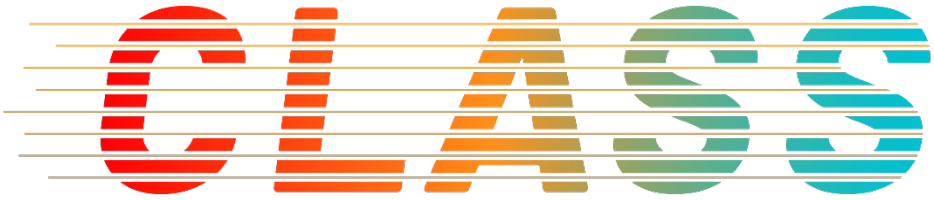

D2.7 – Final release of the CLASS Software Architecture

Version 1.0

Document Information

Contract Number	780622
Project Website	https://class-project.eu/
Contractual Deadline	M42, 30th June 2021
Dissemination Level	CO
Nature	Demonstrator
Author(s)	Elli Kartsakli (BSC)
Contributor(s)	BSC, IBM, ATOS
Reviewer(s)	ATOS
Keywords	software architecture, CLASS scheduling strategy, edge cloud coordination

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".*

© 2021 CLASS Consortium Partners. All rights reserved.

Change Log

Version	Author	Description of Change
0.1	Elli Kartsakli (BSC)	Initial Draft
0.2	Elli Kartsakli (BSC)	Sections 2 and 3
0.3	Erez Hadad (IBM), Rut Palmero (ATOS)	Added deployment steps for Lithops and Rotterdam
0.4	Rut Palmero (ATOS)	Internal reviewer.
1.0	Elli Kartsakli (BSC)	Final version, ready to EC review.

Table of contents

Executive Summary

This deliverable presents the final release of the CLASS software architecture, elaborated during the last phase of the project "Validation". The deliverable is organized in three main parts.

First, an overview of the CLASS software architecture is given, briefly describing all the software components and focusing on any updates with respect to the second release of the architecture, reported in D2.6 [1]. The final release of the CLASS software architecture has been used for the evaluation and validation of the CLASS use cases, as reported in D1.6 [2], as well as the other evaluation-related deliverables of WP2-WP5.

Second, a description of the analytics workflow for the execution of the two CLASS use cases, namely collision detection and air pollution estimation, is given, as well as an example of the workflow execution and the expected outputs. This example showcases the capabilities of the software architecture to facilitate the development and distribution of the workflow across the compute continuum.

Finally, a detailed guide for the deployment of the full stack of the CLASS software architecture is given, along with links to the repositories where the relevant open-source code can be located.

Overall, this deliverable marks the successful completion of Task 2.3 and the achievement of all objectives defined for milestone MS4 of the project.

1 Introduction

This deliverable presents the final release of the CLASS software architecture, elaborated during the last phase of the project ("Validation"). The deliverable is organized in three main parts.

First, Section 2 provides the overall picture of the final release of the CLASS Software Architecture. The proposed architecture is composed of four main components, which are reported and evaluated in isolation in dedicated deliverables of the corresponding Work Packages (WPs): the data analytics platform (reported in D5.5 [3]) the cloud analytics platform (D4.6 [4]), the edge analytics platform (D3.6 [5]) and the computation distribution layer (in D2.8 [6]). Furthermore, the CLASS software architecture has been deployed in the Modena Automotive Smart Area (MASA) and employed for the final validation of the project use cases, as reported in D1.6 [2].

In continuation, Section 3 provides an overview of the data analytics methods developed within CLASS for the two project use cases, namely collision detection and air pollution estimation, and how they are executed over the CLASS Software Architecture (SA). Both the COMPSs task-based methods executed at the edge, as well as the analytics running in the cloud as Lithops serverless functions are briefly described. Two examples are given for the execution of the end-to-end workflow. In the first one, a recorded video from a staged collision using the CLASS vehicles has been used as an input. However, as this video cannot be made public due to privacy/GDPR issues, a second reproducible example has been provided, using a publicly available video of a collision.

Section 4 provides a detailed guide of the steps for the deployment of the CLASS SA. This section is organized in four parts. The first two subsections refer to the deployment for the execution of the two CLASS applications using the offline video. The other two subsections refer to the deployment of two additional features of the CLASS SA, namely, i) the Rotterdam service, which can be used jointly with COMPSs to scale the deployment of containerized COMPSs workers at the cloud, ii) the EXPRESS prototype, which has been delivered as a stand-alone feature in D5.4 [7]. Whenever possible, links to the corresponding deliverables are given, to avoid duplication of the information.

2 The final release of the CLASS Software Architecture

This section will provide a brief overview of the CLASS SA. The first outline of the CLASS SA has been presented in D2.1 [8], and has been refined as each project milestone has been reached, as reported in D2.4 [9] for milestone MS2 and D2.6 [1] for milestone MS3.

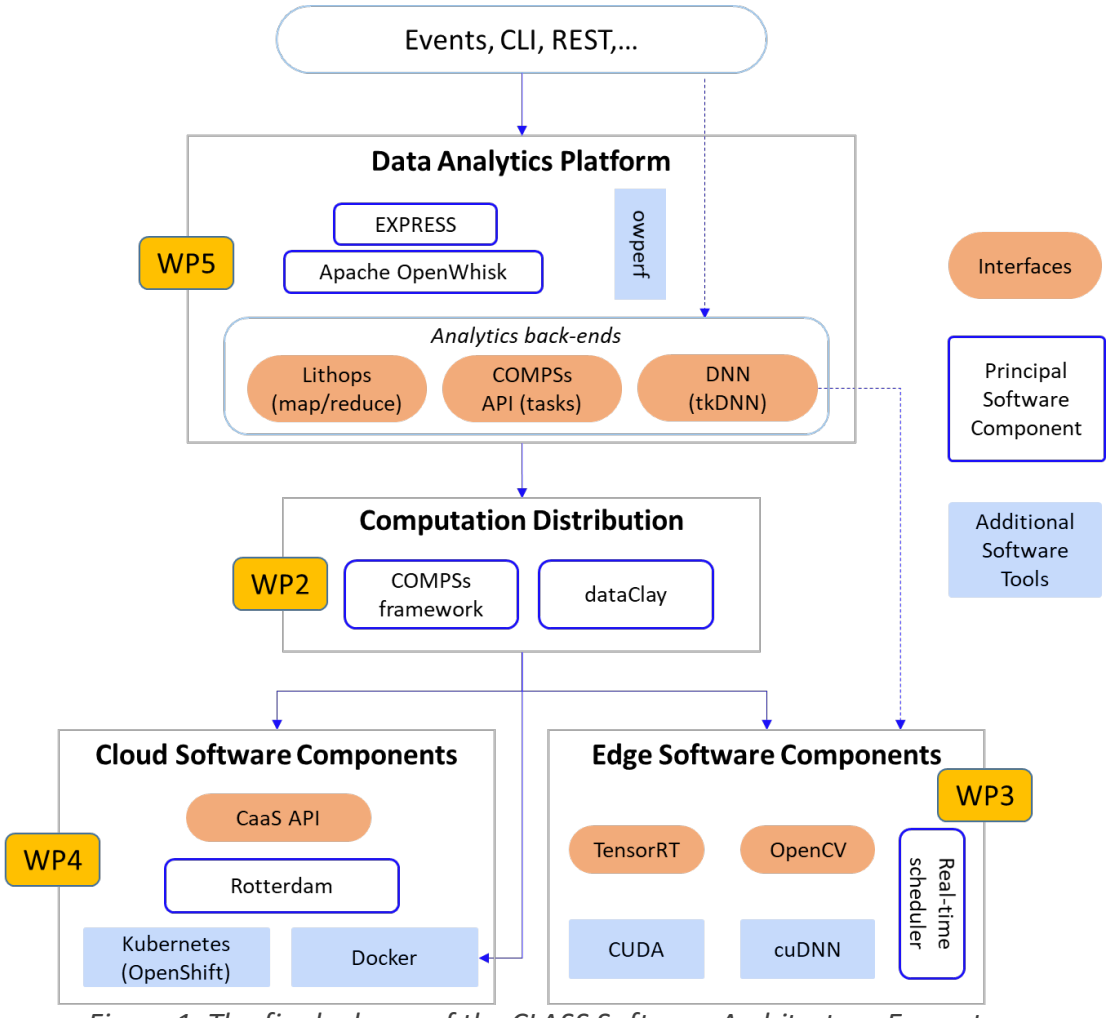

Figure 1. The final release of the CLASS Software Architecture Ecosystem

Figure 1 depicts the components and interfaces that form the final release of the CLASS software development ecosystem, which has been evolving throughout the lifetime of the project. Solid lines represent the interconnections of all the software components, whereas dashed lines represent other possible interactions between software components, not including all the envisioned CLASS functionalities. Each component is also marked with the number of the corresponding Work Package (WP).

The CLASS ecosystem consists of a data analytics platform/layer (WP5) upon which the data analytics methods are executed (WP1). The algorithms can be implemented and parallelized with the big-data analytics programming models provided by the data analytics layer. This layer also exposes the interface of OpenWhisk (WP3) and COMPSs

(WP2), responsible of efficiently distributing the computation across the compute continuum by coordinating and exploiting edge and cloud performance capabilities. These two runtime systems (OpenWhisk and COMPSs) further exploit the parallel programming models available in the edge (WP3), and the containerized tasks for cloud execution by means of the Rotterdam Container as a Service (CaaS) Application Programming Interface (API) (WP4).

The components of the CLASS software architecture are also summarized in Table 1, where the name of the components, their owner and license and the most relevant deliverables with their development are listed.

Table 1. CLASS Software Components

Software Component		Owner	License	Deliverable
Data Analytics Platform	Openwhisk / EXPRESS	IBM	open-source	D5.3, D5.4, D5.5
Data analytics back-ends	Lithops	IBM	open-source	D5.3, D5.4, D5.5
	COMPSs programming model	BSC	open-source	D2.4
	DNN/tkDNN	UNIMORE	open-source	D1.4, D3.3, D5.3
Computation Distribution	COMPSs	BSC	open-source	D2.4, D2.6
	dataClay	BSC	open-source	D2.4, D2.6
Cloud components	Rotterdam	ATOS	open-source	D4.1, D4.2, D4.4, D4.7, D4.6
	SLALite	ATOS	open-source	D4.1, D4.2, D4.4, D4.7, D4.6
	SLA Predictor	ATOS	open-source	D4.6
	Kubernetes	Canonical	open-source	D4.1, D4.2, D4.4, D4.7, D4.6
	Kubeless	Bitnami	open-source	D4.7
	Openshift	Red Hat	open-source	D4.1, D4.2, D4.4, D4.7, D4.6
	Docker	Docker Inc.	open-source	D4.1, D4.2, D4.4, D4.7, D4.6
Edge components	CUDA	NVIDIA	proprietary	D3.1, D3.3, D3.6
	cuDNN	NVIDIA	proprietary	D3.1, D3.3, D3.6
	Micro K8s	Canonical	open-source	D4.7
	Real-time Scheduler	UNIMORE	open-source	D3.4, D3.5

In continuation, a brief detail of the key components and their role in CLASS is given.

2.1 Data analytics platform

The CLASS data analytics platform provides the necessary interfaces for the application programmers to develop complex big-data analytics workflows, for real-time execution across the compute continuum. The platform allows multiple types of big-data analytics back-ends, such as map/reduce, task-based, or Deep Neural Network (DNN), to integrate in a uniform mesh where workloads and components can interact or be invoked via REST, Command Line Interface (CLI) or in response to events. The key to this novel approach is the use of a serverless platform based on ***Apache OpenWhisk*** [10].

OpenWhisk is a serverless, open source cloud platform (originally from IBM) that executes functions (called *actions*) in response to events at any scale. Both actions and events are high-level abstractions that can be implemented in various ways. Actions can be written in virtually any programming language and using many platforms and Software Development Kits (SDKs). Similarly, events can be implemented by any concrete event or signal, such as message arrival, command invocation, device signals, etc. Once defined, events can be bound to actions using *rules,* to create event-driven applications. 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, thus relieving the developers from these concerns. The ***Owperf*** tool has been used for the performance evaluation of the analytics workflows of the OpenWhisk-based deployment.

The data analytics platform can be adapted to support multiple analytics back-ends. Support for the following data analytics back-ends has been provided in CLASS:

- ***The Lithops framework*** (formerly known as PyWren). Lithops is an open-source lightweight implementation of *Map/Reduce programming model* over the Apache Openwhisk serverless platform, aiming to massively scale Python applications and fully support concurrent execution. In CLASS, Lithops has been used to accelerate the computation of the trajectory prediction and collision detection analytics methods (for more details, see D5.4 [7] and D5.5 [3]).
- ***The COMP Superscalar (COMPSs) programming model***. The COMPSs *task-based programming model* enables programmers to develop distributed applications following the sequential programming paradigm and using standard languages (e.g., Python, Java, C/C++), while abstracting applications from the underlying infrastructure. In CLASS, the COMPSs programming model has been employed to enable the distribution and concurrent execution of the data analytics workflow for the object detection and air pollution use cases (see Section 3 for more details).
- ***A Deep Neural Network (DNN)*** - a suited and personalized version of YOLOv3, using the tkDNN library that exploits the capabilities of NVIDIA boards to obtain the best inference performance (for more details see D1.4 [11]).

Another contribution of CLASS has been the delivery of an EXtended PREdictability ServerlesS (EXPRESS) prototype as part of the data analytics platform (for more details, see D5.4 [7] and D5.5 [3]). In CLASS, EXPRESS has been completely redesigned as a portable solution for predictable execution of serverless functions on top of an existing serverless platform (i.e., OpenWhisk in CLASS). EXPRESS is capable of supporting several key predictable serverless function execution features, such as 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). Furthermore, EXPRESS is *portable*, in the sense that it can be implemented on top of any basic Function as a Service (FaaS) system and maintain a similar development experience for both EXPRESS functions and for regular functions.

2.2 Computation distribution

The deployment and execution of the complex data analytics workflows across the compute continuum are handled by ***COMPSs***. In addition to the programming model, mentioned in the previous section, COMPSs provides a *runtime system* that exploits the inherent parallelism of applications at execution time. COMPSs handles the scheduling and distribution of the application tasks over the compute continuum, from edge to cloud, while honouring the required data dependencies and handling any required data transfers in a way transparent to the developer.

dataClay is a distributed data store that enables applications to store and access objects in the same format they have in memory, and executes object methods within the data store. In addition, dataClay enables the execution of code next to the data. By moving computation close to the data, dataClay reduces the amount and size of data transfers between the application and the data store, thus improving performance of applications.

The specific role of dataClay in the CLASS architecture is to: i) ensure the availability of data across the compute continuum, wherever and whenever required by the data analytics, and ii) to create and maintain and periodically clean the Data Knowledge Base (DKB), which contains historical data generated by the analytics.

dataClay is natively integrated with the COMPSs framework, thus easing the development of applications that take advantage of data distribution and data locality. Furthermore, in the context of CLASS, a data manager layer (see D5.5 [3]) and some additional API calls have been implemented to integrate dataClay with Lithops-based serverless functions.

2.3 Cloud analytics platform

The cloud analytics platform provides the cloud data analytics service management and scalability features. At the cloud level, CLASS employs ***Rotterdam***, a Container as a Service (CaaS) facade which facilitates the deployment and lifecycle management of containerized applications and cloud data analytics workloads on container orchestration platforms through API calls, abstracting all the cloud infrastructure details away from developers. It includes the SLALite application, a lightweight

implementation of a Service Level Agreement (SLA) system, responsible for enforcing QoS parameters, including real-time. It makes use of Prometheus monitoring to collect the metrics that are later on evaluated in terms of SLA requirements Furthermore, a new service called **SLA Predictor** has been added to the SLALite, enabling the system to take informed decision to anticipate situations that may lead to performance degradation and thus scaling resources accordingly.

Rotterdam supports the deployment, management and monitoring of multiple containerized applications, serverless functions and workflows on multiple infrastructures simultaneously, allowing the final users to define QoS parameters and to select where to initially run their applications, from Cloud infrastructures to Edge devices and scaling or migrating the application if needed to fulfil the Quality of Service (QoS) requirements. Currently, the supported container orchestrators are Kubernetes, Openshift and Micro Kubernetes (MicroK8s). In addition, Rotterdam can also run serverless functions on Kubeless.

The *Rotterdam Caas API* provides a set of REST calls for managing the lifecycle of sets of containers (enabling actions such as uploading, organizing, executing, stopping, etc.), while abstracting all the resource infrastructure details. The API has been employed for the integration of Rotterdam and COMPSs, enabling the automatic scaling of COMPSs workers at the cloud, so as to match specific performance requirements and secure real-time guarantees.

2.4 Edge analytics platform

The edge analytics platform provides the software tools to execute real-time data-analytics methods in the edge.

A ***real-time scheduler*** for the schedulability analysis of a defined taskset at the edge has been implemented (for more details see D3.5). Given a description of a data analytics workflow as a DAG (Direct Acyclic Graph), representing the set of tasks and their dependencies, the real-time scheduler will assign each task to the most suitable resource for maximizing both efficiency and schedulability. Different engines (CPU, GPU, FPGA) can be supported, whereas different scheduling algorithms (such as EDF, FIFO, preemptive or non-preemptive) can be applied at each engine.

For the object detection and tracking applications, the CUDA model and cuDNN library are employed. The ***CUDA*** parallel programming model is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the development and execution of compute kernels. It has been created by NVIDIA and is designed to work with programming languages such as C/C++. CUDA has evolved into a highly-parallel multi-core system, which is very efficient at large data manipulation. Within CLASS, CUDA will be exploited to obtain the desired compute capability at the edge. The NVIDIA CUDA Deep Neural Network library (***cuDNN***) is a GPU-accelerated library of primitives for deep neural networks.

On top of these components, CLASS employs the NVIDIA TensorRT platform that includes a deep learning inference optimizer and runtime that delivers low latency and high-throughput for deep learning inference applications. The cross-platform OpenCV

library supporting a CUDA-based GPU interface is also used for the implementation of the object detection analytics methods.

3 Development and distribution of data analytics workflows over the CLASS SA

This section will provide an example of the execution of the CLASS use case applications of collision detection and air pollution estimation over the CLASS SA. First, a brief overview of the data analytics methods will be given, followed by a description of how these tasks are developed employing the task-based COMPSs programming model for the workflow executed at the edge, as well as the Lithops map-reduce approach. In Section 3.3, the distribution of the analytics across the compute continuum is discussed. Finally, an example of the execution and expected outcome of the data analytics over the CLASS SA is given.

3.1 Data analytics methods overview

An overview of the data analytics methods employed for the two CLASS use cases, namely the collision detection and the air pollution estimation, are shown in Figure 2. These methods have been extensively described in D1.4 [11] and D1.6 [2], so only a brief summary will be given in this section.

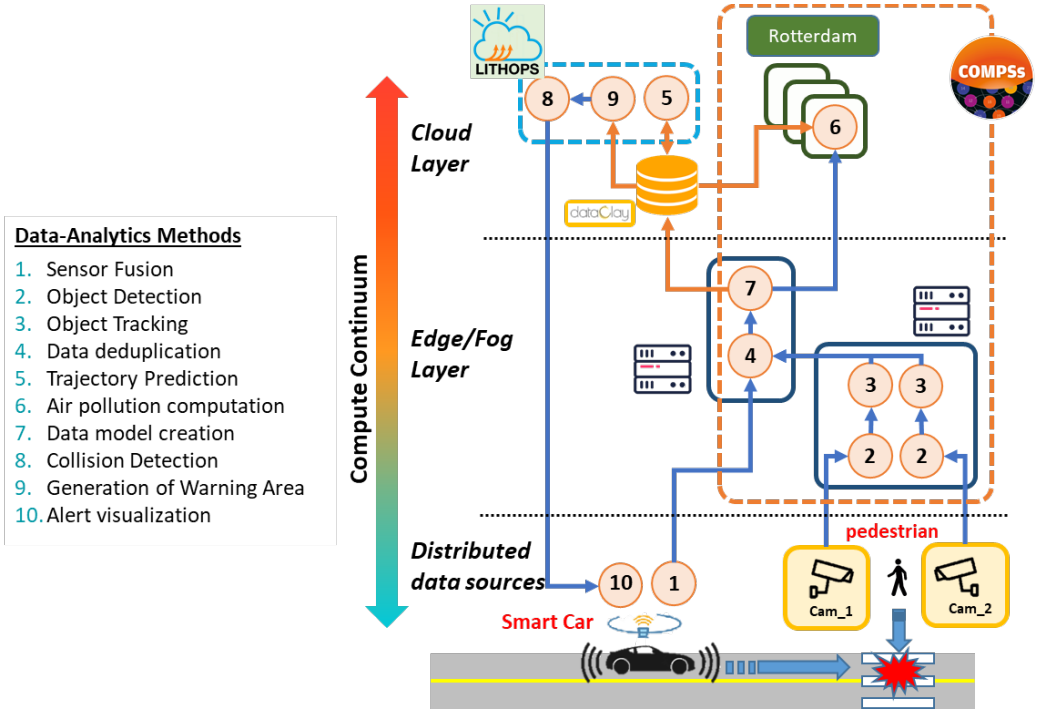

Figure 2. The complete picture of data analytics methods of the CLASS use cases executed over the CLASS infrastructure

The collision detection application consists of a COMPSs application that connects to the *"object detection"* data analytics method and invokes the *"object tracking"*, which identifies and tracks objects. Similarly, the *"sensor fusion"* method detects and tracks the objects sensed by the smart cars. The detected objects from all sources go into the

"data deduplication" method, and the output is stored in dataClay (persisting the information at the fog level) and is federated to the cloud, stored in the Data Knowledge Base (DKB). At the cloud, the *"trajectory prediction"* is invoked, updating the model with the predicted trajectories of the detected objects. Then, the *"collision detection"* method identifies potential collisions and issues a warning when needed to the involved smart/connected cars. The trajectory prediction and collision detection invocation can be either time-based or event-based, depending on the desired behavior of the system.

The air pollution estimation application is executed by periodically invoking the "air pollution computation" method, which obtains the aggregated vehicle-related data and calculates the emission levels of different pollutants, such as nitrogen oxides (NOx), carbon monoxide (CO), carbon dioxide (CO2) and particulate matter (PM).

3.2 Analytics back-ends and computation distribution

3.2.1 COMPSs-based analytics workflow at the edge

Figure 3 shows the pseudo-code of the COMPSs-based Python application that implements the data analytics executed at the edge for the two CLASS use cases.
The application iteratively executes the following functionalities encapsulated in COMPSs tasks:
– `get_detected_objects`: it connects via UDP socket, to the edge device where the "*object detection*" data analytics method (tkDNN) is being executed, namely, the edge node where the input videos from smart cameras are processed. The list of detected objects is received.
- `tracker`: it executes the "*object tracking*" data analytics method. It is implemented in C++, thus a Python binding has been implemented. Each data source (i.e., camera) is connected to a different tracker tasks, enabling the parallelization of the tracking process among multiple cameras.
- `deduplicator`: it deduplicates the objects detected by multiple sources (e.g., different cameras with partial overlapping of the covered area), returning only one copy per object.
- `create_data_model`: it executes a dataClay method to store newly detected objects or update existing objects with new events (i.e., detected positions and other relevant information), creating snapshots. Additionally, the task creates the input files with the vehicle-related information needed for the air pollution computation.
- `federate_info`: it federates the snapshots created at the fog level to the dataClay backend at the cloud (updating the DKB).
- `air_pollution_computation`: it periodically calls a containerized R application that is executed at the cloud and implements the PHEMLight[1] model for the air pollution estimation. More details on the PHEMLight model can be found in D1.2 [12] and D1.6 [2].
It should be noted that the above workflow is mostly common for both CLASS use cases, since the air pollution use case is built upon the vehicle-related information

[1] https://sumo.dlr.de/docs/Models/Emissions/PHEMlight.html

provided by the object detection and tracking analytics. The input flag "with_pollution" is passed at the time of execution to determine whether the air pollution computation will be executed or not.

```
@task(returns=list)
def get_detected_objects (camera_socket):
   return tkDNN_detected_objects(camera_id)

@task(object_list=IN, tracked_objects=IN, returns=list)
def tracker(object_list, tracked_objects):
   return track(object_list, tracked_objects)

@task(object_list=COLLECTION_IN, returns=list)
def deduplicator(tracked_objects):
   return deduplicated_obj(tracked_objects)

@task(deduplicated_objects=IN, dC_model = IN)
def create_data_model(deduplicated_obj):
   snapshot = dC_model.Create_snapshot(deduplicated_obj)
   create_air_pollution_datafile(deduplicated_obj)
   return snapshot

@task(snapshot=IN, backend_to_federate=IN)
def federate_info(snapshot, backend_to_federate):
   snapshot.federate_to_backend(backend_to_federate)

@task(self=INOUT)
def air_pollution_computation(air_pollution)
  return execute_PHEMLight()

## Main function ##
input: with_pollution
while True:
    for i, socket in camera_sockets
        obj_list = get_detected_objects (socket)
        tracked_obj[i] = tracker(obj_list, tracked_obj[i])
    deduplicated_obj = deduplicator(tracked_obj)
    snapshot = create_data_model(deduplicated_obj)
    federate_info(snapshot, external_backend_id)
    if (with_pollution):
        air_pollution = air_pollution_computation(air_pollution)
```

Figure 3. The COMPSs workflow for the CLASS use cases

Figure 4 shows the Direct Acyclic Graph (DAG) representation of the COMPSs tasks for two iterations of the workflow considering a single video source. In reality, the complexity of the DAG is much higher, since typically a higher number of iterations is

considered for each scheduling interval (resulting to a high number of generated tasks), and multiple video sources are employed. As a reference, Figure 5 shows the DAG corresponding to 1 second of processing for a single camera.

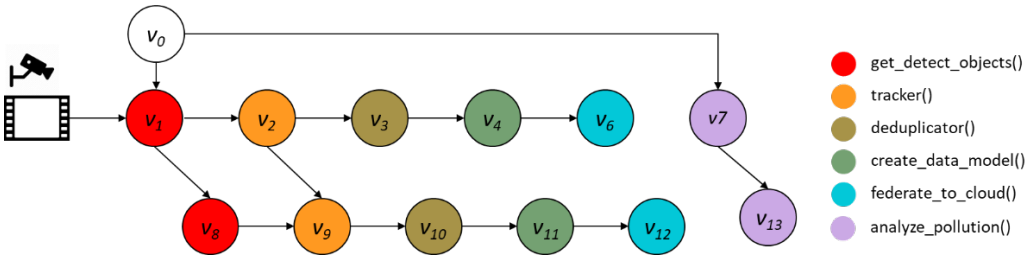

Figure 4. COMPSs DAG for two iterations of the workflow for a single camera source

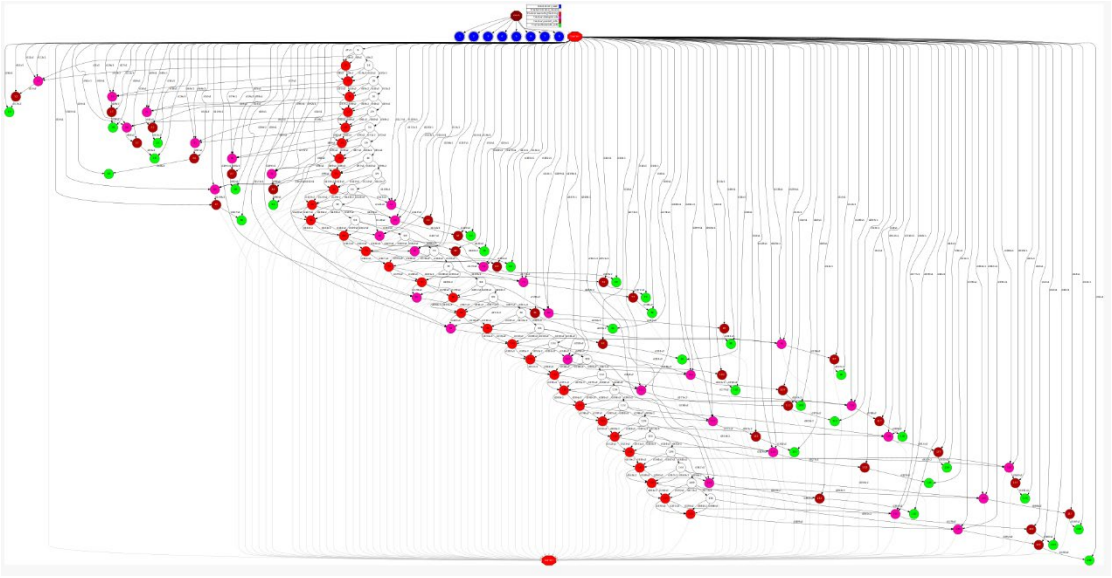

Figure 5. COMPSs DAG for 1s of processing from a single video source

The generated tasks of the DAG are then distributed and scheduled based on the implemented heuristics that aim to minimize the end to end execution time. More details on the scheduling policy can be found in D2.6 [1], whereas the achieved performance has been evaluated and presented in D2.8 [6].

3.2.2 Data analytics at the cloud

For the collision detection use case, two Lithops functions are periodically invoked in the cloud. First, the *trajectory prediction function* calculates the predicted trajectory of all tracked objects with a sufficient history of detected positions (which is a configurable parameter). The integration with the COMPSs workflow has been achieved by a Lithops wrapper function which: i) obtains all relevant dataClay objects with their events by calling a dataClay method, ii) divides them into chunks, and iii) triggers the Lithops actions for the concurrent estimation of predicted trajectories for each chunk of data based on a Python-implemented trajectory prediction method. The predicted trajectories are then appended to the corresponding objects in the DKB.

When the *collision detection function* is invoked, another Lithops wrapper function gets all the relevant objects from the DKB through a dataClay API call and executes the

Python-implemented collision detection method. The detected collisions are then published on a messaging channel[2], where they can be received by the desired endpoint (which in the use case scenario is a client at the smart cars). More information can be found in D5.4 [7] and D5.5 [3]).

The air pollution estimation is executed in a containerized R application (*PHEMLight_advance.R*[3]) that implements the PHEMLight model (for more details see 1.2 [12] and 1.6 [2]). This application is periodically invoked by the COMPSs workflow at the fog, and is executed at the cloud, either as a standalone container, or via Rotterdam (if scaling of the application is required). As an output, the air pollution use case generates a file with the estimated emissions at street level, aggregated per the area covered by each street camera. For the CLASS use case scenario, a visualization tool has been implemented to show the estimated pollution levels on a map of the MASA area.

3.3 The compute continuum infrastructure

Figure 6 shows a description of how the data analytics methods presented in the previous sections are executed on top of the CLASS software architecture, delivering the CLASS use case applications. Thanks to the capabilities of the CLASS SA, the application can be distributed across the edge and cloud infrastructure, with full coordination of resources, software components, and big-data analytics methods.

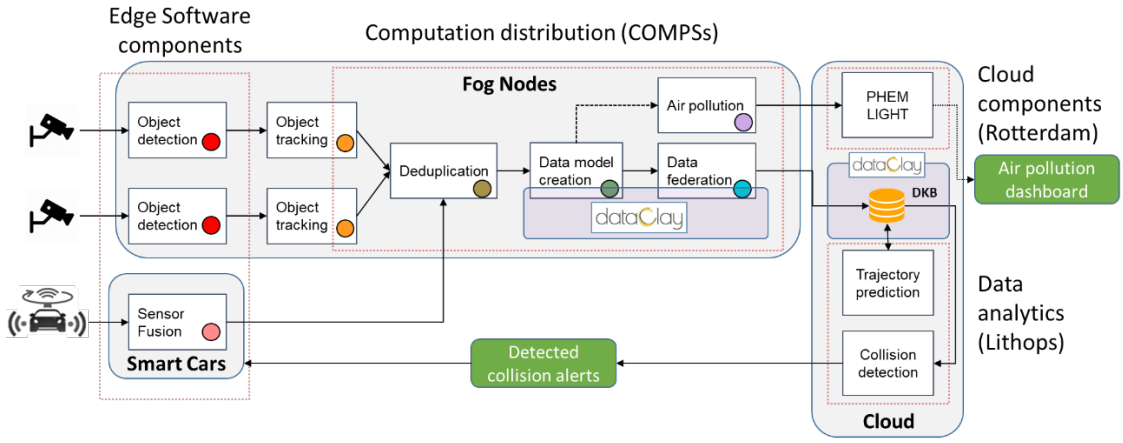

Figure 6. Description of the execution of the collision detection and air pollution estimation applications over the CLASS software architecture

The infrastructure employed for the CLASS use cases at the City of Modena has been fully described in D1.6 [2]. At the edge level, four fog nodes have been used (three connected to the video sources, and the fourth playing the role of aggregator, where the COMPSs master and the dataClay edge backend were deployed). At the cloud side, at least three separate environments have been set up, for hosting the DKB (dataClay cloud backend), the Lithops/Openwhisk environment and the Rotterdam cluster (over Openshift or Kubernetes clusters). However, it should be stressed that the presented

[2] The Message Queuing Telemetry Transport (MQTT) protocol has been used for this message exchange.
[3] https://github.com/class-euproject/phemlight-r

results of the following section are independent of the underlying infrastructure, in the sense that they can be reproduced following the provided deployment steps in any other infrastructure setup.

3.4 Example of CLASS workflow execution

This section will provide an example of the execution of the use case applications over the CLASS SA. In order to highlight the capabilities of the CLASS software architecture and provide a reproducible example that can be tested independently of the available infrastructure, the workflow depicted in Figure 7 has been used.

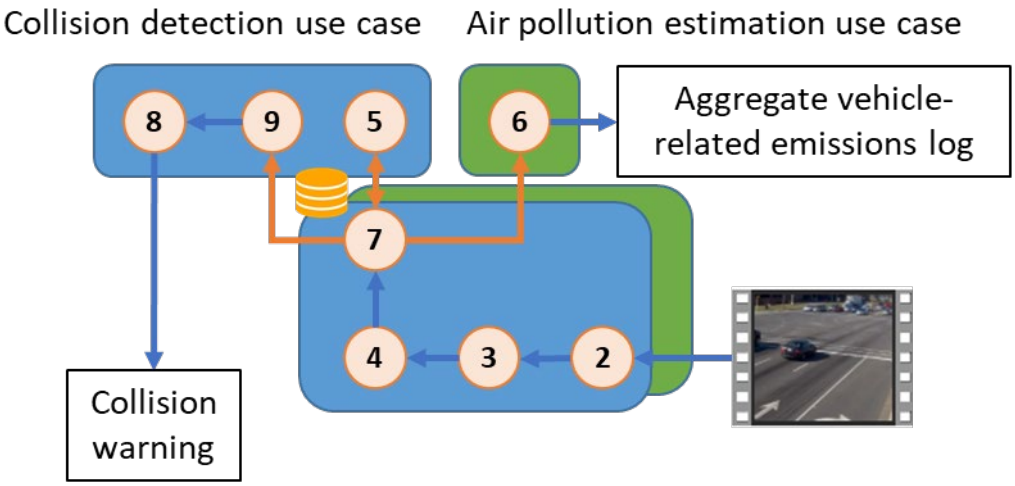

Figure 7. Data analytics methods of the CLASS use case, executed on the CLASS software architecture

The difference from the complete workflow of the use cases (depicted in Figure 2) is that, in this example, the data sources (i.e., the street cameras and the car) have been replaced by a deterministic source (i.e., a recorded video featuring a collision). This video is passed as an input to the object detection method. Upon the execution of the workflow, two different outputs are obtained: i) the output of the collision detection use case, which consists of the detected potential collisions that trigger the generation of a collision warning, and ii) the output of the air pollution estimation use case, which consists of a log file with the aggregated vehicle-related pollution emissions.

It should be stressed that the aim of this section is only to demonstrate the ability of the CLASS SA to successfully execute the data analytics of the CLASS use cases. The performance evaluation of the analytics and the end to end execution of the use cases can be found in D1.6 [2].

3.4.1 Collision detection execution example

To showcase the execution of the collision detection use case, two different examples have been selected.

In the ***first example***, a recorded video that captures a collision scenario set up between two CLASS vehicles, i.e., the white Maserati smart car and a connected car has been used as the main source of data[4]. An example of a detected potential collision is shown in Figure 8. A visualization tool developed within CLASS has been employed to graphically show the output of the collision detection workflow execution. The information printed on the video frame has been extracted from the DKB at the cloud, and refers to the snapshot of a given video frame once trajectory prediction and collision detection have been applied.

The green dots represent the tracked positions of the vehicles during the last 20 frames. The red lines represent the predicted trajectories, which in this example, correspond to 8 predicted points estimated within 500 ms intervals (i.e., an overall 4 seconds prediction of the vehicle movement). Finally, the detected potential collision is depicted as two black circles at the point where the predicted trajectories of the involved vehicles cross.

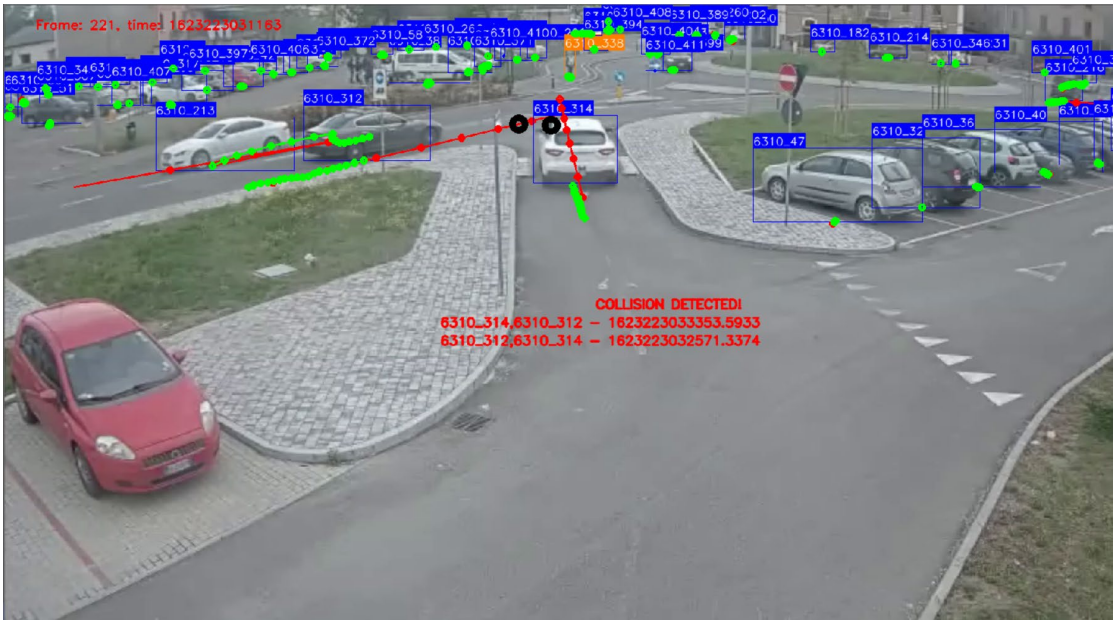

Figure 8. Example of a detected potential collision between two CLASS vehicles in the MASA area

The first experiment is not reproducible outside CLASS because the recorded video used as an input cannot be openly shared, due to privacy concerns (applying anonymization techniques since blurring before the video processing would affect the performance of the object detection analytics).

Hence, a ***second fully reproducible example*** has been provided, based on a publicly available video. The video has been recorded as part of the "Red-Light Safety Camera Program[5]" and depicts a collision taking place at the intersection of Northbound Owen

[4] In this specific execution, in addition to the recorded video, the workflow was processing two additional live streams from the cameras at the MASA area.

[5] The project has bene implemented by the Joint City of Fayetteville and and Cumberland County Liaison Committee. The video can be downloaded from https://www.fayettevillenc.gov/city-services/public-

Drive and Village Drive, at Fayetteville NC. The execution of the CLASS analytics on this video return a collision warning before the actual collision takes place, as shown in Figure 9. For convenience (due to the low resolution of the image), only information relevant to the two involved vehicles is depicted.

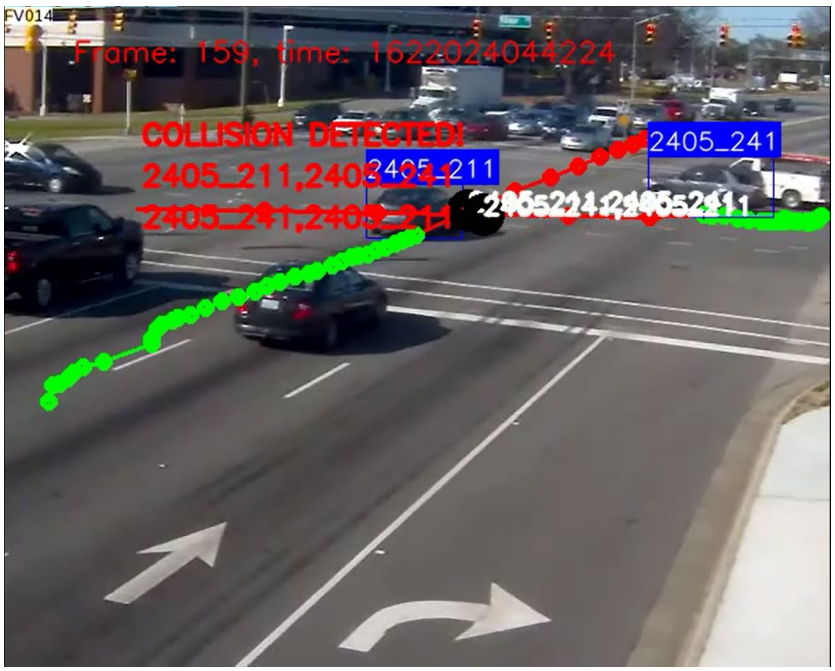

Figure 9. Example of a detected potential collision between two cars in a publicly available video

3.4.2 Air pollution estimation example

This section will present the output of the air pollution estimation over the first example (using the CLASS recorded video). It should be noted that in order to obtain meaningful statistics for the air pollution, a much longer execution of the workflow should take place (in the order of hours or days, if possible).

The information regarding the number and type of vehicles present in a frame, as well as their estimated speed at a given timestamp (i.e., corresponding to a given snapshot processed at the edge) is passed as an input for the air pollution calculation based on the PHEMLight model (Figure 10). The output of the model, executed in the cloud, is the estimated amount of pollutants (in g/s) for the area captured by the camera (Figure 11).

A simple web application has been developed in CLASS, to visualize the output of the PHEMLight calculation. In this example, the output of two cameras is shown in Figure 12 (one corresponding to the intersection where the detected collision took place and the other to the live feed from a MASA street camera).

services/traffic-services/red-light-camera-program [Last accessed 30 June 2021) and is also included in the CLASS repository.

Similar results have been obtained using the second example. However, the air pollution visualization application has been customized to reflect the area covered by the street cameras in MASA and adapting it to another location is beyond the interest of CLASS.

	A	B	C	D	E
1	VehID	LinkID	Time	Vehicle_type	Av_link_speed
2	6310_512	6310	1625044950558	Car	1
3	6310_510	6310	1625044950558	Car	0
4	6310_509	6310	1625044950558	Car	19
5	6310_508	6310	1625044950558	Car	0
6	6310_506	6310	1625044950558	Car	12
7	6310_496	6310	1625044950558	Car	6
8	6310_475	6310	1625044950558	Car	0
9	6310_474	6310	1625044950558	Car	0
10	6310_466	6310	1625044950558	Car	0
11	6310_465	6310	1625044950558	Car	6
12	6310_458	6310	1625044950558	Car	9
13	6310_427	6310	1625044950558	Car	30
14	6310_422	6310	1625044950558	Car	0
15	6310_418	6310	1625044950558	Car	9
16	6310_379	6310	1625044950558	Car	5
17	20936_665	20936	1625044950558	Car	0
18	20936_658	20936	1625044950558	Car	0
19	20936_657	20936	1625044950558	Car	4
20	20936_656	20936	1625044950558	Car	1
21	20936_655	20936	1625044950558	Car	2
22	20936_649	20936	1625044950558	Car	0
23	20936_648	20936	1625044950558	Car	29
24	20936_638	20936	1625044950558	Car	76
25	20936_634	20936	1625044950558	Car	146
26	20936_610	20936	1625044950558	Car	0
27	20936_603	20936	1625044950558	Car	0
28	20936_596	20936	1625044950558	Car	23
29	20936_576	20936	1625044950558	Car	137

Figure 10. Example of the input file for the PHEMLight model generated by the CLASS data analytics at the edge

	A	B	C	D	E	F	G	H	I	J	K
1	LinkID	int	Hr	link_speed_av	NOx	HC	CO	PM	PN	NO	veh_group
2	636	451401375	451388836	40.57635468	0.268097341	0.125747804	0.807217378	0.006385392	7.00553E+12	0.136615526	car
3	6310	451401375	451388836	2.320591862	0.731045493	0.528598276	3.185469381	0.01692593	1.83468E+13	0.34072143	car

Figure 11. Example of the PHEMLight output, estimating the level of different pollutants, aggregated by the ID of the video sources (LinkID)

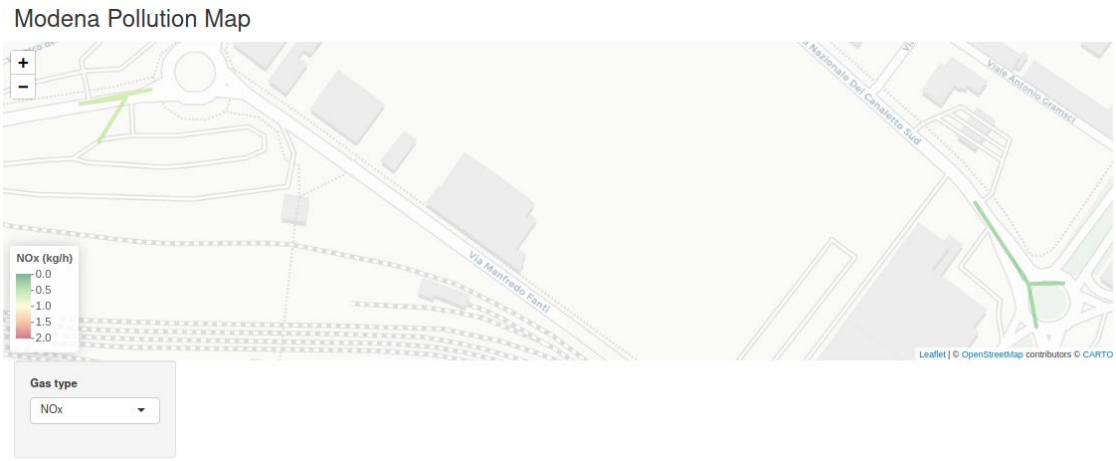

Figure 12. Visualization of the PHEMLight output for two camera inputs at the MASA area in Modena, Italy.

4 Deployment of the CLASS Software Architecture

In this section, we will provide a step by step guide for the deployment of the CLASS SA components. The instructions are intended to help any interested developer to deploy the full stack of the CLASS software architecture, as described in Figure 1, and reproduce the examples provided in Section 3.4. In order to better highlight the usage of the different components, the deployment process has been organized into four subsections, corresponding to four different target scenarios (that could be executed independently or concurrently):

i) Deployment of the SA components required for the execution of the collision detection use case (to reproduce the example of Section 3.4.1)
ii) Deployment of the SA components required for the execution of the air pollution use case (to reproduce the example of Section 3.4.2)
iii) Deployment of the SA components for the scaling of the workflow at the cloud using Rotterdam
iv) Deployment of the EXPRESS platform

The full deployment of the CLASS software architecture assumes the availability of computing resources at both edge and cloud level. All components are containerized, allowing for flexible and scalable deployment over heterogeneous infrastructures. However, it should be noted that for the integration of all the software components, additional configuration is needed, which is tightly dependent on the available infrastructure and networking (e.g., IP and port configurations, etc.).

Finally, additional details and clarifications on the deployment of the different components can be consulted:

- In the technical deliverables of the corresponding work (as summarized in Table 1), namely: WP2 (for COMPSs and dataClay), WP3 (for the edge platform), WP4 (for Rotterdam) and WP5 (for the data analytics platform).
- In the readme files of the corresponding repositories of the CLASS project, at https://github.com/class-euproject.

4.1 Collision detection use case deployment

In this section, the steps to deploy and execute the software components for the collision detection use case are provided.

4.1.1 Computation distribution layer and edge platform setup

Preparation of the environment at the edge nodes

At the edge level, due to the specific requirements of the data analytics employed for object detection, the availability of NVIDIA GPUs is required. The generated images assume an amd64 / x86_64 architecture, however, they could be adapted for other architectures, such as arm64.

For convenience, three types of edge nodes are defined, based on their functionality:

i) The ***master node*** is the edge node where the COMPSs master and the dataClay instance (dataClay-edge) are deployed.

ii) The ***video-source nodes*** are the edge nodes that are connected to one or more video sources, generating the video capture on which object detection will be performed. The video sources can be cameras or recorded videos.

iii) The ***worker nodes*** are the edge nodes where COMPSs workers are deployed by the COMPSs master, for the execution of the task-based data analytics workflow.

In the simplest scenario, a single edge node could implement all three functionalities, however a higher number of nodes is recommended to exploit the distribution capabilities of the CLASS software architecture.

In the provided deployment example, three edge nodes have been considered: a master node (with IP 192.168.0.4), a video-source node (with IP 192.168.2) and a worker node (with IP 192.168.0.3). The video-source node receives video frames from the publicly available collision video considered in the second example of Section 3.4.1 (assigned the camera id 2405).

The following steps should be taken at ***all edge nodes***:

1. Install Docker (version 19.03.06) from: https://docs.docker.com/engine/install/
2. Install Docker Compose (version 1.19.0) from: https://docs.docker.com/compose/install/
3. Clone the dataclay-class repository

```
$ git clone https://github.com/class-euproject/dataclay-class.git -b dockers
```

4. Download the docker image for the object detection based on the tkDNN library[6], including CUDA and other dependencies required (e.g., OpenCV, TensorRT, etc.):

```
$ docker pull bscppc/class-object-detection
```

5. Download the docker image for the COMPSs-based application workflow.

```
$ bscppc/class-object-tracking
```

This image mainly includes: i) an adapted version of the COMPSs framework (release 2.7) with additional functionalities for the integration with the CLASS SA (the advanced workflow scheduler, introduced in D2.6 [1], and the connector with the Rotterdam CaaS, introduced in D4.7 [13]), and ii) the COMPSs application[7], including the libraries and analytics methods for the tracking[8] and deduplication.

For the ***video-source nodes***, the following additional steps must be followed

6. Install the Nvidia Docker Toolkit (nvidia-docker2) from:

https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html

7. Deploy the tkDNN, exporting the video source id and the port. The default values given below correspond to the example presented in Section 3. Note that if multiple video sources are connected to a single node, the corresponding tkDNN containers should be configured and launched.

```
$ cd dataclay-class/dataclay-edge/
$ export CAM_ID=2405
$ export TKDNN_PORT=5560
$ docker-compose -f docker-compose-cuda.yml up
```

Deployment of dataClay at the cloud

With respect to the distribution layer, a dataClay backend must be deployed at the cloud, for the maintenance of the DKB where the information generated by the data analytics will be aggregated and stored.

1. To prepare the environment at the cloud, the same steps 1-3 should be repeated at the cloud node (i.e., the installation of docker and docker compose, as well as the cloning of the dataClay-class).
2. Update the parameters of the *dataClay-class/dataClay-cloud/docker-compose.yml* file to match the available infrastructure.

[6] https://github.com/class-euproject/tkDNN/tree/udpsockets

[7] https://github.com/class-euproject/COMPSs-obstacle-detection/blob/udpsockets/tracker.py

[8] https://github.com/class-euproject/class-edge/tree/udpsockets

3. Deploy dataClay at the cloud by executing:

```
$ cd dataClay-class/dataClay-cloud
$ ./launch_dockers.sh
```

Configuration of the COMPSs master node at the edge

After all the previous steps have been complete, the following configurations must take place at the master node.

1. Update the parameters of the *dataClay-class/dataClay-edge/docker-compose.yml* file to match the available infrastructure at the edge. Set the variables IMPORT_MODELS_FROM_EXTERNAL_DC_HOSTS and IMPORT_MODELS_FROM_EXTERNAL_DC_PORTS to reflect the IP and LOGICMODULE port defined at the dataClay backend at the cloud.
2. Launch dataClay by executing

```
$ cd dataclay-class/dataclay-edge/
$ ./launch_dockers
```

3. Connect to the container by executing

```
$ docker exec -it dataclayedge_object-tracking_1 bash
```

   and update the dataClay configuration in the file *cfgfiles/client.properties* to point to the dcinitializer container's IP.

4. Update the *project.xml* and *resources.xml* files to reflect the available resources (IPs, computing units, etc.) that can be used for the execution of COMPSs tasks. The COMPSs master node will use this information to deploy COMPSs workers where the application tasks will be distributed. More details and examples for the generation of this files can be found in COMPSs documentation[9].

4.1.2 Data analytics platform setup (Lithops/Openwhisk environment)

This section provides the procedure for the execution of the Trajectory Prediction (TP) and Collision Detection (CD) data analytics methods using Lithops over an OpenWhisk environment. In CLASS, this environment has been set up over a Kubernetes cluster at the cloud node. The specific setup considered in CLASS, consisting of 1 master node and 4 workers, is described in detail in D5.5 [3].

Setup of the Lithops/Openwhisk environment

1. Install Ubuntu Server 18.04 on master and worker node machines. For optimal performance, it is recommended to choose worker nodes with similar hardware capabilities: https://releases.ubuntu.com/18.04/

[9] https://compss-doc.readthedocs.io/en/stable/Sections/01_Installation/06_Configuration_files.html

2. Setup a Kubernetes cluster on the master and add the worker nodes. Some helpful guidelines can be found here: https://phoenixnap.com/kb/install-kubernetes-on-ubuntu
3. Add a storage class (e.g., NFS) for persistence support, as indicated here: https://kubernetes.io/docs/concepts/storage/storage-classes/
4. Install OpenWhisk on top of Kubernetes cluster: https://github.com/apache/openwhisk-deploy-kube
5. Setup a local docker registry following the instructions here: https://docs.docker.com/registry/deploying/
6. Setup an mqtt broker, such as Eclipse Mosquitto: https://artifacthub.io/packages/helm/t3n/mosquitto
7. Clone the Lithops repository from the CLASS github project to the root directory: https://github.com/class-euproject/lithops/tree/master/runtime
8. Clone the collision detection repository from the CLASS github project to the root directory: https://github.com/class-euproject/collision-detection
9. Create Lithops runtime by following the instructions on the readme file at: https://github.com/class-euproject/lithops/tree/master/runtime

Trajectory Prediction and Collision Detection methods

The following steps describe the procedure to configure and launch the trajectory prediction and collision detection analytics. These steps should take place after the completion of the setup described in Section 4.1.1 for the deployment of the dataClay cloud back-end.

1. Update the *collision-detection/update_all.sh* script to reflect the setup. This script should include the location from where to retrieve the dataClay stubs, where the input data for the trajectory prediction and collision detection methods are stored. Specifically, an ssh connection to the VM where the dataClay cloud instance is launched must be established.
2. Then run:

```
$ cd collision_detection && ./update_all.sh
```

3. Consequent updates that involve only dataclay stubs can be done faster by running the following script from the collision detection folder (again, after updating it to reflect the current setup):

```
$ fast_update_all.sh
```

4.1.3 Execution of the collision detection use case analytics

1. Steam up workers environment running the following script from the collision detection folder in the Lithops environment (see section 4.1.2):

```
$ cd collision_detection
```

```
$ ./steam_up.sh
```

2. Enter in the dataclayedge_object-tracking_1 container where the COMPSs application is deployed by running.

```
$ docker exec -it dataclayedge_object-tracking_1 bash
```

3. Execute the COMPSs-based workflow, by properly setting the required IP and port settings of the COMPSs master node and of the video source node, as well as the path to the .xml configuration files (marked in red).

```
$ runcompss --master_name=192.168.0.4 \
            --master_port=43001 \
            --python_interpreter=python3 \
            --project=config/project.xml \
            --resources=config/resources.xml \
            --storage_conf=/root/COMPSs-obstacle-
      detection/cfgfiles/session.properties \
            --classpath=/root/COMPSs-obstacle-
      detection/dataclay/dataclay.jar \
            tracker.py 192.168.0.2:5560
```

Execution with a different video source

For this example, the recorded video taken as an input has been included in the tkDNN image for convenience. In order to apply the workflow to a different video source, the video source configuration file should be updated. To do this:

1. Access the tkDNN container at the video-source node

```
$ docker exec -it dataclayedge_object-detection_1 bash
```

2. Open the video source configuration file that can be accessed at:

   root/class-edge/data/all_cameras_en.yaml

3. Modify the camera entry (or add a new one with a unique camera id) to reflect the parameters of the new video source. For reference, the current entry for the recorded is:

```
- maskfile: !<!> ../../data/masks/owen_village.jpg
  input: !<!> ../../data/FayetteVille-OwenVillage.mp4
  cameraCalib: !<!> ../../data/calib_cameras/20936.params
  id: 2405
  encrypted: 0
  pmatrix: !<!> ../../data/pmat_new/proj_matrix_owen_village.txt
  maskFileOrient: !<!> ../data/masks_orient/1920-1080_mask_null.jpg
  tif: !<!> ../../data/fayetteville_owenvillagedr.tif
```

   To point to a new video source, the input variable should be updated with the new path, which can either be a different video file or a live stream from a camera. Furthermore, the projection matrix (pmatrix) for the new video should be provided, as well as a georeferenced tif file for the covered area (required for the conversion of pixel positions to GPS coordinates).

4.2 Air pollution estimation use case deployment

The steps for launching the containerized air pollution application at the cloud are given next.

Prepare the environment for PHEMLight execution at the cloud

1. Pull the air pollution application (PHEMLight) and visualizer (shiny) at the cloud, both contained in a single image:

```
$ docker pull bscppc/r-poll_dash
```

2. Create a new folder named "pollutionMap" and clone the source code:

```
$ mkdir ~/pollutionMap && cd ~/pollutionMap

$ git clone https://github.com/class-euproject/pollution-visualization

$ git clone https://github.com/class-euproject/phemlight-r.git
```

PHEMLight visualization

1. In order to run the PHEMLight visualizer, before executing the COMPSs workflow at the edge, access the visualizer container (shiny_cont) with the following command:

```
$ docker run -d --rm --name shiny_cont -p 5554:8888 \
       --name pollShiny_cont --group-add users
       --user "$(id -u)"
       -w="~/pollutionMap/pollution-visualization/R"
       --mount source=pollVolume,destination=~/pollutionMap
       jupyter/r-poll_dash Rscript dashboard-Modena.r
```

2. Access the service from a browser, at the IP of the host VM at the cloud where PHEMLight is deployed, at port 5554, e.g., http://CLOUD-IP:5554/

Note that the PHEMLight visualizer is customized for the MASA area in Modena where the CLASS use cases are executed, so it cannot be directly applied to the recorded video. To plot a different area:

3. Access the "/R/createRoads.R" in the PHEMLight container (phemlight_cont)
4. Introduce lines to map the streets covered by the new video source(s)
5. After modifying the script execute

```
$ 'Rscript R/create_roads.R'
```

   to create 'Data/test_roads.csv', which is the file which used by the map application

Preparation of the COMPSs workflow at the edge

1. For the air pollution estimation use case, the same COMPSs workflow developed for the collision use case is used. Hence, if not previously set up, the steps described in Section 4.1.1 must be followed.
2. The COMPSs workflow at the edge will generate the necessary input files and launch the air pollution container (*phemlight_cont*) at the selected intervals

(everytime the *air_pollution_computation method* is called). This is done by establishing an ssh connection to the cloud and launch a script that starts the container. The details of this connection must be configured appropriately by accessing the container:

```
$ docker exec -it dataclayedge_object-tracking_1 bash
```

and modifying the tracker.py where the ssh connection is established (marked in red):

```
ssh CLOUD_USERNAME@CLOUD_IP nohup bash ~/pollutionMap/phemlight-
r/dockerScripts/phemlightCommand.sh {a} {b} {c} &>/dev/null &
```

Make sure to configure the SSH keys between the involved machines (master and cloud node where PHEMLight is executed) so that a passwordless connection can be established.

3. In the same file (tracker.py), the frequency for invoking the air pollution computation can be also configured, by setting the parameter NUM_ITERS_POLLUTION. By default, the computation is called every 300 frames (approximately every 30 seconds).

4. Finally, to execute the use case, at the master node (the edge node 192.168.0.4) run the COMPSs worklow as indicated in Section 4.1.3, but with the "with_pollution" flat, i.e.,:

```
$ runcompss --master_name=192.168.0.4 \
            --master_port=43001 \
            --python_interpreter=python3 \
            --project=config/project.xml \
            --resources=config/resources.xml \
            --storage_conf=/root/COMPSs-obstacle-
      detection/cfgfiles/session.properties \
            --classpath=/root/COMPSs-obstacle-
      detection/dataclay/dataclay.jar \
            tracker.py 192.168.0.2:5560 \
            --with_pollution
```

4.3 Deployment of the cloud analytics platform (Rotterdam)

The steps for installing Rotterdam at the cloud node have been provided in detail in the Section 4.2 of deliverable D4.7 [13]. For convenience a summary of the steps is included here, but the complete set of instructions to follow should be consulted in D4.7.

1. Install the container environment (Openshift or Kubernetes cluster)
2. Install and configure monitoring tools (Prometheus Pushgateway and Grafana)
3. Install Rotterdam, available in the CLASS repository: https://github.com/class-euproject/Rotterdam
4. A docker image can also be found in: https://hub.docker.com/r/atosclass/rotterdam-caas

5. Install and configure the SLA manager, provided as a docker image in: https://hub.docker.com/r/atosclass/slalite
6. To launch the COMPSs-based application (e.g., the air pollution estimation or any other task-based application in general), the COMPSs resource.xml and project.xml files must be updated accordingly to reflect the Rotterdam configuration. For specific details, see D4.7.

4.4 Deployment of the EXPRESS platform

The EXPRESS prototype is available at the CLASS git repository at https://github.com/class-euproject/express and has been delivered as a standalone component of the CLASS SA. Detailed instructions on its installation can be found in Section 3.1.2 of deliverable D5.4 [7].

Acronyms and Abbreviations

API – Application Programming Interface
Caas – Container as a Service (CaaS)
CLI – Command Line Interface
D – Deliverable
DAG – Direct Acyclic Graph
DKB – Data Knowledge Base
DNN – Deep Neural Network
EXPRESS – EXtended PREdictability ServerlesS
FaaS – Function as a Service
MASA – Modena Automotive Smart Area
MQTT – Message Queuing Telemetry Transport
MS – Milestone
SA – Software Architecture
SDK – Software Development Kit
SLA – Service Level Agreement
QoS – Quality of Service
WP – Work Package

References

[1] CLASS, "D2.6 - Second release of the CLASS Software Architecture," July 2020.

[2] CLASS, "D1.6 - Use case evaluation," June 2021.

[3] CLASS, "D5.5 - Final Release of CLASS Big-Data Analytics Layer," June 2021.

[4] CLASS, "D4.6 - Validation of the Cloud Data Analytics Service Management and Scalability Components," March 2021.

[5] CLASS, "D3.6 - Validation of the CLASS edge computing subsystem," June 2021.

[6] CLASS, "D2.8 - Evaluation of the CLASS Software Architecture," June 2021.

[7] CLASS, "D5.4 - FInal release of an augmented platform for analytics workloads," July 2020.

[8] CLASS, "D2.1 - CLASS Software Architecture Requirements and Integration Plan," 2018.

[9] CLASS, "D2.4 - First release of the CLASS software architecture," March 2019.

[10] Apache OpenWhisk, "Apache OpenWhisk is a serverless, open source cloud platform," Apache Foundation, [Online]. Available: http://openwhisk.apache.org/. [Accessed 14 June 2021].

[11] CLASS, "D1.4 - Final release of the smart city use case," July 2020.

[12] CLASS, "D1.2 - Final release of the Smart City Use-Cases," March 2019.

[13] CLASS, "D4.7 - Final release of the Cloud Data Analytics Service Management and Scalability components," July 2020.

[14] A. Melani, M. A. Serrano, M. Bertogna, I. Cerutti, E. Quinones and A. Buttazzo, "A static scheduling approach to enable safety-critical OpenMP applications," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.

[15] M. Pinedo, Scheduling: theory, algorithms, and systems, Springer-Verlag, New York, 2012.

[16] K. E. Raheb, C. T. Kiranoudis, P. P. Reoussis and C. D. Tarantilis, "Production scheduling with complex precedence constraints in parallel machines," *Computing and Informatics,* vol. 24, no. 3, pp. 297-319, 2012.

[17] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna and E. Quiñones, "Timing characterization of OpenMP4 tasking model," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[18] CLASS, "D3.3 - Final release of Edge Analytics Platform Agent," July 2020.

[19] CLASS, "D3.5 - Final release of the real-time analysis methods and tools on the edge," July 2020.

[20] IBM ILOG, "CPLEX optimization studio," url: https://www.ibm.com/products/, 2014.

[21] A. Tirumala, J. M. Dugan, J. Ferguson and K. A. Gibbs, "iPerf: TCP/UDP bandwidth measurement tool," 2005.

[22] IEEE Standard for Ethernet, "IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015),," August 2018.

[23] E. Jones , T. Oliphant, P. Peterson and et al., "SciPy: Open source scientific tools for Python," 2001.

[24] J. Levinson, J. Askeland, J. Becker and et al., "Towards fully autonomous driving: Systems and algorithms," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, Baden-Baden, 2011.

[25] E. Mezzetti and T. Vardanega, "On the industrial fitness of WCET analysis," in *11th International Workshop on Worst-Case Execution Time analysis*, 2011.

[26] CLASS Deliverables, "D1.2 First release of the smart city use cases," 2019.