OW. It executes actions using IaaS or cluster facilities. By default, an invoker uses Docker containers for running actions, but there are variations that use Kubernetes and other facilities.

OW has a simple interface consisting of a REST API and a CLI (`wsk` command) which wraps the REST API. It allows creating actions and invoking them, creating event triggers from event feeds of actual events, binding event triggers to actions via rules, and several secondary operations. OW programming model is documented in detail in [30].

5.2 PyWren

PyWren [31] is a system that was built at UC Berkeley's RISELAB to enable highly scalable execution of existing Python functions on the cloud using the serverless platform. It started on AWS Lambda, the serverless platform of AWS Cloud, and later it was converted [8] to use IBM's Cloud Functions, based on Apache OpenWhisk.

PyWren's programming interface is based on Map/Reduce. The developer writes a *client* program that during runtime, creates an *executor*, which issues highly parallel computations as `map` and `reduce` operations in its API. These operations are executed using a set of concurrent serverless functions/actions. Each action's environment is prepared to include the dependencies needed to execute the map or reduce function, including dependencies of the function code and of PyWren itself. The functions used in `map` and `reduce`, as well as the input and output datasets, are shared between the client and the actions via object storage.

The current basic API for *map* and *reduce* in PyWren is as following [8]:

- `executor.map(func, dataset)`
- `executor.map_reduce(map_func, dataset, reduce_func)`

Both operations perform map first, which applies `func` (`map_func` in `reduce`) to all elements of the dataset. For `reduce`, it later applies a reduce function to the dataset resulting from map. The `reduce` function has an internal accumulator carrying results from one computation to the next, ending with a single final result. Figure 7 demonstrates the operation of a `map` computation in PyWren, involving the client, IBM Cloud Functions (OpenWhisk) and object storage, in an IBM Cloud setup.

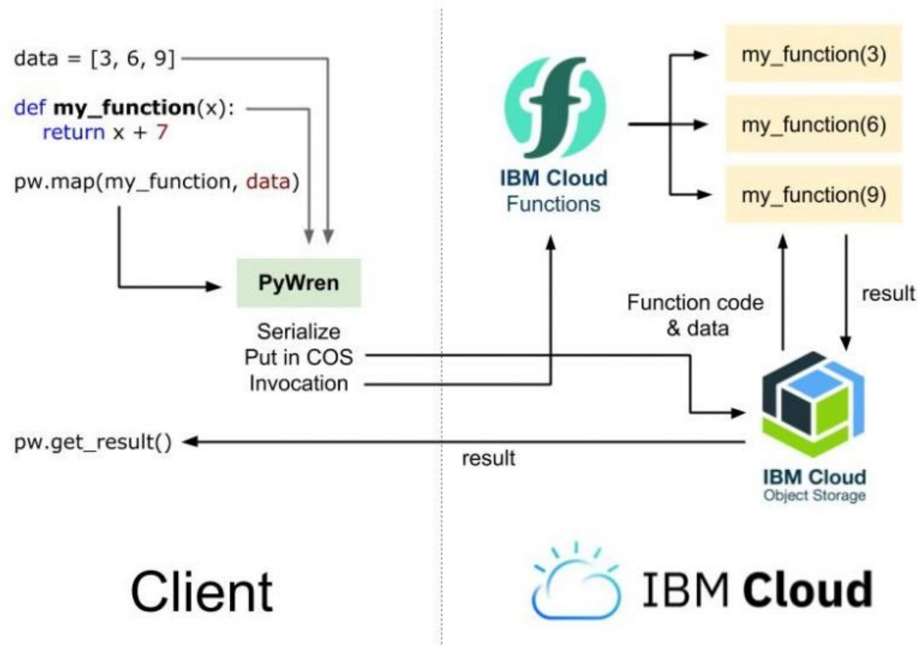


Figure 7: Execution of a PyWren map operation.

5.3 COMPSs

Deliverable D2.1 [27] provides a detailed description of COMPSs.

6 References

[1] CLASS Consortium, "CLASS: Edge&CLoud Computation: A Highly Distributed Software Architecture for Big Data AnalyticS," Baercelona, 2017.

[2] E. Hadad, *D5.1 Analytics Requirements,* 2018.

[3] E. Hadad, "D5.3 Initial Release of CLASS Big-Data Analytics Layer," CLASS, 2019.

[4] Apache OpenWhisk, "Apache OpenWhisk is a serverless, open source cloud platform," Apache Foundation, [Online]. Available: http://openwhisk.apache.org/. [Accessed 21 June 2018].

[5] Linux Foundation, "Production-Grade Container Orchestration - Kubernetes," [Online]. Available: https://kubernetes.io. [Accessed 3 2019].

[6] Docker Inc, "Enterprise Application Container Platform - Docker," [Online]. Available: https://www.docker.com/. [Accessed 3 2019].

[7] Barcelona Supercomputing Center, "COMP Superscalar - BSC-CNS," [Online]. Available: https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar/. [Accessed 3 2019].

[8] G. Vernik and J. Sampe, "Process large data sets at massive scale with PyWren over IBM Cloud Functions - IBM Cloud Blog," IBM, [Online]. Available: https://www.ibm.com/blogs/bluemix/2018/04/process-large-data-sets-massive-scale-pywren-ibm-cloud-functions/. [Accessed 21 June 2018].

[9] Barcelona Supercomputing Center, "dataClay - BSC-CNS," [Online]. Available: https://www.bsc.es/research-and-development/software-and-apps/software-list/dataclay. [Accessed 3 2019].

[10] N. Mammadli and J. Álvarez , "D2.5 – Final release of Spark and COMPSs integrated in CLASS architecture," CLASS, 2020.

[11] J. Redmon, "YOLO: Real-Time Object Detection," [Online]. Available: https://pjreddie.com/darknet/yolo/.

[12] "NVIDIA TensorRT | NVIDIA Developer," [Online]. Available: https://developer.nvidia.com/tensorrt.

[13] M. Corredoira, "D1.2 Final Release Of The Smart City Use Cases," 2019.

[14] Knative Community, "Knative," [Online]. Available: https://knative.dev/. [Accessed March 2020].

[15] J. Beswick, "New for AWS Lambda – Predictable start-up times with Provisioned Concurrency," 4 December 2019. [Online]. Available: https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/. [Accessed 2 2020].

[16] iguazio, inc, "Nuclio: Serverless Platform for Automated Data Science," [Online]. Available: https://nuclio.io/. [Accessed 2 2020].

[17] M. Shilkov, "Cold Starts in AWS Lambda," 26 September 2019. [Online]. Available: https://mikhail.io/serverless/coldstarts/aws/.

[18] H2 Database, "H2 Performance Comparison," [Online]. Available: http://www.h2database.com/html/performance.html. [Accessed 2 2020].

[19] Amazon Web Services, "AWS Lambda Execution Context," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/runtimes-context.html. [Accessed 2 2020].

[20] M. Groves, "Lazy Initializing Within Azure Functions," 15 September 2017. [Online]. Available: https://blog.couchbase.com/azure-functions-lazy-initialization-couchbase-server/.

[21] J. Daly, "Lambda Warmer: Optimize AWS Lambda Function Cold Starts," 14 July 2018. [Online]. Available: https://www.jeremydaly.com/lambda-warmer-optimize-aws-lambda-function-cold-starts/.

[22] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman and A. Akella, "Archipelago: A Scalable Low-Latency Serverless Platform," 22 November 2019. [Online]. Available: https://arxiv.org/pdf/1911.09849.pdf.

[23] C. W. X. C. L. J. S.-S. J. M. F. J. E. G. J. M. H. A. T. Vikram Sreekanti, "Cloudburst: Stateful Functions-as-a-Service," 27 January 2020. [Online]. Available: https://arxiv.org/pdf/2001.04592.pdf.

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

[25] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak and V. Sukhomlinov, "Agile Cold Starts for Scalable Serverless," in *HotCloud'19*, Renton, WA, 2019.

[26] D. Amendola, "D1.4 - Final Release Of The Smart City Use Case," CLASS, 2020.

[27] E. Quinones, "D2.1 – CLASS software architecture requirements and integration plan," CLASS, 2018.

[28] M. Corredoira, *D1.2 First release of the smart city use cases,* 2019.

[29] CLASS, "D2.6 - Second release of the CLASS software architecture," July 2020.

[30] Apache OpenWhisk, "incubator-openwhisk/docs at master · apache/incubator-openwhisk · GitHub," [Online]. Available: https://github.com/apache/incubator-openwhisk/tree/master/docs#readme. [Accessed 21 June 2018].

[31] "pywren -- run your python code on thousands of cores - pywren," RiseLab, UC Berkeley, [Online]. Available: http://pywren.io. [Accessed 21 June 2018].

[32] Barcelona Supercomputing Center, "Application development guide. COMPSs user manual," 9 November 2018. [Online]. Available: http://compss.bsc.es/releases/compss/latest/docs/COMPSs_User_Manual_App_Development.pdf.

[33] Barcelona Supercomputing Center, "Application execution guide. COMPSs User Manual," 9 November 2018. [Online]. Available: http://compss.bsc.es/releases/compss/latest/docs/COMPSs_User_Manual_App_Exec.pdf.

[34] E. Quinones, "D2.4 First Release Of The CLASS Software Architecture," 2019.

[35] D. Breitgand, "Lean OpenWhisk: Open Source FaaS for Edge Computing," 7 2018. [Online]. Available: https://medium.com/openwhisk/lean-openwhisk-open-source-faas-for-edge-computing-fb823c6bbb9b. [Accessed 3 2019].

[36] IBM, "OpenWhisk REST API," [Online]. Available: https://openwhisk.ng.bluemix.net/api/v1/docs/index.html?url=/api/v1/api-docs#/. [Accessed 3 2019].

[37] OpenWhisk Community, "[OpenWhisk] Documenation," [Online]. Available: https://openwhisk.apache.org/documentation.html. [Accessed 3 2019].

[38] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes and J. Labarta, "PyCOMPSs: Parallel computational workflows in Python," *IJHPCA,* vol. 31, no. 1, pp. 66-82, 2017.

[39] OpenWhisk Community, "Creating triggers and rules," [Online]. Available: https://github.com/apache/incubator-openwhisk/blob/master/docs/triggers_rules.md. [Accessed 3 2019].

[40] Wikipedia, "Network Time Protocol," [Online]. Available: https://en.wikipedia.org/wiki/Network_Time_Protocol. [Accessed 3 2019].

[41] R. Cavicchioli, "D3.4 First release of the real-time analysis methods and tools on the edge," 2019.

[42] OpenWhisk Community, "Setting up OpenWhisk on Ubuntu server(s)," [Online]. Available: https://github.com/apache/incubator-openwhisk/blob/master/tools/ubuntu-setup/README.md. [Accessed 3 2019].

[43] MASERATI, BSC, "D1.2 - Final release of the smart city use cases".

[44] ATOS, "D4.1. - Cloud requirement specification and definition," 2018.

[45] OpenWhisk Community, "Apache OpenWhisk package that can be used to create periodic, time-based alarms," [Online]. Available: https://github.com/apache/incubator-openwhisk-package-alarms. [Accessed 3 2019].

[46] E. Hadad, "erezh16/express: A serverless platform with predictability enhancements - based on OpenWhisk," IBM Research, [Online]. Available: https://github.com/erezh16/express. [Accessed 3 2019].

[47] MinIO, Inc., "MinIO - Object Storage for AI," [Online]. Available: https://min.io/. [Accessed 3 2019].

[48] IBM Corp., "IBM Cloud Object Storage - Overview," [Online]. Available: https://www.ibm.com/cloud/object-storage. [Accessed 3 2019].