A Cognitive Serverless Framework for the Cloud-Edge Continuum

D5.3 Use Cases - Scientific Report - c

Version 1.0

30 April 2024

Abstract

COGNIT is an AI-Enabled Adaptive Serverless Framework for the Cognitive Cloud-Edge Continuum that enables the seamless, transparent, and trustworthy integration of data processing resources from providers and on-premises data centers in the cloud-edge continuum, and their automatic and intelligent adaptation to optimise where and how data is processed according to application requirements, changes in application demands and behaviour, and the operation of the infrastructure in terms of the main environmental sustainability metrics. This document provides an overall status of the contribution of the Project's software requirements towards meeting the user requirements that guide the development of the COGNIT Framework, offers additional information about the domains targeted by the Use Cases and the Partners involved in them, and provides an update on the Project's software integration process and infrastructure, on its testbed environment, and on the progress of the software requirement verification tasks during the Second Research & Innovation Cycle (M10-M15).

This project is funded by the European Union's Horizon Europe research and innovation programme under Grant Agreement 101092711 – SovereignEdge.Cognit

Deliverable Metadata

Project Title:	A Cognitive Serverless Framework for the Cloud-Edge Continuum
Project Acronym:	SovereignEdge.Cognit
Call:	HORIZON-CL4-2022-DATA-01-02
Grant Agreement:	101092711
WP number and Title:	WP5. Adaptive Serverless Framework Integration and Validation
Nature:	R: Report
Dissemination Level:	PU: Public
Version:	1.0
Contractual Date of Delivery:	31/03/2024
Actual Date of Delivery:	30/04/2024
Lead Author:	Thomas Ohlson Timoudas (RISE)
Authors:	Monowar Bhuyan (UMU), Marek Białowąs (Phoenix), Dominik Bocheński (Atende), Simon Bonér (UMU), Malik Bouhou (CETIC), Aritz Brosa (Ikerlan), Idoia de la Iglesia (Ikerlan), Sébastien Dupont (CETIC), Agnieszka Frąc (Atende), Grzegorz Gil (Atende), Torsten Hallmann (SUSE), Joan Iglesias (ACISA), Mateusz Kobak (Phoenix), Tomasz Korniluk (Phoenix), Johan Kristiansson (RISE), Antonio Lalaguna (ACISA), Martxel Lasa (Ikerlan), Carlos Lopez (ACISA), Marco Mancini (OpenNebula), Alberto P. Martí (OpenNebula), Philippe Massonet (CETIC), Nikolaos Matskanis (CETIC), Behnam Ojaghi (ACISA), Daniel Olsson (RISE), Goiuri Peralta (Ikerlan), Samuel Pérez (Ikerlan), Holger Pfister (SUSE), Tomasz Piasecki (Atende), Francesco Renzi (Nature4.0), Bruno Rodríguez (OpenNebula), Juan José Ruiz (ACISA), Kaja Swat (Phoenix), Paul Townend (UMU), Iván Valdés (Ikerlan), Riccardo Valentini (Nature4.0), Constantino Vázquez (OpenNebula), Pavel Czerny (OpenNebula).
Status:	Submitted

Document History

Version	Issue Date	Status[1]	Content and changes
0.1	25/04/2024	Draft	Initial Draft
0.2	29/04/2024	Peer-Reviewed	Reviewed Draft
1.0	30/04/2024	Submitted	Final Version

Peer Review History

Version	Peer Review Date	Reviewed By
0.1	29/04/2024	Torsten Hallmann (SUSE)
0.1	29/04/2024	Antonio Álvarez (OpenNebula)

Summary of Changes from Previous Versions

First Version of Deliverable D5.3

[1] A deliverable can be in one of these stages: Draft, Peer-Reviewed, Submitted, and Approved.

Executive Summary

Deliverable D5.3, released at the end of the Second Research & Innovation Cycle (M15), is the third version of the Use Cases Scientific Report in WP5 ”Adaptive Serverless Framework Integration and Validation”. It offers a summary of the work done in this cycle and for the demonstration of the first version of the COGNIT Framework and its integration with the Use Cases to be demonstrated in the testbed environment for the project. Also, it provides information on the progress of the software requirements verification tasks per component

In connection with the main components of the COGNIT Architecture (i.e. Device Client, Serverless Runtime, Provisioning Engine, Cloud-Edge Manager, and AI-Enabled Orchestrator), the Project has delivered progress specifically in those Software Requirements needed to achieve Milestone 2 in M15, providing a basic set of functionalities for the Use Cases to launch their own research and development activities and start integrating their devices with the COGNIT Framework.

Apart from this document (Deliverable D5.3) and the Project's global overview provided through Deliverable D2.3, specific research and development activities performed in WP3 “Distributed FaaS Model for Edge Application Development” (related to the Device Client, the Serverless Runtime, the Provisioning Engine, and the Secure and Trusted Execution of Computing Environments) are described in detail in reports D3.2 “COGNIT FaaS Model - Scientific Report - b” and D3.7 “COGNIT FaaS Model - Software Source - b”, whereas those performed in WP4 “AI-Enabled Distributed Serverless Platform and Workload Orchestration” (related to the Cloud-Edge Manager, the AI-Enabled Orchestrator, and the Energy Efficiency Optimization in the Multi-Provider Cloud-Edge Continuum) are described in reports D4.2 “COGNIT Serverless Platform - Scientific Report - b” and D4.7 “COGNIT Serverless Platform - Software Source - b”.

The information in this report will be updated with incremental releases at the end of each research and innovation cycle (i.e. M21, M27, and M33).

Table of Contents

Abbreviations and Acronyms

3GPP	3rd Generation Partnership Project
4G	Fourth Generation Mobile Network
5G	Fifth Generation Mobile Network
6G	Sixth Generation Mobile Network
5GAA	5G Automotive Association
AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CCAM	Cooperative, Connected and Automated Mobility
C-V2X	Cellular Vehicle to Everything communication
DaaS	Data as a Service
DSRC	Dedicated Short Range Communications
EC2	(Amazon) Elastic Compute Cloud
EN	European Norms
ETSI	European Telecommunications Standards Institute
EV	Electric Vehicle
FaaS	Function as a Service
FAQ	Frequently Asked Questions
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ID	Identifier
IP	Internet Protocol
IPv6	Internet Protocol version 6
ITS	Intelligent Transport System
kW	kiloWatt(s)
LTE	Long-Term Evolution
MAPEM	MAP (Topology) Extended Message
ML	Machine Learning
M-HUB	Mobility Hub (advanced TLC)
OBU	On Board Unit
OCPP	Open Charge Point Protocol

OS	Operating System
PCI	Peripheral Component Interconnect
PV	Photovoltaic
QA	Quality Assurance
QoS	Quality of Service
RES	Renewable Energy Source
REST	Representational State Transfer
RSU	Road Side Unit
RTOS	Real-Time Operating System
SAE	Society of Automotive Engineers
SEM	Smart Energy Meter
SPATEM	Signal Phase And Timing Extended Message
SREM	Signal Request Extended Message
SSEM	Signal request Status Extended Message
SSH	Secure Shell
SSL	Secure Sockets Layer
SUMO	Simulation of Urban Mobility[2]
TCC	Traffic Control Center
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TLC	Traffic Light Controller
TS	Technology Specifications
TSP	Traffic/Transit Signal Priority
V2X	Vehicle to Everything communication technology
VM	Virtual Machine
VPN	Virtual Private Network

[2] An open source, highly portable, microscopic and continuous multi-modal traffic simulation package designed to handle large networks (https://eclipse.dev/sumo/).

1. Introduction

This is the third version of the Use Cases Scientific Report. The initial version of the Use Cases Scientific Report (Deliverable D5.1), released in M3, provided an initial collection of user requirements, a description of each of the Use Cases—including an initial architecture design and a plan for the demonstration and validation to take place in Tasks T5.3, T5.4, T5.5, T5.6—and an update of the COGNIT Testbed environment.

The second version of the report (Deliverable D5.2), released in M9, provided a summary of the overall status of the contribution of the Project's software requirements towards meeting the user requirements that guide the development of the COGNIT Framework at the end of the First Research & Innovation Cycle (M4-M9). It also provided additional information about the domains targeted by the Use Cases and the Partners involved in them, listing new user requirements identified during the cycle, and offering an update on the Project's software integration process and infrastructure, on its testbed environment, and on the progress of the software requirement verification tasks per component.

Following the same structure as in previous versions, D5.3 is composed of an introductory section and nine additional sections organised in two main blocks of content:

- **Part I** focuses on tracking the progress of the research and development work performed for each of the Use Cases in the second cycle and their current status, each of which has a dedicated section (Sections 3 to 6).
- **Part II** focuses on the Project's software integration process and infrastructure (Section 7), on the evolution of the testbed environment (Section 8), and on the software requirements verification tasks carried out per component (Section 9).

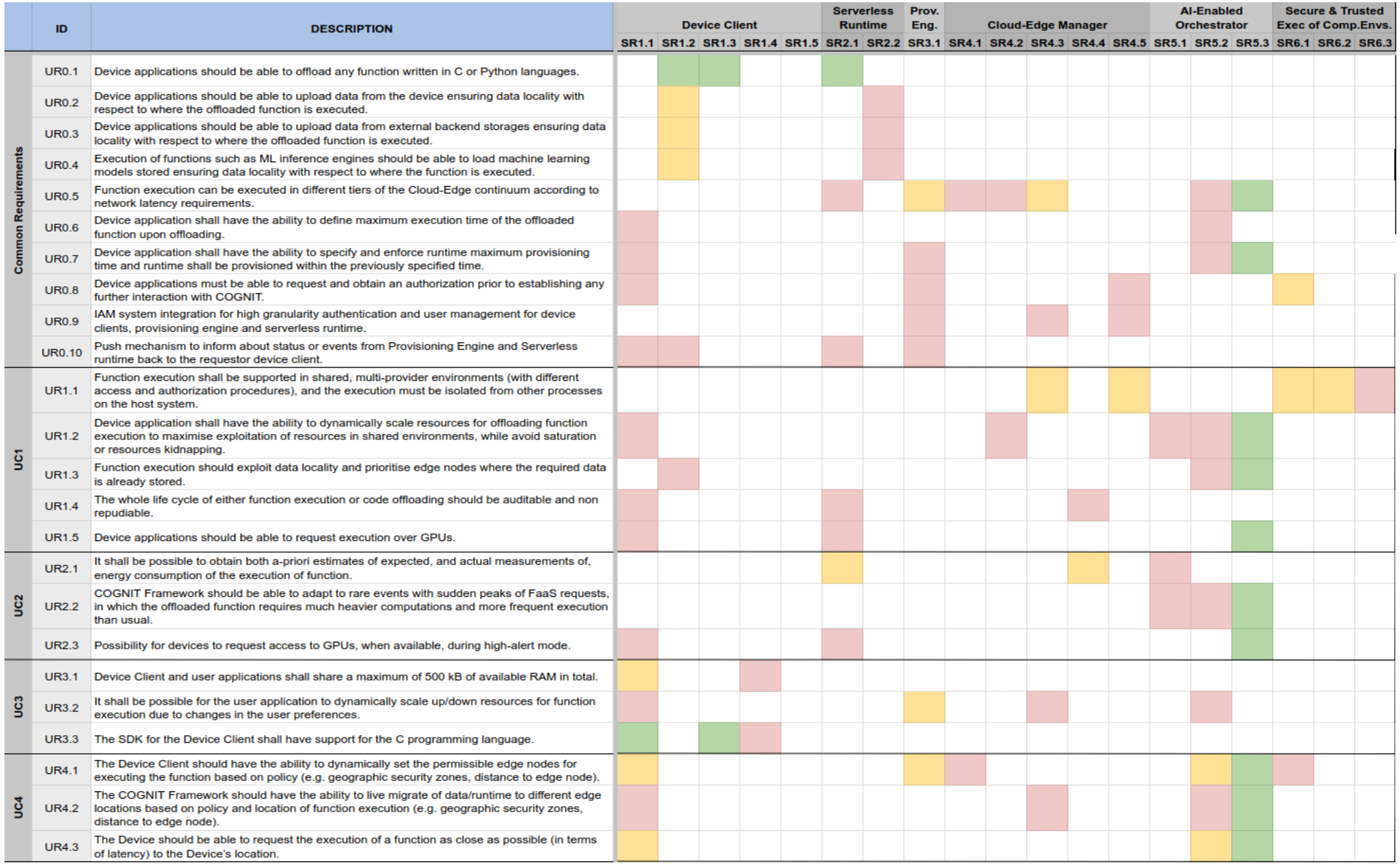
The document ends with a conclusion section.

PART I. Validation Use Cases

2. Overall Status

The table below shows the current status of each Software Requirement towards meeting its associated global and user requirements, following a simple colour code: for activities that have not started yet, for activities in progress, and for completed activities:

	ID	DESCRIPTION	Device Client					Serverless Runtime		Prov. Eng.	Cloud-Edge Manager					AI-Enabled Orchestrator			Secure & Trusted Exec of Comp.Envs.		
			SR1.1	SR1.2	SR1.3	SR1.4	SR1.5	SR2.1	SR2.2	SR3.1	SR4.1	SR4.2	SR4.3	SR4.4	SR4.5	SR5.1	SR5.2	SR5.3	SR6.1	SR6.2	SR6.3
Sovereignty	SOR0.1	The COGNIT Framework shall be able to leverage public, private, and self-hosted cloud and edge infrastructures hosted in the European Union.																			
	SOR0.2	The implementation of the COGNIT Architecture shall maximise the use of European open source technologies and frameworks.																			
	SOR0.3	The COGNIT Framework shall provide an abstraction layer that ensures workload portability seamlessly across different infrastructure providers.																			
	SOR0.4	Data handling by the COGNIT Framework shall be compliant with the GDPR.																			
Sustainability	SUR0.1	Sustainability performance needs to be measurable (e.g. energy profiles should be queryable and updatable for every feature/component within the framework), including energy sources (e.g. renewable, non-renewable) and energy consumption profiles (e.g. estimated power consumption).																			
	SUR0.2	Sustainability needs to be maximised to reduce environmental footprint by leveraging edge characteristics (e.g. by increasing the share of renewables, minimising battery use/size, using energy otherwise wasted, or scaling down active Runtimes).																			
	SUR0.3	The whole energy lifecycle should be taken into account in order to implement a circular economy, including e.g. energy availability and cost and hardware degradation.																			
Interoperability	IR0.1	Deployment of the COGNIT Framework and of its components should be as portable as possible across heterogeneous infrastructures or cloud/edge service providers (e.g. by using broadly-adopted virtualisation and container technologies).																			
	IR0.2	Preference should be given to expanding existing frameworks, tools, and open standards.																			
	IR0.3	The interfaces of the COGNIT Framework shall be documented in order to facilitate discovery of its features by third-parties.																			
Security	SER0.1	Communications inside COGNIT, and between the COGNIT environment and the outside (e.g. IoT devices) should be encrypted and signed using security mechanisms such as SSLv3.																			
	SER0.2	The COGNIT Framework should be built following security-by-design and Zero Trust practices.																			
	SER0.3	The implementation of the COGNIT Framework should be aligned with the latest legislative frameworks, such as the NIS2 Directive, the GDPR, and the future Cyber Resilience Act (CRA).																			
	SER0.4	Runtimes should be protected against threats by the enforcement of security controls such as secure defaults, vulnerability scans, intrusion and anomaly detection and continuous security assessment (the specific controls to be implemented will be determined by a risk analysis).																			
	SER0.5	Resources should be protected by an Identity and Access Management (IAM) system, implementing role based access control (RBAC), security zones, and support for a multi-tenant security model.																			
	SER0.6	Integrity of the offloaded functions needs to be guaranteed, including the function inputs and outputs (also during the live migration of FaaS Runtimes).																			



	ID	DESCRIPTION	Device Client					Serverless Runtime		Prov. Eng.	Cloud-Edge Manager					AI-Enabled Orchestrator			Secure & Trusted Exec of Comp.Envs.		
			SR1.1	SR1.2	SR1.3	SR1.4	SR1.5	SR2.1	SR2.2	SR3.1	SR4.1	SR4.2	SR4.3	SR4.4	SR4.5	SR5.1	SR5.2	SR5.3	SR6.1	SR6.2	SR6.3
Common Requirements	UR0.1	Device applications should be able to offload any function written in C or Python languages.																			
	UR0.2	Device applications should be able to upload data from the device ensuring data locality with respect to where the offloaded function is executed.																			
	UR0.3	Device applications should be able to upload data from external backend storages ensuring data locality with respect to where the offloaded function is executed.																			
	UR0.4	Execution of functions such as ML inference engines should be able to load machine learning models stored ensuring data locality with respect to where the function is executed.																			
	UR0.5	Function execution can be executed in different tiers of the Cloud-Edge continuum according to network latency requirements.																			
	UR0.6	Device application shall have the ability to define maximum execution time of the offloaded function upon offloading.																			
	UR0.7	Device application shall have the ability to specify and enforce runtime maximum provisioning time and runtime shall be provisioned within the previously specified time.																			
	UR0.8	Device applications must be able to request and obtain an authorization prior to establishing any further interaction with COGNIT.																			
	UR0.9	IAM system integration for high granularity authentication and user management for device clients, provisioning engine and serverless runtime.																			
	UR0.10	Push mechanism to inform about status or events from Provisioning Engine and Serverless runtime back to the requestor device client.																			
UC1	UR1.1	Function execution shall be supported in shared, multi-provider environments (with different access and authorization procedures), and the execution must be isolated from other processes on the host system.																			
	UR1.2	Device application shall have the ability to dynamically scale resources for offloading function execution to maximise exploitation of resources in shared environments, while avoid saturation or resources kidnapping.																			
	UR1.3	Function execution should exploit data locality and prioritise edge nodes where the required data is already stored.																			
	UR1.4	The whole life cycle of either function execution or code offloading should be auditable and non repudiable.																			
	UR1.5	Device applications should be able to request execution over GPUs.																			
UC2	UR2.1	It shall be possible to obtain both a-priori estimates of expected, and actual measurements of, energy consumption of the execution of function.																			
	UR2.2	COGNIT Framework should be able to adapt to rare events with sudden peaks of FaaS requests, in which the offloaded function requires much heavier computations and more frequent execution than usual.																			
	UR2.3	Possibility for devices to request access to GPUs, when available, during high-alert mode.																			
UC3	UR3.1	Device Client and user applications shall share a maximum of 500 kB of available RAM in total.																			
	UR3.2	It shall be possible for the user application to dynamically scale up/down resources for function execution due to changes in the user preferences.																			
	UR3.3	The SDK for the Device Client shall have support for the C programming language.																			
UC4	UR4.1	The Device Client should have the ability to dynamically set the permissible edge nodes for executing the function based on policy (e.g. geographic security zones, distance to edge node).																			
	UR4.2	The COGNIT Framework should have the ability to live migrate of data/runtime to different edge locations based on policy and location of function execution (e.g. geographic security zones, distance to edge node).																			
	UR4.3	The Device should be able to request the execution of a function as close as possible (in terms of latency) to the Device's location.																			

Table 2.1. Current status of each Software Requirement towards meeting its associated global and/or user requirements.

3. Use Case #1: Smart Cities

To demonstrate the capabilities of edge computing for improving the efficiency of public transportation and emergency services, this use case focuses on public bus and emergency vehicles prioritisation. A key element in this is the traffic signal priority mechanism, that allows special vehicles to get priority over other vehicles, when approaching an intersection. This is achieved by modifying Traffic Light Controller (TLC) phase timings when necessary, either by extending the green phase, reducing the red one or even forcing a new phase (in the case of emergency vehicles).

Nowadays, when a priority is requested by any of those special vehicles, the central traffic management system first needs to decide if it should be approved or not. This is typically based on variables like traffic congestion, bus schedules, etc., and also external systems that might get consulted. Once priority is granted, the objective is that the vehicle encounters a green light upon reaching the TLC. Therefore, the vehicle's arrival time must be calculated in real time, taking into account factors such as its position, speed, intersection layout, traffic status, etc.

Vehicle priority is currently resolved with custom proprietary solutions that lack standardisation and interoperability, and leaves no room for improvement with new features or variables to be considered. All the information needed to process traffic priority requests is held centrally in a Traffic Control Center (TCC), and managed by specialised traffic software suites like the one commercialised by ACISA, called Saturno.

In the context of this use case, an innovative approach will be adopted by using standardised V2X communications between vehicles and infrastructure (V2I) to support the priority service.

Cooperative ITS (C-ITS)[3] embrace a wide variety of communications-related applications and standards including V2X communication technology based on Dedicated Short Range Communications (DSRC), and documented under several European Norms (EN) and Technology Specifications (TS) published by ETSI[4]. 3GPP also included V2X functionalities starting from LTE release 14, based on cellular radio technology (LTE, 5G, 6G). In January 2020, ETSI with the collaboration of 5GAA published EN 303 613[5] defining the use of C-V2X as an access layer technology for Intelligent Transportation Systems (ITS) in the 5 GHz frequency band. This ensures that all vendors compliant with the standards will be interoperable, avoiding a vendor lock-in effect. In addition, cooperative V2X technology was designed to meet the needs of many safety use cases that can be added over time, with no additional investment in infrastructure.

Other than those communication standards, application layer V2X services are also being harmonised to ensure vendor interoperability. The current use case is covered in "*UNE-CEN ISO/TS 19091 Intelligent transport systems - Cooperative ITS - Using V2I and I2V communications for applications related to signalized intersections*".

To seamlessly integrate these advanced technologies into urban traffic management, additional requirements must be considered. Specifically, augmenting computation power

[3] https://www.etsi.org/technologies/automotive-intelligent-transport
[4] https://www.etsi.org/committee/1402-its
[5] https://www.etsi.org/deliver/etsi_en/303600_303699/303613/01.01.01_60/en_303613v010101p.pdf

at the edge is crucial to meet these data processing demands and low-latency applications effectively.

Mobility-Hub (henceforth M-HUB) is a new generation of TLC designed by ACISA, which incorporates edge computing capabilities, aiming to accelerate the adoption of Cooperative, Connected and Automated Mobility (CCAM) services in urban areas, while providing a common edge infrastructure to other mobility related systems. The M-HUBs come in two versions with different onboard computing capacity: the silver M-HUBs with limited capacity, and the gold M-HUBs that can act as edge computing nodes for offloading requests.

Considering that traffic infrastructures are owned by the city councils, enabling an open, standard-based far-edge platform will allow cities to deploy robust mobility and IoT services on top of existing traffic infrastructure. Other stakeholders include traffic/mobility departments inside city councils, and other local authorities, such as police, ambulance operators or fire-fighters. The objectives of the Smart City use case are:

- Implement a solution for Traffic/Transit Signal Priority (TSP)[6] service for public transport and emergency vehicles relying on a continuum infrastructure.
- The new approach must optimise delays at intersections, and emergency vehicles accidents through direct interaction between the V2X On-Board Unit (OBU), the Road Side Unit (RSU) and the integration with the M-HUB.
- Relay on the serverless platform COGNIT enables at the edge, to design new powerful applications optimising distributed on-prem resources.
- Seamlessly integrate far-edge clients (M-HUBs) with on-prem edge and cloud infrastructure, by means of function as a Services (FaaS) paradigm.

3.1 Current architecture and scenario

The Smart City use case will explore the use of V2X technology for public bus and emergency vehicles prioritisation at urban road intersections. In this scenario, a vehicle equipped with a V2X OBU sends a V2X priority request message, which is intercepted by an RSU, which then forwards it to its nearby M-HUB, installed at the intersection.

The objective of the pilot is to move this decision closer to the requesting vehicle in the far edge premises of a customer. M-HUB Silver, as clients of the COGNIT platform will make a remote FaaS request to M-HUB Gold installed in the far edge, to trigger the computational processes needed to grant or deny priority pass to the vehicle.

[6] https://www.emtracsystems.com/wp-content/uploads/2021/01/TSPHandbook10-20-05.pdf

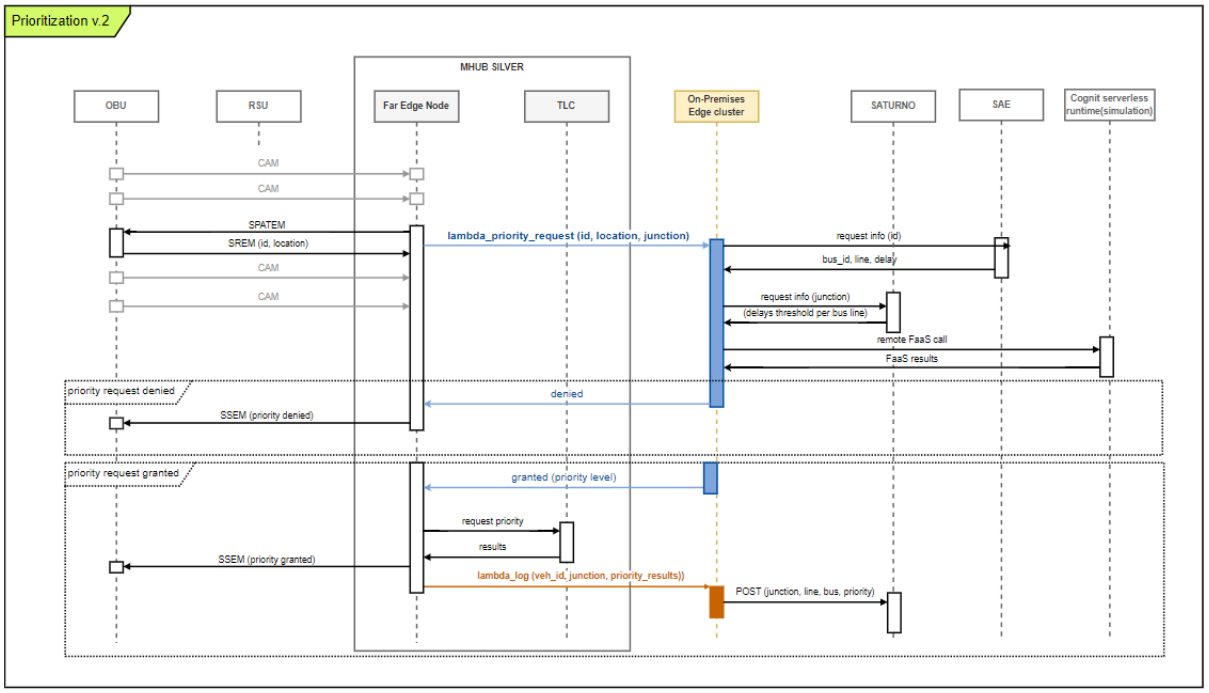


Figure 3.1. Sequence diagram showing different dependent systems

Vehicle communication is facilitated by an OBU equipped with V2X capabilities and a Global Navigation Satellite System (GNSS) receiver. The OBU communicates via radio with V2X (RSUs) installed at every intersection, and these RSUs are connected to the M-HUB Silver.

The following diagram illustrates the reference use case architecture. In this architecture, priority functionality is executed by serverless functions provisioned through the COGNIT Framework and deployed on edge cluster nodes, which are installed at the customer's far-edge premises.

A bus requests priority by sending a standardised V2X SREM message after receiving intersection data through SPATEM and MAPEM V2X messages, as defined in SAE J2735[7] Message Set Dictionary. The SREM message includes essential details such as the bus ID, ingress/egress lanes and, when available, bus delay information. The M-HUB offloads additional verifications to the serverless COGNIT Framework, which possesses all the necessary data to perform traffic simulations, and approve or deny the priority request.

Upon receiving a priority request from an authorised vehicle, the M-HUB may initiate a function call to the COGNIT FaaS service including additional contextual information to execute on demand traffic simulation by the SUMO[8]. This is some heavyweight functionality that wouldn't make sense to deploy on the client and will be leveraged using the remote FaaS. This requires the previous preparation in the COGNIT Framework's image of the simulation, the models of the junctions and some scripts to process the outcomes, so that these elements are already available when the remote call is made. The context

[7] https://www.sae.org/standards/content/j2735_202007/
[8] Simulation of Urban Mobility, an open source, highly portable, microscopic and continuous multi-modal traffic simulation package designed to handle large networks (https://eclipse.dev/sumo/).

information necessary to evaluate the priority requested, may be collected ideally by means of the DaaS service in the COGNIT Framework, although it may initially be stored in Saturno storage systems.

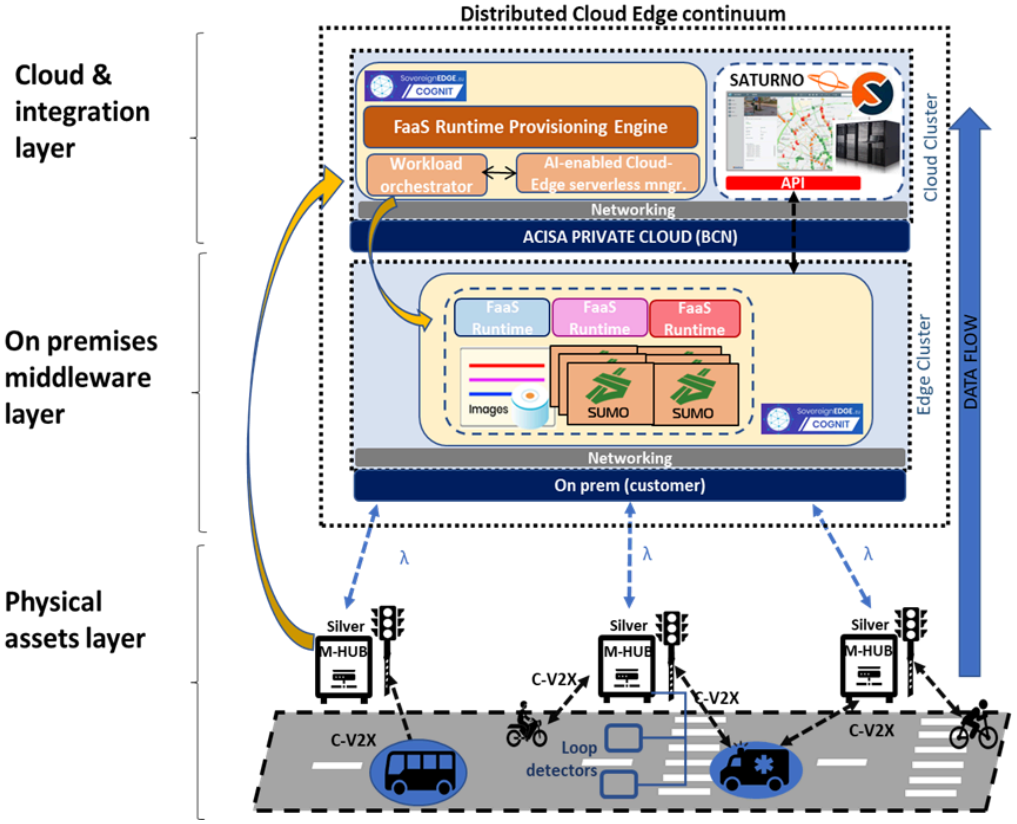


Figure 3.2: Smart City Use Case architecture

Once the priority analysis is done and communicated back to the M-HUB, it generates a response to be transmitted to the requesting vehicle. This response takes the form of another standardised V2X message known as the Signal Request Status Extended Message (SSEM), informing whether the priority has been conceded or rejected.

Our use case is focused on leveraging the capabilities of remote execution of complex simulation on the COGNIT Framework. This will provide our edge systems with valuable information to make decisions about how to deal with specific traffic situations.

To run complex simulations, this use case needs some specific functionality, so that the models can be executed quickly. Installing the required simulation tool, in our case SUMO, and loading the corresponding models, is a time consuming operation. Running these steps on demand would be an inefficient way of preparing the remote call.

For this reason, we opted for including the necessary tools and models into the image associated with the service, which gets triggered when a remote call is requested. That way the model is ready to be executed as quickly as possible on demand. In future releases the model might be a target for updates with recent traffic data stored in the DaaS component.

Digital Twin of urban road intersections

Every intersection within a city varies in terms of its topology, traffic model, M-HUB program, and the specific manoeuvres that a vehicle may request priority for, resulting in a high degree of complexity. The following diagram provides a few examples extracted from UNE/CEN ISO 19091[9], but there will be additional unique scenarios. Each intersection needs to be modelled with its particular differences in terms of topology, traffic flow, traffic demand, bus stop location, crosswalk, TLC programs, bus stops, etc.

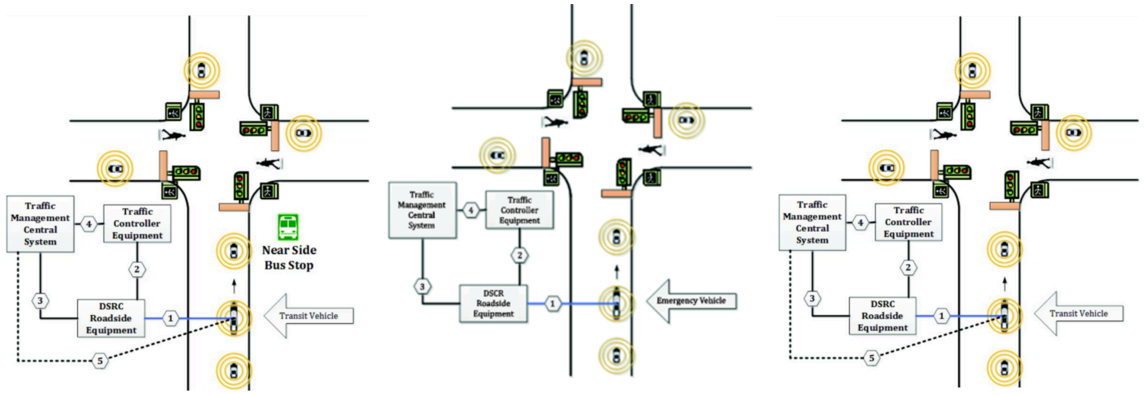


Figure 3.3. Examples of potential intersection scenarios in Transit Signal Priority (TSP). (Source UNE-CEN ISO/TS 19091)

Each intersection has its counterpart digital representation, a.k.a. Digital Shadow, holding updated historical data about its layout, current traffic status, subsystems status and alarms (M-HUB, detectors, others), etc. By incorporating the traffic simulator, the Digital Shadow evolves to a Digital Twin, who will act over the intersection traffic lights to grant or deny the priority based not only on current traffic status, but also on its simulated predictions.

3.2 Summary of research done during cycle M10-M15

In this section we will first justify the use of V2X technology to coordinate the elements in the traffic context, and the decision making at the edge will leverage the heavy lifting tasks processed by the COGNIT Framework. Next, we will get into the technical details of the work made during this phase, like:

- Remote call requests.
- Installation of the simulation tool into COGNIT Serverless Runtime image.
- Simulation execution.
- Parsing the results and returning them to the edge.

During this research cycle, ACISA also performed workload tests on the testbed, that are presented in Section 8 of this Deliverable.

[9] https://www.iso.org/standard/69897.html

Standards for C-ITS applications

The priority request functionality described above adheres to the principles outlined in UNE-CEN ISO/TS 19091, titled "Intelligent transport systems - Cooperative ITS - Using V2I and I2V communications for applications related to signalised intersections." This standard is built upon several specifications from ETSI and SAE. We would like to emphasise the following ones:

- ETSI TS 103 301 (V2.1.1)[10]: "Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Facilities layer protocols and communication requirements for infrastructure services"
- ETSI EN 302 665 (V1.1.1)[11]: "Intelligent Transport Systems (ITS); Communications Architecture".
- ETSI TS 102 894-2 (V1.3.1)[12]: "Intelligent Transport Systems (ITS); Users and applications requirements; Part 2: Applications and facilities layer common data dictionary".
- ETSI EN 302 636-4-1 (V1.4.1)[13]: "Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 4: Geographical addressing and forwarding for point-to-point and point-to-multipoint communications; Sub-part 1: Media-Independent Functionality".
- ETSI TS 103 097 (V1.4.1)[14]: "Intelligent Transport Systems (ITS); Security; Security header and certificate formats".
- ETSI EN 302 637-2[15]: "Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service".
- SAE J2735:2016[16], Dedicated Short Range Communications (DSRC) Message Set Dictionary {A, B, C}

Enabling V2X communication

The use case, among the multiple options, selected a bus priority and emergency vehicles (E-V) system based on C-V2X (cellular V2X and ETSI C-ITS-G5) connected vehicle technology, which will give a new perspective to the city's bus and E-V priority. All the solutions will be deployed within a continuous cloud-edge infrastructure and taking advantage of the serverless infrastructure provided by COGNIT.

V2X technologies encompass communication between vehicles (V2V), vehicles and infrastructure (V2I), vehicles and pedestrians (V2P), and vehicles and the network (V2N). V2X allows vehicles to exchange important information and data, enhancing safety, efficiency, and cooperating to exchange information and avoid hazardous situations, in an overall driving experience.

[10] https://www.etsi.org/deliver/etsi_ts/103300_103399/103301/02.01.01_60/ts_103301v020101p.pdf
[11] https://www.etsi.org/deliver/etsi_en/302600_302699/302665/01.01.01_60/en_302665v010101p.pdf
[12] https://www.etsi.org/deliver/etsi_ts/102800_102899/10289402/01.03.01_60/ts_10289402v010301p.pdf
[13] https://www.etsi.org/deliver/etsi_en/302600_302699/3026360401/01.04.01_30/en_3026360401v010401v.pdf
[14] https://www.etsi.org/deliver/etsi_ts/103000_103099/103097/01.04.01_60/ts_103097v010401p.pdf
[15] https://www.etsi.org/deliver/etsi_en/302600_302699/30263702/01.03.01_30/en_30263702v010301v.pdf
[16] https://www.sae.org/standards/content/j2735_201603/

This COGNIT use case will make use of C-V2X technology to enable real-time communications between public transport and emergency vehicles, such as ambulance, police or firefighters and traffic signals, allowing for a more dynamic and efficient traffic light prioritisation. With the use of C-V2X technology, public transport and emergency vehicles can communicate with the traffic light infrastructure and other vehicles in real time, enabling them to move more quickly and with greater reliability. V2X-based priority systems can reduce emissions and improve air quality, making public transportation more sustainable by reducing buses' time idling at intersections and providing a better QoS. In the case of E-V the improvement will impact directly on the safety of the E-V and its occupants.

The main reason why this use case needs the COGNIT Framework is that all the connected vehicles (not only bus and E-V) will generate in the near future a massive data throughput that will have to be processed in the edge with extremely low latency in order to take decisions that affect directly to the safety of the users. These decisions have to be taken by small computing devices (like TLCs) that will need to offload some parts of the code with FaaS serverless service that COGNIT will implement.

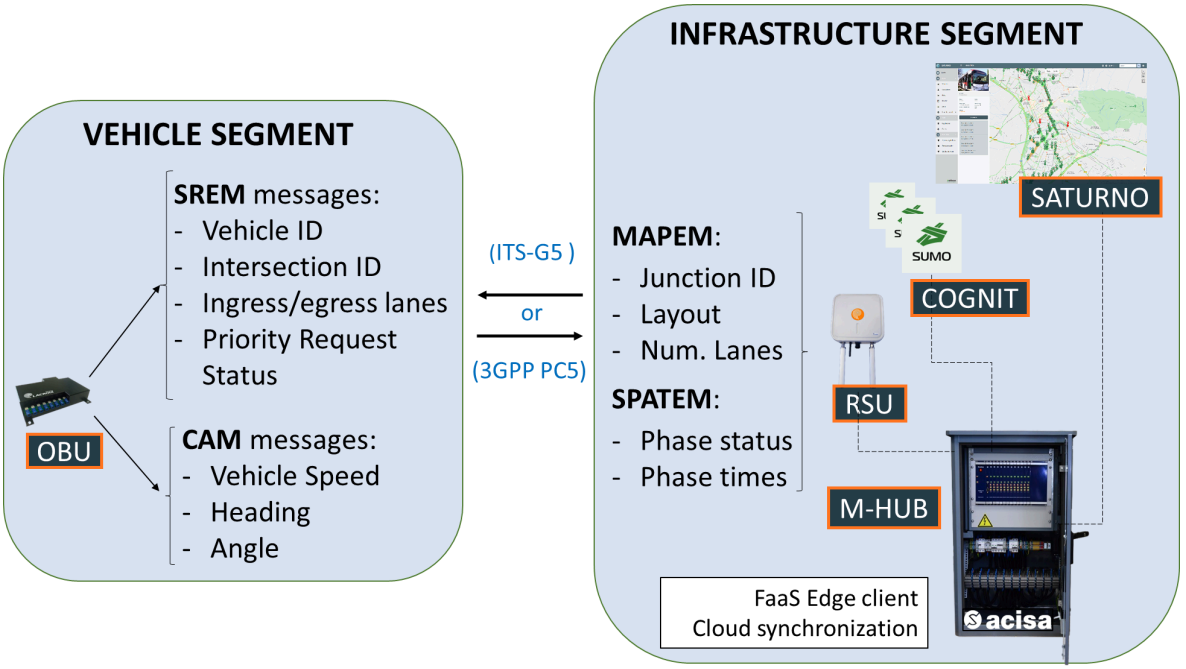


Figure 3.4. Main C-ITS messages used in TSP service.

ACISA wants to promote the progressive adoption of C-V2X technology through this Use Case that has the particularity that it doesn't require the direct participation of the car manufacturers or a massive adoption of the technology. However, in all locations where the BUS priority system will be installed, C-V2X coverage will be provided through the installation of RSUs, which can be used to enable new C-V2X use cases in the future.

The Use Case includes the installation of C-V2X modems in several public buses and Emergency Vehicle and all the necessary RSUs and TLCs in the test area and will connect with COGNIT infrastructure.

Remote call requests

A common task for all the use cases was to start deploying the tools that enable the edge to make remote calls to the COGNIT Framework, solve issues with dependencies and requirements, and connectivity issues.

This involved also getting familiar with the tools to manage the cloud environment (OpenNebula) in which the images would be configured and executed on demand.

Simulation tool installation

We will use SUMO to model and simulate several intersections in the City of Granada (Spain) and create workloads for FaaS Runtime Server. There are two main steps as prerequisites to create network scenarios and workloads for FaaS Runtime Server.

1) Install SUMO 1.2.0, Python, and Traffic Control Interface[17] (TraCI) Library to interface Python with SUMO.
2) Import Map data files of intersections inside the SUMO tool.

To install SUMO in our use case's virtual machine we followed this procedure:

```
Unset
# sumo requirements
sudo zypper install python-xml
sudo zypper install unzip

# Installing sumo
zypper addrepo
https://download.opensuse.org/repositories/home:behrisch/15.4/home:behrisch.
repo
zypper refresh
zypper install sumo
export SUMO_HOME=/usr/share/sumo

# Installing TRACI
pip install traci
```

After installing SUMO and its requirements, we need to select the intersections and import their map data inside the SUMO tool. For this purpose we use OSMWebWizard[18] which provides a graphical interface to zoom and select specific intersections and get their

[17] https://pypi.org/project/traci/
[18] https://sumo.dlr.de/docs/Tutorials/OSMWebWizard.html

map data. Indeed we can get snapshots of particular intersections and create real network scenarios using OSMWebWizard. Figure 3.5 shows an example of importing a scenario from intersections of the City of Granada and generating network scenarios using a Python script:

```
Unset
>> python osmWebWizard.py
```

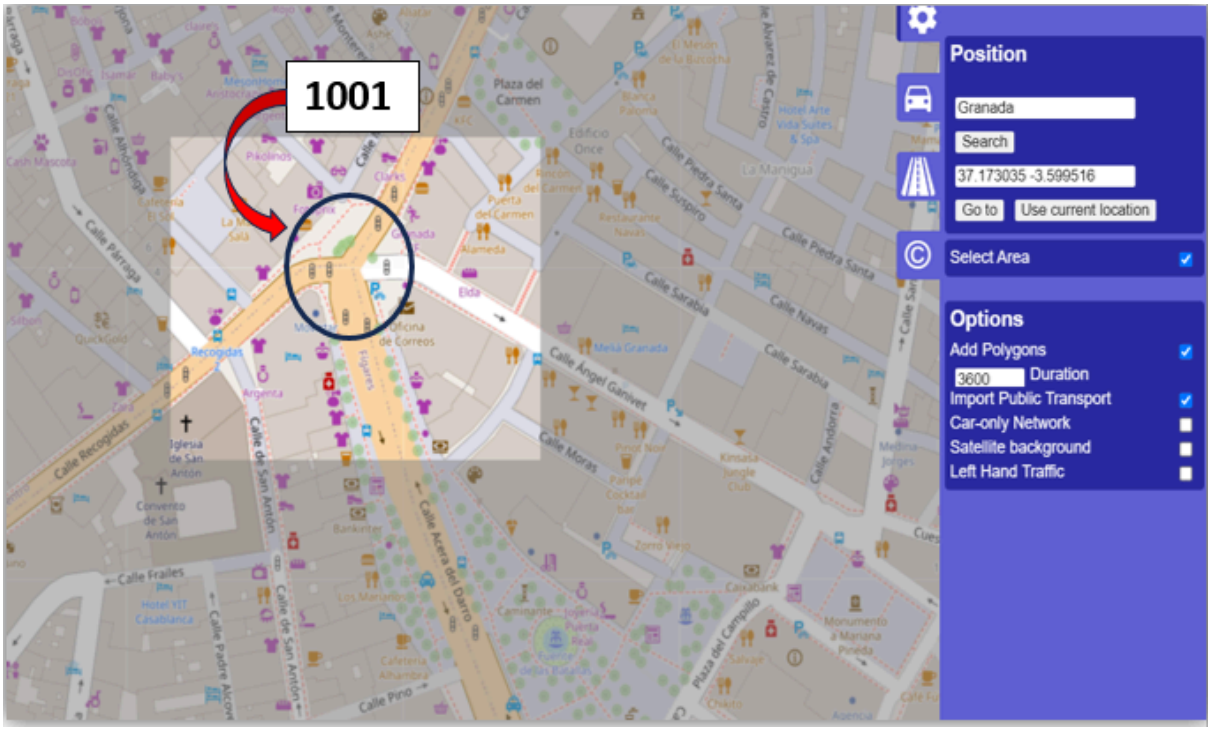


Figure 3.5. The snapshot[19] of intersection with id 1001 from City of Granada

Once we create network scenarios by importing the map data of intersections in the SUMO tool, we will need to generate a traffic demand for them. Accordingly, we can generate a random traffic demand for each intersection such as different vehicles, bicycles, pedestrians, etc. The following Python script shows an example of generating a random traffic demand on top of the network scenario:

```
Unset
>>python randomTrips.py -n osm.net.xml -r osm.rou.xml
```

[19] In this figure, map data from OpenStreetMap. In the models, the included data map is from OpenStreetMap, please see this copyright notice: https://www.openstreetmap.org/copyright

Model preparation

Our output models have the following data structure, which is prepared as workloads for the Serverless Runtime. We upload and store this workload in the reference VM of COGNIT Framework for our use case. In the next part, we will explain how to call and run the simulation for each intersection.

```
Unset
Granada/1001
├── build.bat
├── osm_bbox.osm.xml.gz
├── osm.bus.trips.xml
├── osm.netccfg
├── osm.net.xml
├── osm.passenger.trips.xml
├── osm.polycfg
├── osm.poly.xml
├── osm_ptlines.xml
├── osm_pt.rou.xml
├── osm_stops.add.xml
├── osm.sumocfg
├── osm.view.xml
├── run.bat
├── stopinfos.xml
├── trips.trips.xml
├── trips.trips.xml.errorlog
└── vehroutes.xml
```

Simulation execution

Having the directory structure outlined above, i.e. City, Junction code, which are parameters of the remote execution call we are doing this, on the VM spinned up by the COGNIT Framework:

```
Unset
cd /opt/smartcity_faas/model/$1/$2
sumo -c osm.sumocfg --emission-output output/my_emission_file.xml >
std_output.log 2>std_error.log
```

Parsing results

The results of the simulation can be exported to an output file which provides a huge amount of information. From that output, we need to parse the results and extract the information relevant to the Edge for decision making.

In general, we are extracting some information of interest, aggregating it and returning it to the edge. Here is a code snippet we prepared for our proof of concept (`parse_emission_gr.py`):

Python

```python
import xml.etree.ElementTree as ET
import argparse

parser = argparse.ArgumentParser(
   prog='parse_emission',
   description='Extract and process values of emissions.')

parser.add_argument('folder', help='Folder to search for model')
parser.add_argument('junction', help='Junction code one of 1001, 1002, 1003,
1004')
parser.add_argument('-n', '--names', nargs='+', default=[], help='List of
emission indicators to search for')

args = parser.parse_args()
folder = args.folder
junction_code = args.junction
targets = args.names
print("folder: {} ".format(folder))
print("junction_code: {}".format(junction_code))
print("output file: model/{}/{}/output/my_emission_file.xml".format(folder,
junction_code))
accum = {key: 0.0 for key in targets}
count = 0
for event, element in
ET.iterparse("model/{}/{}/output/my_emission_file.xml".format(folder,
junction_code), ["start"]):
   if element.tag == "vehicle":
       count += 1
       for key in element.attrib:
           if key in targets:
               accum[key] += float(element.attrib[key])

result = {key: accum[key] / count for key in targets}

print("result: {} - count: {}".format(result, count))
```

Returning results

One of the mechanisms available at COGNIT Framework to return results from the remote call is to return data as standard output, which will be returned to the Edge as a result object.

3.3 End-to-end integration with COGNIT Framework

As stated before, we rely on a simulation tool deployed into the image, alongside with models and scripts, so that it is already available to the remote FaaS call when the COGNIT runtime environment is triggered by a remote call to a service.

Our use case has the following pattern:

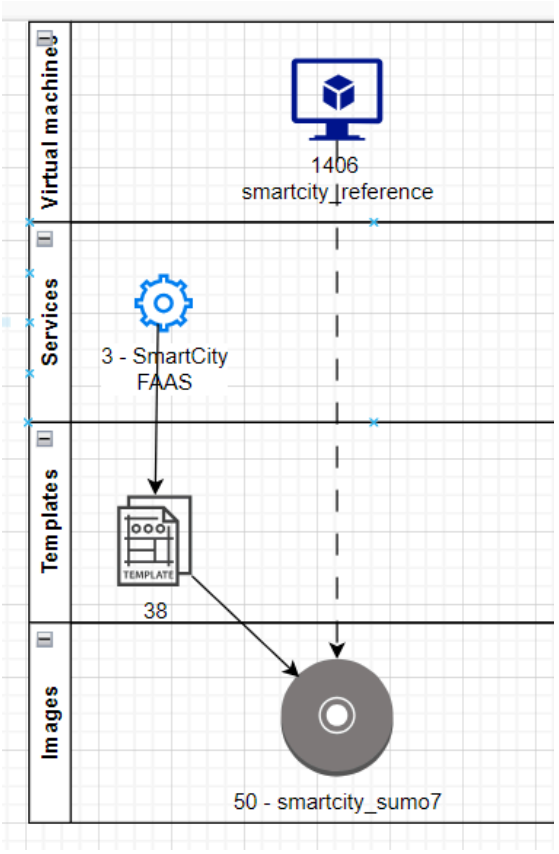


Figure 3.6. Generate customised image

Where we are installing products, models and scripts in a Virtual Machine (1406), and from that Virtual Machine, an image (50) is generated containing such elements. The COGNIT Service (3) includes a reference to a template (38), which refers as well to an image (50).

Figure 3.7 shows when a remote call is requested from the Edge, to the COGNIT Service (3), then a temporary virtual machine is spinned up: i.e. FAAS_0 executes the simulation previously uploaded in the image, and returns the result of the simulation to the Edge.

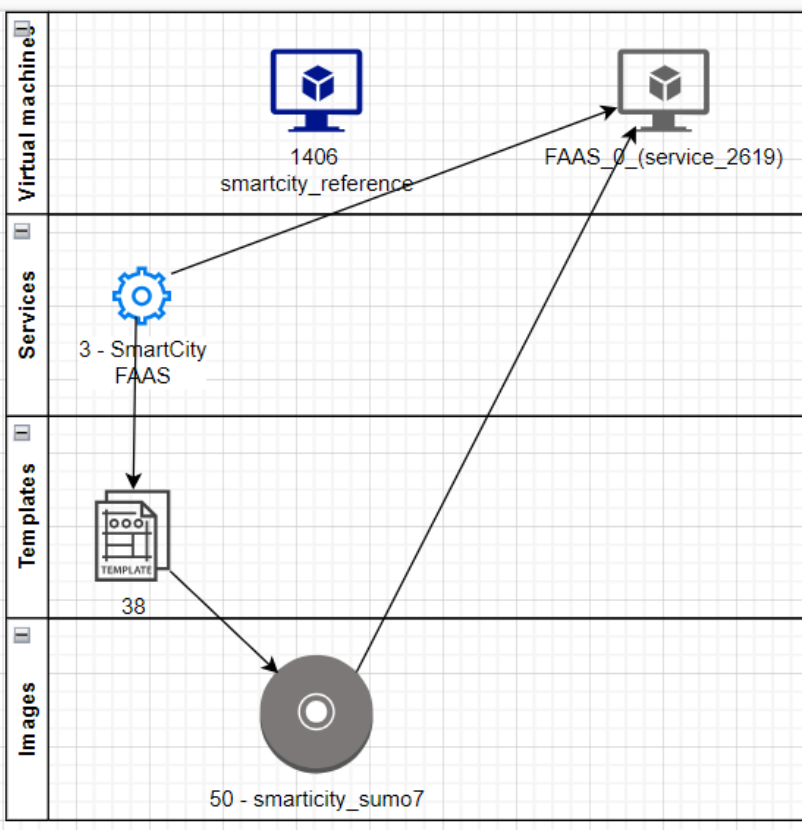


Figure 3.7. Starting instance to attend remote FaaS

On the Edge, we prepared some code, derived from the sample provided, to perform the remote execution call to the COGNIT Framework:

```python
def remote_call():
    cognit_logger = CognitLogger()
    # Configure the Serverless Runtime requeriments

    cognit_logger.info(">>>>>>>> Initiating remote call")
    sr_conf = ServerlessRuntimeConfig()
    sr_conf.name = "Example Serverless Runtime"
    sr_conf.scheduling_policies = [EnergySchedulingPolicy(50)]
    sr_conf.faas_flavour = "SmartCity"

    # Request the creation of the Serverless Runtime to the COGNIT
Provisioning Engine
    try:
        # Set the COGNIT Serverless Runtime instance based on 'cognit.yml'
config file
        # (Provisioning Engine address and port...)
        my_cognit_runtime =
ServerlessRuntimeContext(config_path="./examples/cognit.yml")
        # Perform the request of generating and assigning a Serverless
Runtime to this Serverless Runtime context.
```

```
        ret = my_cognit_runtime.create(sr_conf)
    except Exception as e:
        print("Error in config file content: {}".format(e))
        exit(1)

    # Wait until the runtime is ready

    # Checks the status of the request of creating the Serverless Runtime,
and sleeps 1 sec if still not available.
    while my_cognit_runtime.status != FaaSState.RUNNING:
        time.sleep(1)

    print("COGNIT Serverless Runtime ready!")

    # Example offloading a function call to the Serverless Runtime

    # call_sync sendsto execute sync.ly to the already assigned Serverless
Runtime.
    # First argument is the function, followed by the parameters to execute
it.
    folder = random.randint(1001, 1004)
    for i in range(100):
        result = my_cognit_runtime.call_sync(cli, f"Granada {folder} -n CO2
CO NOx")
        cognit_logger.info(f">>>>>>>> result {result}")
        if result.ret_code == ExecReturnCode.ERROR:
            print(i)
            time.sleep(1)
        else:
            break

    cognit_logger.info(">>>>>>>> Offloaded function result {result.res}")

    # This sends a request to delete this COGNIT context.
    if result.ret_code == ExecReturnCode.SUCCESS:
        my_cognit_runtime.delete()
        cognit_logger.info("COGNIT Serverless Runtime deleted!")
        return None
    else:
        cognit_logger.error("COGNIT EXECUTION FAILED")
        return Exception
```

The remote_call() function, in addition to preparing the configuration for the remote call, is checking that the expected Virtual Machine is running and when ready, it makes a remote call to an existing script, previously stored in the image, alongside the model:

```python
def cli(indicators: str):
   import subprocess

   return subprocess.run(["/opt/smartcity_faas/call_sumo_gr.sh
{}".format(indicators)], capture_output=True, text=True,
shell=True).stdout.strip("\n")
```

This script, partially shown above, runs the simulation, and calls the parser to extract the results from output of the model:

```
#!/bin/bash
# Parameters: $1: junction code
cd /opt/smartcity_faas/model/$1/$2
sumo -c osm.sumocfg --emission-output output/my_emission_file.xml >
std_output.log 2>std_error.log
cd ../../..
echo "parameters $@"
python parse_emission_gr.py "$@"
```

The `parse_emission_gr.py` has been shown above in a code snippet in chapter “Parsing results” and will not be repeated here.

3.4 Summary of technology developments

- As we need a complex process executed by the COGNIT Framework, instead of sending functions for remote execution, we opted for calling remote scripts already deployed in the COGNIT Framework image to execute a SUMO simulation model.
- The result of the simulation contains a lot of information in a complex format that needs to be parsed, aggregated, and some values are returned as a result of the remote FaaS execution. This is processed by the COGNIT Framework to send as a result only the relevant data.
- Integrating remote FaaS calls with workload performance tests. To start measuring the capacity of the allocated resources to process concurrent tasks, we added some testing class that allows us to generate in a convenient way concurrent tasks in a delimited period of time.

3.5 Plans for the next research cycle M16-M21

For the third Research & Innovation Cycle, this Use Case aims to research on the following topics:

1. **Automatic Deployment of UC1 Dependencies**:
    - Integration of dependencies into the images used for remote calls to Function as a Service (FaaS).
2. **Framework Development and Real-World Integration**:
    - Our next step involves developing and integrating the framework in an actual operational environment. Here are the critical aspects:
        - **FaaS Launch by M-HUBs**: Automatically launch a FaaS instance on each M-HUB once a priority is requested by Public Bus or Emergency Vehicle.
        - **Simulation Parameters**: Include relevant simulation parameters in the call to the FaaS. These parameters guide decision-making and optimise priority assignment.
        - **Results Handling**: Retrieve the results from the COGNIT FaaS execution from both the client M-HUB to finish priority process, and from Saturno to archive results for historical and audit evidence.
3. **Send relevant data to the Data as a Service (DaaS) component**:
4. **Enhancing Priority Strategies with V2X Data**:
    - Leverage Vehicle-to-Everything (V2X) communication data to enhance priority strategies within the M-HUB.
    - By incorporating real-time V2X information (such as vehicle positions, traffic flow, and emergency alerts), we can optimise traffic signal timings and prioritise E-V requests effectively.
5. **Deploy a new COGNIT node into ACISA premises:**
    - This deployment will show the possibilities of using COGNIT Framework on private data-centres, enabling the use of serverless FaaS technology within the edge infrastructure, contributing to the objective of technology sovereignty.
6. **Real Data for AI Model Training**:
    - As the project progresses, we will provide relevant operational data to train provisioning AI models used by COGNIT.
7. **Granada Project: The Ultimate TestBed**:
    - The Granada Project serves as the definitive testbed for the Smart Cities use case in COGNIT. Its specifications include:
        - **Intersections**: A network of 38 intersections, where traffic management strategies will be evaluated. Each intersection is equipped with a RSU and a M-HUB.
        - **Equipped Buses**: Outfit 18 public buses with C-V2X OBUs to participate in the real-world testing.

4. Use Case #2: Wildfire Detection

The wildfire detection use case explores the use of IoT technologies for fire detection in forests. The early detection of wildfire is of utmost importance to obtain a timely intervention from civil protection and fire fighters to minimise damages in terms of forestry resources and human lives. However, developing and deploying a reliable sensor network in the forest is challenging due to the strict power constraints and the lack of a strong connectivity, as well as for the cost of maintenance in remote areas. Furthermore, a false alarm resulting in an unnecessary intervention leads also to an increase of costs.

4.1 Current architecture and scenario

For these reasons, a low power device is being designed that collects the data from multiple sensors to assess the presence of fire. The collected parameters are:

- Carbon dioxide concentration in air (ppm).
- Ozone concentration in air (ppm).
- Particulate matter PM10, PM2.5, PM1 (µg/m3).
- Air temperature (°C).
- Air relative humidity (%).
- Flame detection sensor with 200 m range.
- Camera for image acquisition.

Sensors installed in forests can be used for the first detection of fire flames and to collect useful metrics for wildfire risk assessment. On the other hand, the trigger of one or more sensors is not sufficient to detect a fire, while some of them could be triggered when a wildfire has already massively spread, depending on the wind or other environmental conditions. For this reason, the camera should confirm the presence of fire and together with the other sensor data help track the spread of a wildfire. While for most of the sensors it is sufficient to set a threshold, image recognition requires a suitable machine learning algorithm that is likely to exceed power and computing resource capabilities of IoT devices. While the possibility for the device to run the ML algorithm is still under assessment, it makes sense to try offloading it considering the microprocessor constraints and the application requirements. The devices will be equipped with a NB-IoT and/or 4G connectivity to exchange information with the external applications.

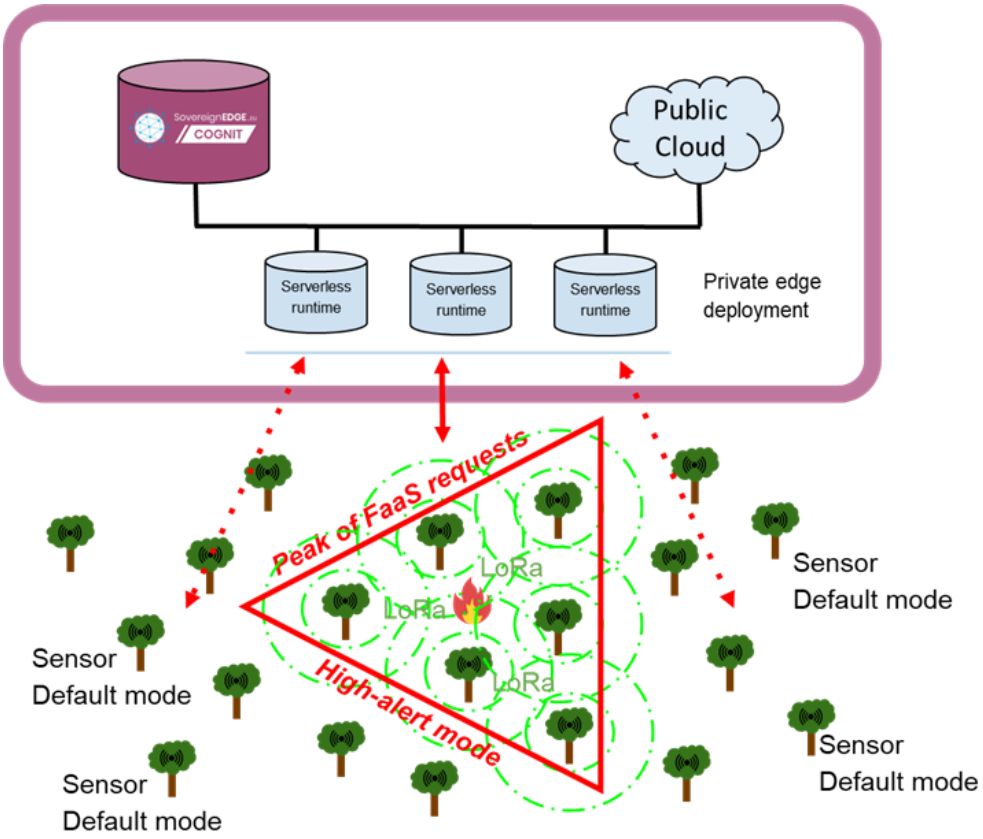


Figure 4.1. Schematization of wildfire use case scenario

The device routine consists of monitoring at fixed time intervals its surroundings with all the sensors except the camera to save energy. In most cases no flames will be detected and the device will send the collected data to an external database for further environmental analysis. This behaviour represents the “Default mode”. When the collected parameters may indicate the presence of a flame, the “High alert mode” is triggered.

In this mode, the camera is turned on and the measurement frequency is increased to one measurement per minute. In “High alert mode”, a FaaS request is sent to COGNIT for each measurement to run the image recognition algorithm and confirm the presence of a flame. In case flame detection is confirmed, the data is sent to civil protection. The FaaS requests are performed until a given number of requests return a negative answer.

When a device enters the “High alert mode” and every time a FaaS request returns a positive result, a distress signal is sent to the nearby devices using LoRa technology. All devices within reach wake up and collect data, including camera images, and trigger a FaaS request to COGNIT. If the request returns a positive result (a flame has been detected) the device enters “High alert mode”, otherwise it returns to sleep.

A device exits the “High alert mode” if a certain number of requests return a negative result and will make no further request for a given period.

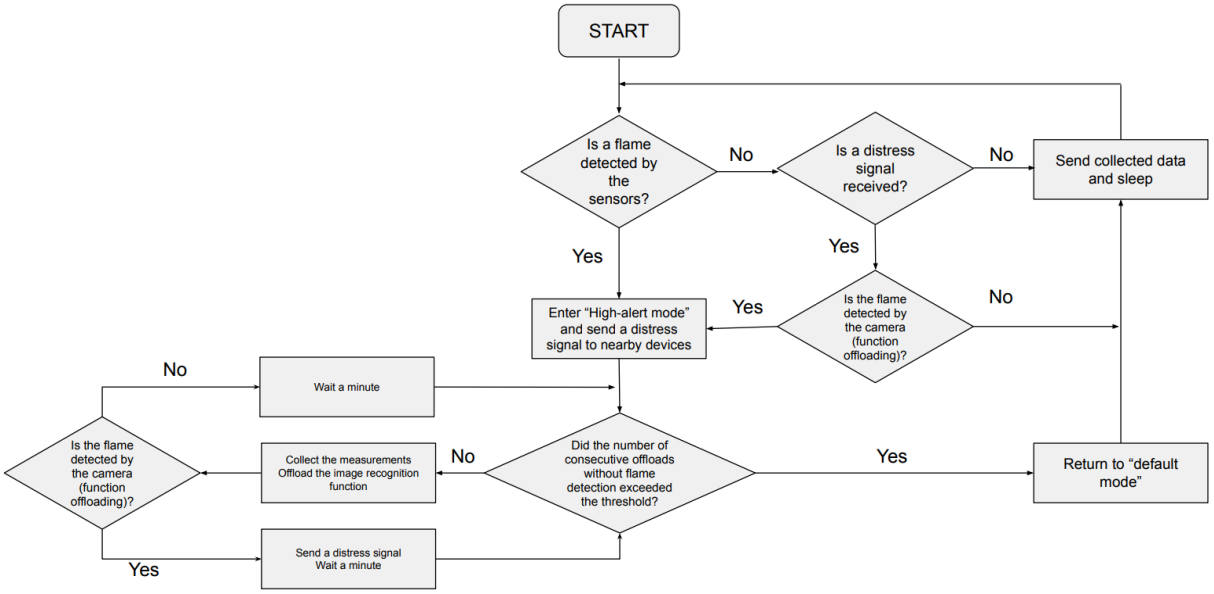


Figure 4.2. Flow chart of wildfire sensor network logic.

The described architecture has been slightly polished compared to the previous cycle and allows to keep tracking the wildfire while minimising the power consumption and the impact of false detection. On the other hand, when there is presence of fire, the number of devices involved and the subsequent number of FaaS requests can rapidly increase asking for a fast response to the sudden peak of requests. Moreover, while a wildfire risk assessment can be performed, the actual emergency can hardly be forecasted, requiring a high scalability and adaptability.

For all these reasons, allocating a given number of resources based on the maximum capabilities requested by the application would be inefficient causing a waste of computational resources for most of the time. However, all the resources required must be timely available when needed. The sudden increase of requests could be partially mitigated by the COGNIT Framework knowing that a flame detected by a sensor will wake up other sensors and allocating some resources for the following requests. Finally, a priority policy could be implemented to assign high priority on resource access when a wildfire is detected.

The machine learning model, along with all the required libraries for the function execution, will be saved into the image associated with the service. This approach aims to speed up the execution and spare bandwidth.

The devices will offload the function given the captured image as argument.

Our use case aims to test the capability of the COGNIT Framework to respond to an unforeseen sudden peak of requests and to provide hardware resources such as GPUs when required by the application.

4.2 Summary of research done during cycle M10-M15

During the M10-M15 cycle the effort was focused on finding a suitable ML algorithm for forest fire recognition and learning how to make it compatible with the COGNIT Framework.

Machine learning algorithm selection and testing

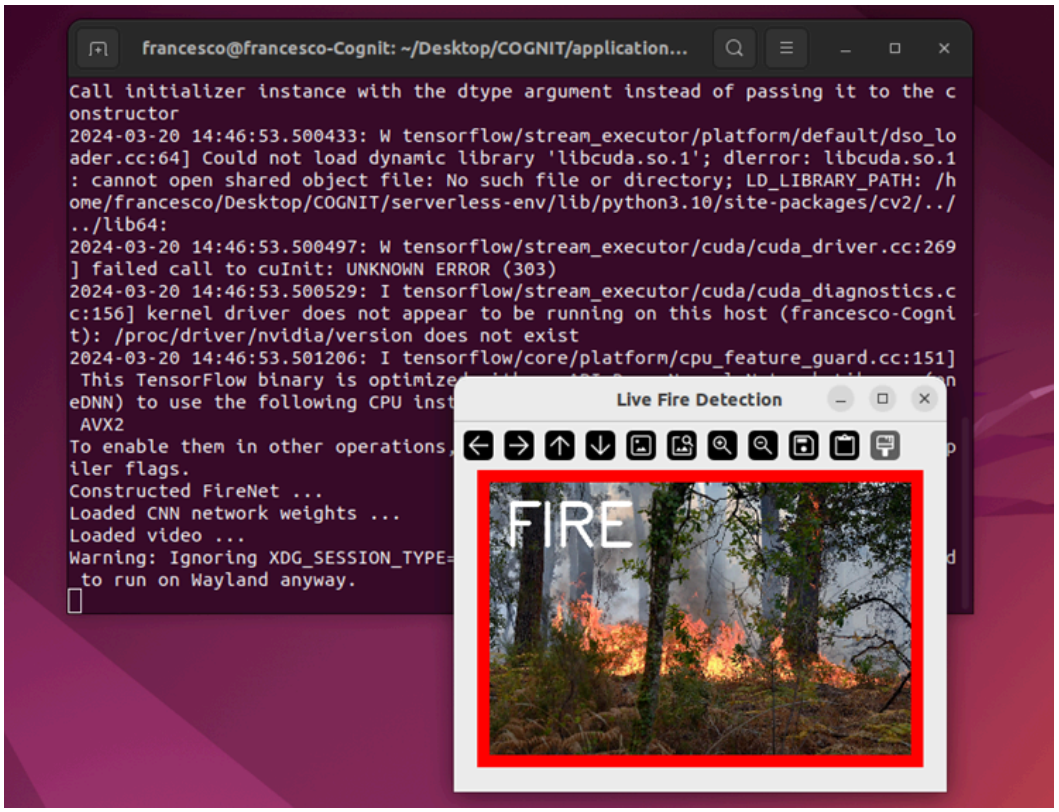


Figure 4.3. Execution of the fire image recognition function with visual rendering.

The programming language chosen for the development of the application is Python. It has a wide choice of ML algorithms and can partially be run on microprocessors as Micropython. A thorough research has been conducted regarding pre-trained fire image recognition machine learning models with an MIT licence and good performances. A model natively compatible with TensorFlow Lite based on an inceptionV1 Convolutional Neural Network was identified [1,2]. The model is specialised in fire recognition during videos and is also defined by different functions. It has been reduced and adapted to run in a single function and return a true or false depending on the detection of a fire in each image provided to the function as an argument.

Moreover, the required libraries (dependencies) were reduced to the bare minimum. The software was tested on an Ubuntu virtual machine on different images to assess the performances. Also, compatibility issues with existing libraries were fixed to make it run, selecting the most recent compatible versions of the required libraries (tensorflow==2.8.0, Pillow==9.0.1, protobuf==3.12.4). Providing the model with static images requires them to

be in png format due to the use of “opencv-python” so a piece of code to convert most common image formats into png was created. The imported image is not a VideoCapture object, so it is saved and reuploaded in the correct format after being passed to the function. The directory structure used for the application is illustrated in Figure 4.4:

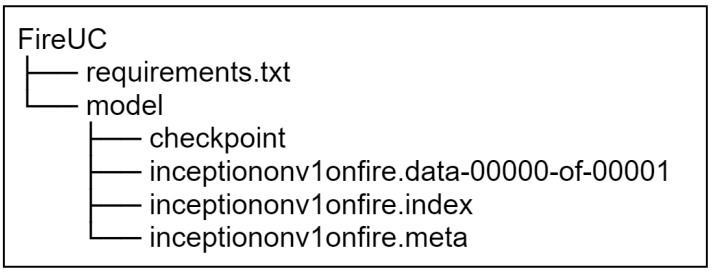


Figure 4.4. Directory structure of uploaded files.

The “model” folder contains the pre-trained neural network model in TensorFlow format.

COGNIT Framework testing and function deployment

One of the main goals of this cycle was to offload a function to the COGNIT Framework. The access to IPv6 network was obtained through the wireguard configuration provided by OpenNebula on Ubuntu. The COGNIT Framework capabilities have been explored both from a device client point of view and using the cloud environment provided by OpenNebula for creating and managing the image and virtual machines. Due to the dependencies and the model needed by the application, a dedicated image for the Use Case was created. The virtual machine was created, accessed through SSH and the dependencies were installed. Some issues with the standard virtual machine configuration arose during the installation of the libraries due to the size of TensorFlow library and the lack of some dependencies. After some debugging the TensorFlow issue was solved by increasing the default VM memory size to 3.072 GB and the disk size to 8 GB.

After the successful installation of TensorFlow a problem with the absence of libGL library during the installation of opencv-python as well as the lack of support of SSE4.1 instructions from the default CPU was found. The second issue was solved by OpenNebula updating the VM template CPU mode to “host-passthrough”. The successful installation of the libGL library on openSUSE distribution was performed through the following commands:

```
Unset
zypper install Mesa-libGL1
zypper install libgthread-2_0-0
```

After the successful installation of all the required libraries, the folder containing the model and some images was saved on the VM and the fire image recognition analysis was successfully performed on the VM so the fire images where removed and the disk image

was saved as “FireUC” and associated with the template called by “Nature” flavour (Nature FaaS Runtime f2bd3f5 IPv6). However, the attempt to offload the function from the outside resulted in an error. It turned out it was necessary to set a higher timeout for the REQ_TIMEOUT variable in the Serverless Runtime client because the tensorflow module requires some time to be loaded but the timeout value will be increased by default in the next release. Other minor changes have been performed on the code to correctly offload the function. Now the function can be successfully offloaded.

Device development

Two microcontroller families to test the offload of the function and the image acquisition feature have been selected: esp32 and STM32L4. Both of them have great computing power but the STM32 has lower power consumption while the esp32 is already widely used for image recognition tasks. The chosen board to perform the assessment of esp32 capabilities is a commercial one (esp32-CAM) that is already equipped with an OV2640 camera while a brand new board has been internally designed to test the STM32L433RCT6. It already has all the interfaces required by the sensors that should be read by the TTFire device including a SPI connector for the camera. Moreover, our SIM7020 board can be interfaced with it providing NB-IoT connectivity. The two solutions can also be combined together if needed. Both solutions are currently being tested.

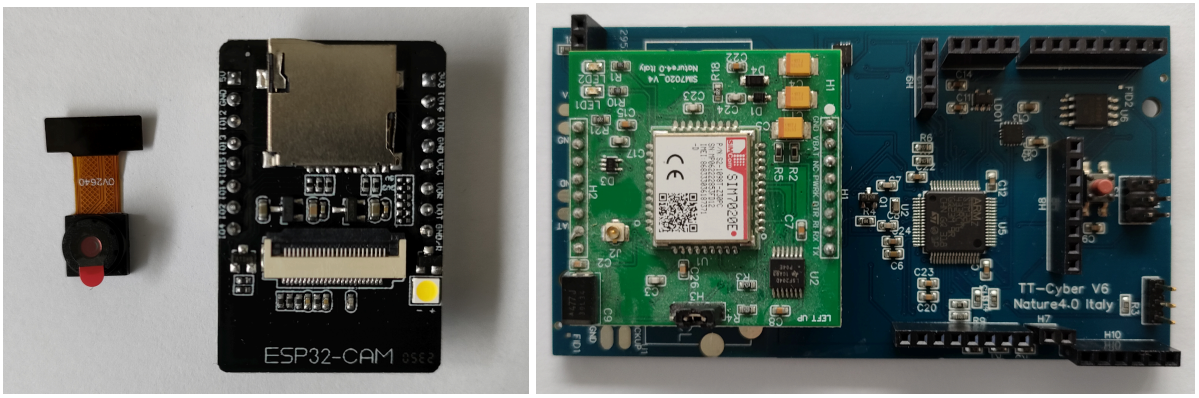


Figure 4.5. Boards for testing the offload of the function from a microcontroller

4.3 End-to-end integration with COGNIT Framework

To run the fire recognition function, a suitable image has been prepared and linked to the “Nature” template. The folder containing all the necessary files and dependencies is placed in the root directory and its folders tree is shown in Figure 4.4. The requirements are the following:

- A VM of with at least 3.072 GB memory and a disk size of 8 GB;
- A (v)CPU capable of executing SSE4.1 instructions (obtained through the “host-passthrough” setting for the VM CPU);

- Mesa-libGL1 and libgthread-2_0-0 to install and use Python opencv-contrib-python library;
- The following Python libraries: opencv-contrib-python, tensorflow==2.8.0, tflearn, Pillow==9.0.1 and protobuf==3.12.4;
- The folder with the model to run.

The client application is composed of two parts, the call to the COGNIT Framework and the function to offload. The call to the COGNIT Framework is shown below.

Python

```
import sys
import time

sys.path.append(".")

from cognit import (
    EnergySchedulingPolicy,
    FaaSState,
    ServerlessRuntimeConfig,
    ServerlessRuntimeContext,
)

# Configure the Serverless Runtime requirements
sr_conf = ServerlessRuntimeConfig()
sr_conf.name = "Example Serverless Runtime"
sr_conf.scheduling_policies = [EnergySchedulingPolicy(50)]

# Configure "Nature" flavour to use FireUC image
sr_conf.faas_flavour = "Nature"

# Request the creation of the Serverless Runtime to the COGNIT Provisioning
Engine

try:
# Set the COGNIT Serverless Runtime instance based on 'cognit.yml' config file
       my_cognit_runtime = ServerlessRuntimeContext(config_path="cognit.yml")

# Perform the request of generating and assigning a Serverless Runtime to this
Serverless Runtime context.
       ret = my_cognit_runtime.create(sr_conf)
except Exception as e:
       print("Error in config file content: {}".format(e))
       exit(1)

# Checks the status of the request of creating the Serverless Runtime, and sleeps
1 sec if still not available.
while my_cognit_runtime.status != FaaSState.RUNNING:
       time.sleep(1)

time.sleep(5)

print("COGNIT Serverless Runtime ready!")

import cv2

# select the image to analyze and import it
image = "Test_images/fire3.png"
image = cv2.imread(image)
```

```
# perform the function offloading
result = my_cognit_runtime.call_sync(fire_presence_detection, image)

print("Offloaded function result", result)
print("Only result", result.res)

# This sends a request to delete this COGNIT context.
my_cognit_runtime.delete()

print("COGNIT Serverless Runtime deleted!")
```

The offloaded function is a machine learning algorithm based on an inceptionV1 convolutional neural network pre-trained on fire images. The model is uploaded from a folder already present in the FireUC image. The function returns true or false depending on the detection of a fire in the image provided.

Python

```
def fire_presence_detection(im):

	try:
		import cv2
		import tflearn

		from tflearn.layers.core import input_data, dropout,
		fully_connected

		from tflearn.layers.conv import conv_2d, max_pool_2d, avg_pool_2d,
		global_avg_pool

		from tflearn.layers.normalization import
		local_response_normalization, batch_normalization

		from tflearn.layers.merge_ops import merge

		from tflearn.layers.estimator import regression

		import os
		import tensorflow as tf

		tf.compat.v1.reset_default_graph()

		cv2.imwrite('.image.png', im)
		image = cv2.VideoCapture('.image.png')

		rows = 224
		cols = 224

		x = rows
		y = cols
		enable_batch_norm=True

		# Build network as per architecture in [Dunnings/Breckon, 2018]

		network = input_data(shape=[None, y, x, 3])

		conv1_7_7 = conv_2d(network, 64, 5, strides=2, activation='relu',
		name='conv1_7_7_s2')
```

```
pool1_3_3 = max_pool_2d(conv1_7_7, 3,strides=2)
pool1_3_3 = local_response_normalization(pool1_3_3)

conv2_3_3_reduce = conv_2d(pool1_3_3, 64,1,
activation='relu',name='conv2_3_3_reduce')

conv2_3_3 = conv_2d(conv2_3_3_reduce, 128,3, activation='relu',
name='conv2_3_3')

conv2_3_3 = local_response_normalization(conv2_3_3)
pool2_3_3 = max_pool_2d(conv2_3_3, kernel_size=3, strides=2,
name='pool2_3_3_s2')

inception_3a_1_1 = conv_2d(pool2_3_3, 64, 1, activation='relu',
name='inception_3a_1_1')

inception_3a_3_3_reduce = conv_2d(pool2_3_3, 96,1,
activation='relu', name='inception_3a_3_3_reduce')

inception_3a_3_3 = conv_2d(inception_3a_3_3_reduce, 128,
filter_size=3, activation='relu', name ='inception_3a_3_3')

inception_3a_5_5_reduce = conv_2d (pool2_3_3,16, filter_size=1,
activation='relu',name ='inception_3a_5_5_reduce')

inception_3a_5_5 = conv_2d(inception_3a_5_5_reduce, 32,
filter_size=5, activation='relu', name='inception_3a_5_5')

inception_3a_pool = max_pool_2d(pool2_3_3, kernel_size=3,
strides=1, )

inception_3a_pool_1_1 = conv_2d(inception_3a_pool, 32,
filter_size=1, activation='relu', name='inception_3a_pool_1_1')

# merge the inception_3a__
inception_3a_output = merge([inception_3a_1_1, inception_3a_3_3,
inception_3a_5_5, inception_3a_pool_1_1], mode='concat', axis=3)

inception_3b_1_1 = conv_2d(inception_3a_output, 128, filter_size=1,
activation='relu', name='inception_3b_1_1' )

inception_3b_3_3_reduce = conv_2d(inception_3a_output, 128,
filter_size=1, activation='relu', name='inception_3b_3_3_reduce')

inception_3b_3_3 = conv_2d(inception_3b_3_3_reduce, 192,
filter_size=3,  activation='relu',name='inception_3b_3_3')

inception_3b_5_5_reduce = conv_2d(inception_3a_output, 32,
filter_size=1, activation='relu', name='inception_3b_5_5_reduce')

inception_3b_5_5 = conv_2d(inception_3b_5_5_reduce, 96,
filter_size=5,  name='inception_3b_5_5')

inception_3b_pool = max_pool_2d(inception_3a_output, kernel_size=3,
strides=1,  name='inception_3b_pool')

inception_3b_pool_1_1 = conv_2d(inception_3b_pool, 64,
filter_size=1, activation='relu', name='inception_3b_pool_1_1')

#merge the inception_3b_*
```

```
            inception_3b_output = merge([inception_3b_1_1, inception_3b_3_3,
            inception_3b_5_5, inception_3b_pool_1_1], mode='concat', axis=3,
            name='inception_3b_output')

            pool3_3_3 = max_pool_2d(inception_3b_output, kernel_size=3,
            strides=2, name='pool3_3_3')

            inception_4a_1_1 = conv_2d(pool3_3_3, 192, filter_size=1,
            activation='relu', name='inception_4a_1_1')

            inception_4a_3_3_reduce = conv_2d(pool3_3_3, 96, filter_size=1,
            activation='relu', name='inception_4a_3_3_reduce')

            inception_4a_3_3 = conv_2d(inception_4a_3_3_reduce, 208,
            filter_size=3,  activation='relu', name='inception_4a_3_3')
            inception_4a_5_5_reduce = conv_2d(pool3_3_3, 16, filter_size=1,
            activation='relu', name='inception_4a_5_5_reduce')

            inception_4a_5_5 = conv_2d(inception_4a_5_5_reduce, 48,
            filter_size=5,  activation='relu', name='inception_4a_5_5')

            inception_4a_pool = max_pool_2d(pool3_3_3, kernel_size=3,
            strides=1,  name='inception_4a_pool')

            inception_4a_pool_1_1 = conv_2d(inception_4a_pool, 64,
            filter_size=1, activation='relu', name='inception_4a_pool_1_1')

            inception_4a_output = merge([inception_4a_1_1, inception_4a_3_3,
            inception_4a_5_5, inception_4a_pool_1_1], mode='concat', axis=3,
            name='inception_4a_output')

            pool5_7_7 = avg_pool_2d(inception_4a_output, kernel_size=5,
            strides=1)

            network = loss

            model = tflearn.DNN(network, checkpoint_path='inceptiononv1onfire',
            max_checkpoints=1, tensorboard_verbose=2)

            model.load(os.path.join("/root/FireUC/model",
            "inceptiononv1onfire"), weights_only=True)

#-----------------------------------MODEL LOAD-------------------------------

            # load video file from first command line argument

            ret, frame = image.read()


            # re-size image to network input size and perform prediction

            small_frame = cv2.resize(frame, (rows, cols), cv2.INTER_AREA)

            # perform prediction on the image frame which is:
            # - an image (tensor) of dimension 224 x 224 x 3
            # - a 3 channel colour image with channel ordering BGR (not RGB)
            # - un-normalised (i.e. pixel range going into network is 0->255)

            output = model.predict([small_frame])

            # label image based on prediction
```

```
            # equiv. to 0.5 threshold in [Dunnings / Breckon, 2018],
            [Samarth/Bhowmik/Breckon, 2019] test code

            if round(output[0][0]) == 1:
                  return 1
            else:
                  return 0
      except:
            return 2
```

4.4 Summary of technology developments

The use-case-2 repository which is part of the COGNIT Project's GitHub summarises the requirements to offload the image recognition algorithm and contains an example of the algorithm itself along with some images for testing purposes.

4.5 Plans for the next research cycle M16-M21

Having the function to offload to the COGNIT Framework ready, the next cycle will aim to test some scenarios and deploy them into a first prototype.

In particular, two possible simplified models could be used to test the sensor network and the COGNIT Framework behaviour as well as interaction during a wildfire spread: one that can be called "statistical" and another one that can be called "spatial".

The "statistical" model will simulate a sensor network homogeneously distributed in the forest. In the simulation, when a fire starts and is detected by the first sensor, an image recognition function is offloaded and n devices are woken up resulting in as many functions offloading. The devices have a certain probability to offload an image containing a fire. If it is the case, the "High alert mode" is triggered and further n devices will wake up. The new devices will have even less probability to pick an image with a fire in it. With each iteration, the probability of each device to detect a fire increases until they are all woken up.

Figure 4.6. Representation of the "spatial" simulation of the sensor network

The “spatial” model consists of a grid in which the sensors are placed. Each sensor has a defined field of view. In the simulation, a fire is started at a random point and when it is detected by a sensor it enters the “High alert mode”. This then sends a distress signal to the nearby devices. Every following cycle, the flame spreads and the devices that have the flame in their field of view enter the “High alert mode”. This solution takes into account the distribution of the sensors in an area.

During the next cycle the goals will be:

- Test the function offloading of a sensor network using at least one of the two methods described above.
- Develop a first functioning prototype and try to offload the function from it (C or micropython version).

Additional References

1. Experimentally defined Convolutional Neural Network Architecture Variants for Non-temporal Real-time Fire Detection (Dunnings, Breckon), In Proc. International Conference on Image Processing, IEEE, 2018

2. Experimental Exploration of Compact Convolutional Neural Network Architectures for Non-temporal Real-time Fire Detection (Samarth, Bhowmik, Breckon), In Proc. International Conference on Machine Learning Applications, IEEE, 2019

5. Use Case #3: Energy

This Use Case is exploring the scenario of using smart electricity meters to optimise green local energy usage in a household context, in which energy consumers are also energy producers (*prosumers*). The Use Case is developed and tested by meeting its goals in Poland. In most Polish locations, the current energy system is carbon-intensive and centralised, which means that there are only a few locations in the country where energy is generated. As a consequence electricity must be transmitted over long distances through the transmission and distribution network resulting in high losses. In addition bottlenecks and disruptions in the network have the potential to affect huge areas and populations.

The energy industry of the future will be based on distributed systems, relying on renewable energy sources (RESs) and energy storage solutions. This highly distributed model of the energy network features many small producers of energy, aiming to reduce costs, risks, and intensity of greenhouse gas emissions, as well as to eliminate transmission energy losses because energy is produced and consumed locally. To make this a reality, there is a strong need to manage energy consumption as well as its production to optimise usage of local energy. Electricity meters, already at the interface between the building and the power grid, are ideally positioned to manage such distributed smart energy systems.

5.1 Current architecture and scenario

In this scenario, the smart electricity meters run a number of user applications to manage important appliances and energy assets installed (from grid topology perspective) behind the meter, adjusting and optimising operation in real time, according to user preferences. These appliances and assets include energy storages, photovoltaic installations, heat pumps, electrical vehicle chargers, and electric floor heating. By empowering electricity meters with apps, connected to services equipped with advanced decision-making algorithms (and eventually pre-trained AI models), they turn into highly personalised Energy Assistants. Running user apps is possible thanks to the Phoenix-RTOS (Real-Time Operating System), which offers all needed mechanisms for effective partitioning and as a result to reach full safety in separation of the partition for user apps and Distribution System Operator (DSO) partition, including the legally relevant Measuring Instruments Directive (directive 2014/32/UE) part.

Ultimately, this approach leads to cost savings because of more effective usage of energy and lowering overall demand for coal energy, as an example.

This Use Case demonstrates the capabilities of edge computing to support the ongoing transformation of the energy sector. It is moving from a hierarchical, centralised structure towards a more decentralised and distributed way of managing energy assets and networks.

Main aspects of demonstration and validation are to test performance of the COGNIT Framework:

- in case when there are large amounts of concurrent requests in the area of the same local edge node;

- in case of dynamic changes in Serverless Runtime performance requirements due to changes in user preferences.

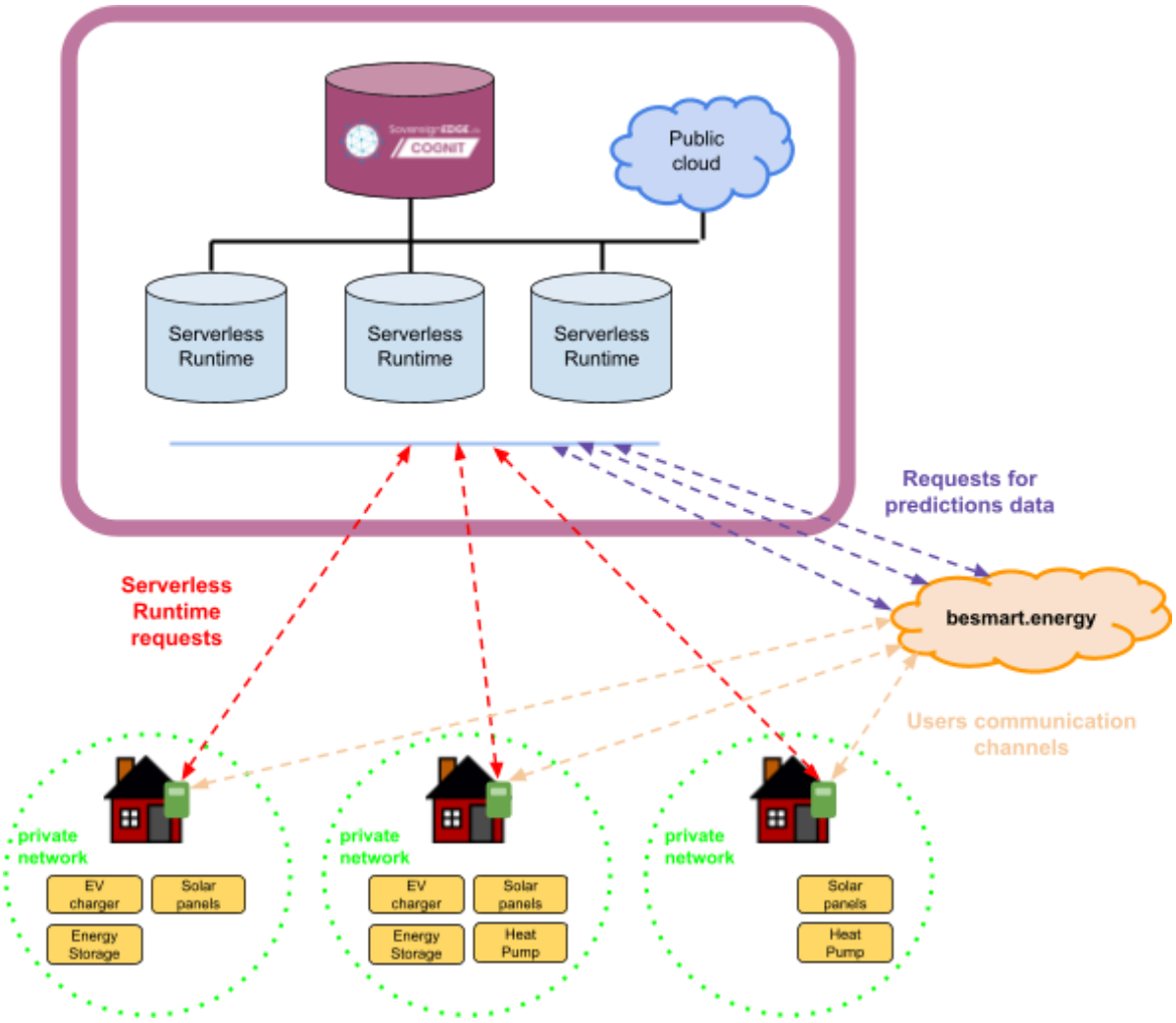


Figure 5.1. Architecture for the Energy Use Case

The Use Case involves the following system components:

- ***besmart.energy*** – a cloud platform for smart energy systems.
- **Devices Controllers** – actuating devices.
- **Electricity Meter-COGNIT Client** – current data provisioner, manager of actuators.
- **Provisioning Engine** – Serverless Runtime manager.
- **Serverless Runtime** – consists of a FaaS Runtime providing computing resources. environment and a Data Service providing data storage for FaaS Runtime usage.
- **User** – stakeholder.

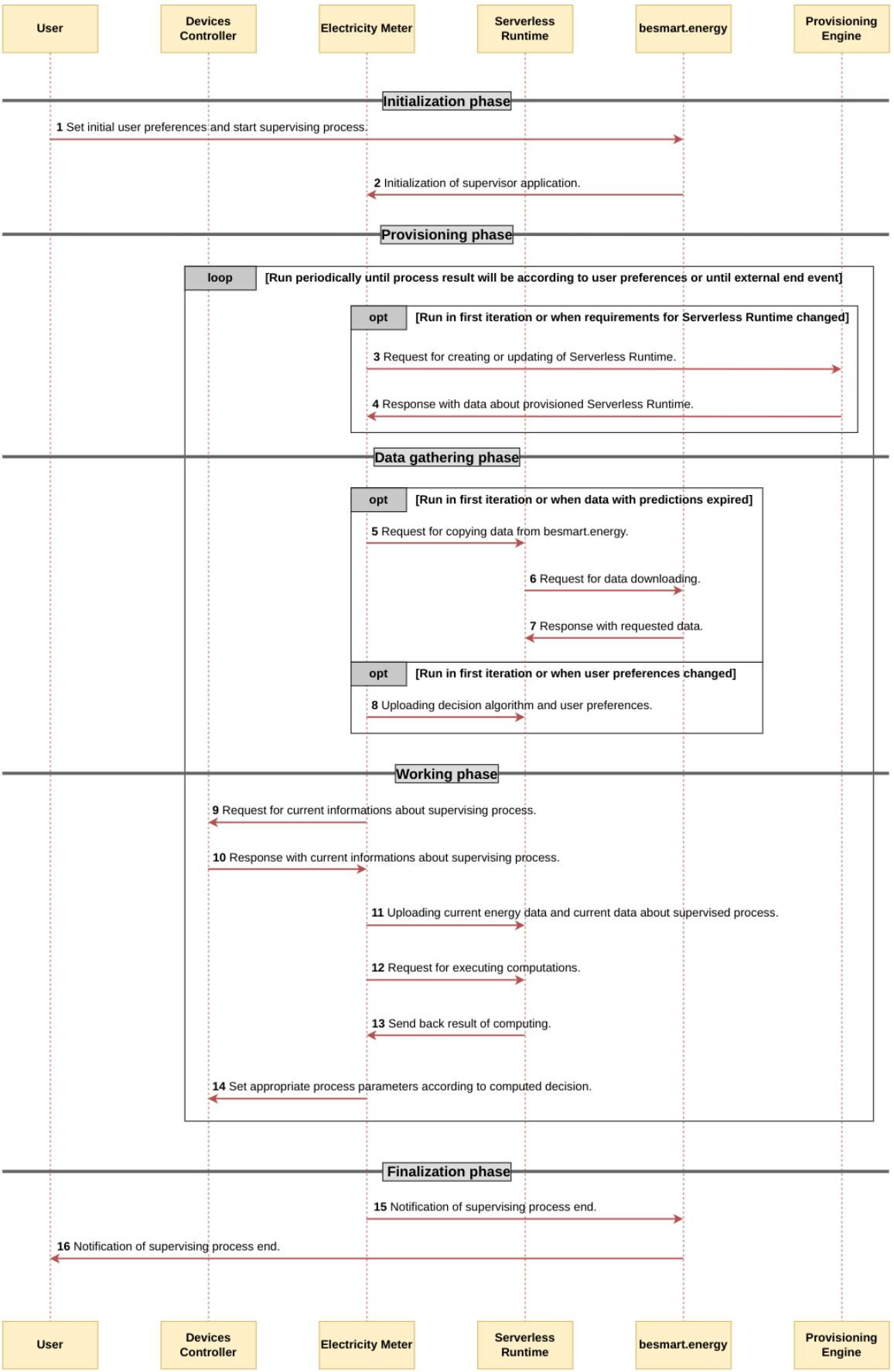


Figure 5.2. Process diagram for the Energy Use Case.

5.2 Summary of research done during cycle M10-M15

During the last research cycle M10-M15 the work has been focused on following topics:

- Data collection and processing from the test environment.
- Understanding end devices in terms of energy storage and heating system physics.
- Decision algorithm for maximising use of locally produced energy.
- Simulation of smart energy meters (SEM) and energetically important appliances.

Data collection and processing

The aim of this use case is to explore the use of smart energy meters to optimise green local energy usage in household context. For better understanding of the issue, there was a modern electricity meter installed in one household in Poland, which is a real representative for reference scenario and will be used as a testground in next cycles. It takes part in energy exchange not only as a consumer, but also a producer of renewable energy, becoming a prosumer. This private network includes a photovoltaic array connected with a battery energy storage, an electric vehicle charger and electric heating controlled using a wireless system.

Data from this environment are sent to the TStorage database and presented in besmart.energy platform, so they can be accessed from everywhere. The electricity meter collects data about energy consumed from the grid and energy returned to the grid from local network every 15 minutes. It also gives the opportunity to connect to appliances controllers and communicate with them about parameters/settings and available data. In this cycle we achieved successful integration of the following metrics for regular data queries:

- Energy returned to local grid by PV inverter.
- Storage state of charge.
- Maximum limit of storage charging and discharging power.
- Temperature from each sensor.
- Temperature setting in each heating zone.
- State of switch of each heating device.

Unfortunately, the integration with EV charger controller using the Open Charge Point Protocol (OCPP) protocol was not established, so it would be one of the goals for the next cycle.

Analysing the data, comparing and working with it to create a decision algorithm based on it required prior pre-processing, which included few steps:

1. Unification of units – The energy meter counts the total energy consumed/returned, expressed in kilowatt-hours, hence it is necessary to reproduce the instantaneous values in kW. Similarly, it is needed to transform the percentage level of storage charge into its current capacity in kW.
2. Unification of time base/frequency - By querying device controllers, only current values can be obtained, therefore the data gained this way is at irregular intervals.

To compare data from different sources, they must refer to the same time steps. Using aggregation and interpolation, it is possible to obtain values for points with regular frequency, e.g. every 15 minutes as for energy consumed/returned signals or every 1 hour, as this is the standard unit of time for energy transactions.

3. Unification of time range – Because of required research and work, collecting of different data started at different time. There were also some technical issues regarding energy meter, so it was not sending data for some time. Moreover, there may always be some errors on any device that make reading impossible. Therefore, the ranges of the raw series did not overlap 100%, so it was necessary to supplement the missing data. In the case of single points, interpolation can be used, but for longer periods of time, more complex methods like machine learning models trained on available data are preferable.

This part of research work also included getting familiar with the communication protocols (like e.g. Modbus TCP) that the mentioned appliances use. The gained knowledge as well as the current data-collecting scripts will be beneficial at the stage of deploying the whole infrastructure of Energy Use Case at the testing environment.

End devices

Understanding how the most important devices in a household function from the perspective of the local energy network is essential to optimise their management. Implementing the correct model is important for designing a decision algorithm and creating an environment simulator. To understand current modelling approaches, a literature review was conducted.

Energy storage

Main topic of research was energy storage systems, especially battery solutions – method of operation, parameters, modelling and charging/discharging management. Particular attention was paid to papers where storage was used in households to increase the consumption of energy generated by PV panels, so similar to our testing scenario. We formulated a rather simple battery storage bidding model with constant charging power limit, as it is suitable for integration in optimisation problems. State of energy can be described by the following relation:

$$soe_t = soe_{t-1} + (\Delta t * P_{ch,t} * \eta - \Delta t * P_{dis,t}) * 100 / C_{max} \quad (5.1)$$

subject to

$$P_{ch,t} \leq \alpha_{ch,t} * P_{max} \quad (5.2)$$

$$P_{dis,t} \leq \alpha_{dis,t} * P_{max} \quad (5.3)$$

$$soe_t \leq C_{max} \quad (5.4)$$

where:

- soe_t is state of energy (expressed in percentages);
- η the energy efficiency of charging storage;
- C_{max} the battery energy capacity (Wh);
- P_{max} nominal charging/discharging power (W);
- $P_{ch,\,t}$ (W) and $P_{dis,\,t}$ (W) charging and discharging powers (both always assumed positive); and
- $\alpha_{ch,\,t}$, $\alpha_{dis,\,t}$ - limitations of those powers.

It is important not to discharge storage below a certain level, not only because of energy reservation in the event of failure and unavailability of energy from the grid, but also considering its longevity. The smaller the depth of discharge, the greater the number of cycles before it needs to be replaced. In addition, the process of optimising storage management should include limiting the number of charge/discharge cycles preferably to one per day.

This basic model is the most commonly used battery storage model in the power system economics literature. However, there is an opportunity to better reflect the battery charging characteristic, which consists of two distinctive parts, constant-current and constant-voltage. In the next cycle, we are going to introduce a linear model with the charging limit, which reduces for state of energy levels above a certain value.

Heating

Heating constitutes the largest share of energy consumption in a household. The heating source determines how electricity is converted into raising the temperature inside a home. In the case of our testing environment, we are dealing with mats and electric heaters, which can only work with full power. They can be switched on or off by controller based on current temperature in the room measured by sensor and referenced temperature, set by the user. It can be assumed that the air in the room heats up directly as a result of the device operating at a specific power and efficiency for a given time.

The more challenging issue is heat loss through the floor, walls, doors, windows, roof and other surfaces that separate indoors from outdoors. Each element of the building has its own heat transfer coefficient value related to the material it is made of, which measures how much heat it allows to pass through, but one has to also consider its area and thickness. There is also another component of ventilation losses, which occurs when hot air inside the building is replaced by colder outside air through ventilation or infiltration. It is impossible to create a model that would take into account all physical dependencies, so we decided to first approximate the losses using one parameter.

The temperature in the room can be determined as follows:

$$T_t = E_t / h \quad (5.5)$$

$$E_t = E_{t-1} + s_t * \eta * P_{heat} * \Delta t + \Theta * (T_{out,t} - T_{t-1}) * \Delta t \quad (5.6)$$

where

- T_t is temperature inside room (K);
- E_t energy accumulated in the room (J);
- h heat capacity (J/K);
- s_t state of heating device ($\epsilon \{0, 1\}$);
- η heating efficiency;
- Θ heat loss parameter (W/K); and
- $T_{out,t}$ temperature outside (K).

Decision algorithm

In simplified scenario we focused on optimisation of self-consumption of single prosumer. From financial and ecological point of view, local consumption is the most sufficient. Our task is to decide on settings of end devices controllers for the next period of time (default 1 hour) to minimise energy consumed from grid, but taking into consideration preferences of users. It is preferred we only use green energy produced from PV or collected previously in battery energy storage. For the first iteration, decision algorithm makes decision based on current values (without any predictions of future). It is assumed it knows ideal models of home heating and storage charging/discharging, so can compute energy distributed between devices during next time step. Scheme of decision algorithm is presented below in Figure 5.3.

The Algorithm gets information about all settings of controllers and the current values of:

- energy drawn from the grid in the previous time step,
- energy returned to the grid in the previous time step,
- energy produced by PV array in the previous time step,
- outdoor temperature,
- current and last storage state of charge,
- status of heating devices,
- measured temperature per room,
- preferred temperature by user per room.

Based on the energy transmitted between the local and global grid, energy production and changes in storage capacity, the energy consumed by the household is calculated. Knowledge about the state of heating device switches over the last time step and their power allows for determining the energy consumed for heating. By subtracting this value from the total energy consumed, we obtain the energy required by other appliances, which we cannot control and must provide. It is assumed that this energy demand from the previous step can be a good approximation of unmodifiable energy consumption in the

step under consideration. The algorithm determines the energy flow needed to fulfil this request in order:

1. Primarily, the energy generated by PV array is used.
2. In the event of a shortage of energy directly from a renewable source, the availability of energy stored is checked.
3. If we are still unable to meet the energy demand, it has to be drawn from the grid.

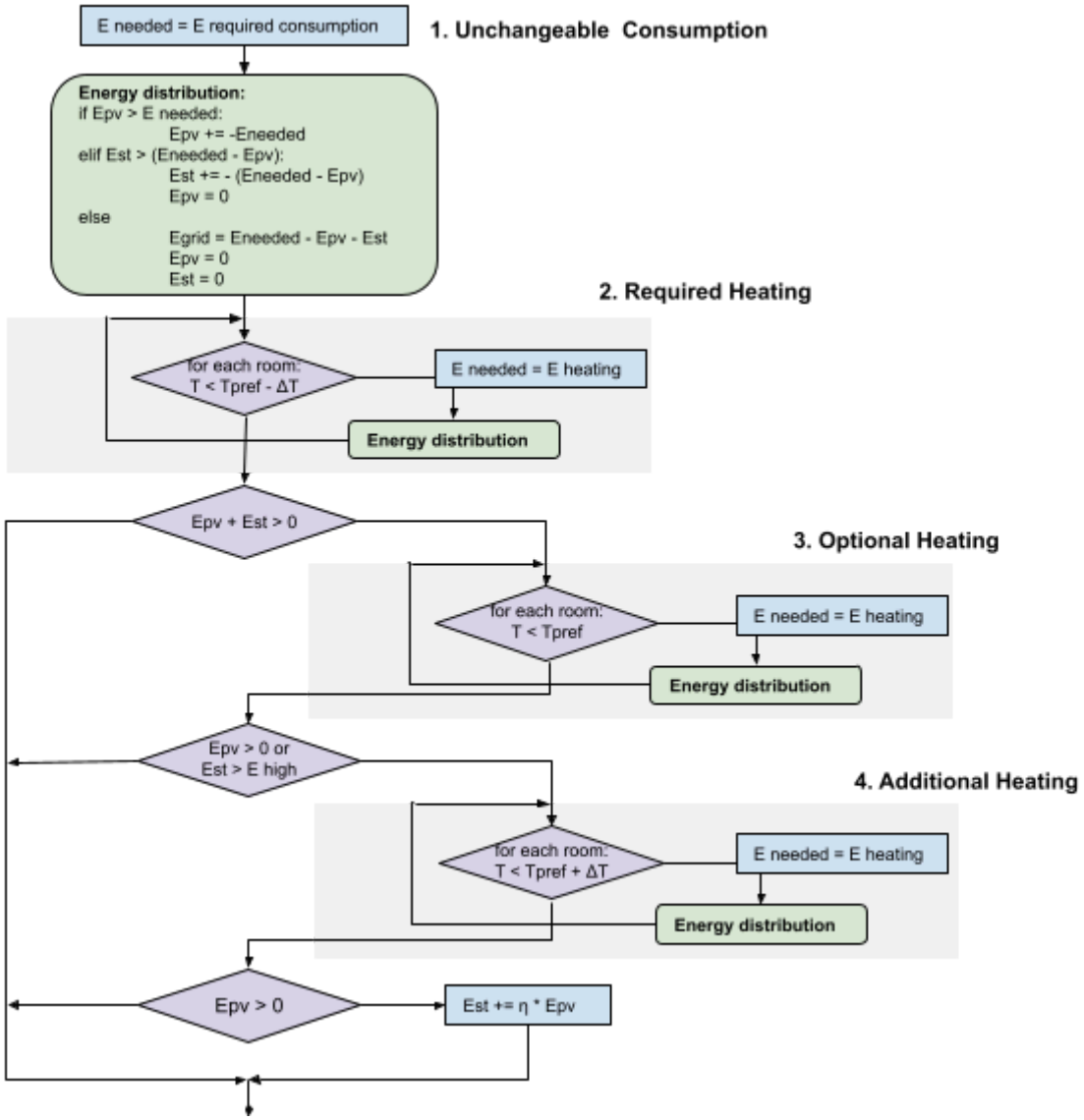


Figure 5.3. Decision algorithm logic diagram.

Then we check the heating priorities of individual rooms. Those where the measured temperature is ΔT lower than expected by the user are heated regardless of the availability of green energy. However, if, after reserving energy for necessary heating, we have energy produced or stored in a warehouse at our disposal, it is used to heat rooms with lower priority. Excess renewable energy (the remaining energy produced and energy stored above the set threshold of a high state of charge) is transferred to heat-up rooms

where the current temperature exceeds the expected one, but not by more than ΔT. In this way, we can accumulate green energy in the form of heat for the future, to reduce consumption from the grid required to meet user requirements. Any remaining produced energy after its distribution for consumption is used to recharge the storage. Only when more energy cannot be transmitted to the storage (due to reaching the maximum capacity of storage or using the nominal power for the charging process), the energy is returned to the global grid. The actions described ensure the maximisation of momentary self-consumption.

SEM Simulator

One of the challenges of this cycle was to develop a simulator of the smart energy meter (SEM simulator). The following requirements were established for this component:

- Scalability – one of the primary goals of the simulator is to test the behaviour of the COGNIT Framework in the situation of a vast number of requests from a considerable number of devices. Thus the SEM simulator should be easy to build and run with multiple instances of the simulator processing different scenarios.
- Portability – the simulator should be working both on a PC with Linux OS as well as on a device with limited resources running on the Phoenix-RTOS system. This enforces usage of C as the language of the implementation.
- Compatibility with meter message API – the meter message API is an interface provided by SEM for applications running on it. This requirement ensures that the application that is developed to run on SEM can use the same interface when paired with the SEM simulator.
- Flexibility – the simulator should take as input the predefined scenarios of energy usage and/or power grid state.
- Interface for household appliances – the SEM simulator should be able to attach simulators of different appliances (photovoltaic, energy storage, heating pump, etc.), so that it tracks the energy consumption and production that they create. This is a crucial feature for simulating the overall status of the Energy Use Case scenario.
- Interface in Python – this provides simplicity of integrating the SEM simulator to various applications. In particular creating the application that uses COGNIT Device Runtime in Python.

Through the process of development of the SEM simulator process a few challenges emerged. These included the following topics:

1. **Format of the scenarios input**. The possible structures of definitions of the scenario were analysed. In particular, many considerations referred to whether the SEM simulator should allow continuous (linear, or curved) fluctuations of voltage and current rather than allowing only discrete changes every quant of time (1 second). The continuous changes seemed as a solution with many advantages, since it reproduces the real life situation more accurately, but for the time being it was decided to stick to a simpler solution with discrete changes as a good enough

option for current purposes. Apart from that, different file formats were analysed to find the best fit.

2. **Interface between SEM simulator and simulators of appliances**. The appliances should provide the SEM simulator with the information about the energy consumption and production that they create. These parameters can change based on a schedule defined in the scenario, or dynamically throughout the simulation (e.g. caused by change of device setting triggered by result from the decision algorithm). In real situations the energy consumed by a device is a result of complicated physics. These may involve different phenomena such as e.g. voltage drops, which were not intended to be simulated precisely. Thus for the time being it was decided that the appliances should provide the SEM simulator with the data on the current (in Amperes).

   Moreover, appliances are not independent from each other, as e.g. functioning of energy storage depends on the amount of energy produced by photovoltaic panels. This requires creating a "gateway" between the SEM simulator and appliances simulators that handles this dependance logic and prepares the data for the SEM simulator.

5.3 End-to-end integration with COGNIT Framework

The simulation facility for the energy use case uses the Device Runtime API in Python for acquiring access to the VM with COGNIT Serverless Runtime and for offloading the decision algorithm described in the previous sections. This mocks a real situation, where the smart energy meter should use COGNIT Serverless Runtime for the same purpose.

The script simulating the application running on the smart energy meter in regular intervals collects data from the simulators of the appliances (photovoltaic, heating devices, etc.) and provides it as the argument to the decision algorithm by offloading its computations to a COGNIT Serverless Runtime. When the result of the algorithm is obtained, the application uses it to trigger possible changes in the settings of the appliances. For convenience the simulation has an adjustable speedup factor.

Description of the application's structure

The simulation environment has been designed in a way that maximises the resemblance of the current simulator of the application running on the smart energy meter and the future implementation of this application running on an actual smart energy meter. Below the descriptions of the important parts of the application are provided alongside the relevant code snippets.

- At the initialisation the script tries to acquire an instance of the Serverless Runtime by communicating with the Provisioning Engine:

```python
sr_conf = ServerlessRuntimeConfig()
sr_conf.name = "Smart Energy Meter Serverless Runtime"
sr_conf.scheduling_policies = [EnergySchedulingPolicy(50)]
sr_conf.faas_flavour = "Energy"

try:
	self.runtime = ServerlessRuntimeContext(config_path="cognit.yml")
	self.runtime.create(sr_conf)
except Exception as e:
	self.cognit_logger.error(f"Error in config file content: {e}")
	sys.exit(1)

while self.runtime.status != FaaSState.RUNNING:
	time.sleep(1)
self.cognit_logger.info("Runtime should be ready now!")
```

- The cycle of the main loop of the application consists of:
  - collecting the data, which serves as the input of the decision algorithm,
  - offloading the decision algorithm,
  - introducing changes to the appliances determined by the result of the algorithm.

```python
# MAIN LOOP CYCLE

now = time.clock_gettime(time.CLOCK_MONOTONIC)
algo_input = self.update_algo_input(now)
algo_res = self.run_algo(algo_input)

# ... some logging

if algo_res is not None:
	self.execute_algo_response(algo_res)
else:
	self.app_logger.warning(f"Decision algorithm call failed")
```

- The script collects the data from the appliance simulators and from the energy meter metrology simulator. The code below illustrates what kind of data is provided for the decision algorithm as input.

```python
step_timedelta_s = math.floor((now - self.last_algo_run) * self.speedup)
self.last_algo_run = now
storage_parameters = self.energy_storage.get_info()
room_heating_params_list = []
for room, value in self.heating_user_preferences.items():
	params = self.room_heating[room].get_info()
```

```
        params["preferred_temp"] = value.get_temp()
        room_heating_params_list.append(params)

energy = self.metrology.get_energy_total()
energy_drawn_from_grid = energy.active_plus - self.last_active_plus
energy_returned_to_grid = energy.active_minus - self.last_active_minus
self.last_active_plus = energy.active_plus
self.last_active_minus = energy.active_minus

pv_reg = self.pv.get_info()["energy_produced"]
energy_pv_produced = pv_reg - self.last_pv_energy
self.last_pv_energy = pv_reg

temp_outdoor = self.temp_outside_sensor.get_info()["temperature"]

charge_level_of_storage = self.energy_storage.get_info()["curr_charge_level"]
prev_charge_level_of_storage = self.last_storage_charge_level
self.last_storage_charge_level = charge_level_of_storage

heating_status_per_room = {}
temp_per_room = {}
for room in room_heating_params_list:
        heating_status_per_room[room["name"]] = room["is_device_switch_on"]
        temp_per_room[room["name"]] = room["curr_temp"]

algo_input = AlgoParams(
        self.model_parameters,
        step_timedelta_s,
        storage_parameters,
        room_heating_params_list,
        energy_drawn_from_grid / 3.6e6,
        energy_returned_to_grid / 3.6e6,
        energy_pv_produced / 3.6e6,
        temp_outdoor,
        charge_level_of_storage,
        prev_charge_level_of_storage,
        heating_status_per_room,
        temp_per_room,
)
```

- The decision algorithm is offloaded to COGNIT Serverless Runtime:

Python

```
def run_algo(self, algo_input: AlgoParams) -> Any:
        ret = None
        if not self.use_cognit:
                ret = self.decision_algo(*astuple(algo_input))
        else:
                offload_ctx = self.runtime.call_async(self.decision_algo,
*astuple(algo_input))
                if offload_ctx is not None:
```

```
                        res_ctx = self.runtime.wait(offload_ctx.exec_id,
self.cognit_timeout)
                        if res_ctx is not None and res_ctx.res is not None:
                                ret = res_ctx.res.res
        return ret
```

- The response of the decision algorithm is used to change settings of the appliances. For the current cycle this means adjusting charge and discharge rate of the energy storage and setting preferred temperature for the heating facility.

```
Python
(
        conf_temp_per_room,
        storage_params,
        next_temp_per_room,
        next_charge_level_of_storage,
        energy_from_power_grid,
) = algo_res
self.energy_storage.set_params(storage_params)
for key, value in self.room_heating.items():
        value.set_params(
                {"optimal_temp": conf_temp_per_room[key]}
        )
```

Additional functionalities

The simulation environment provides the possibility of changing the frequency of offloading of the decision algorithm as well as the speedup factor of the simulation. This may be beneficial for testing the COGNIT Framework against frequent requests to the edge nodes.

When the application is already running and offloading the decision algorithm, the script also enables changing the Serverless Runtime configuration in terms of green energy percentage that the Serverless Runtime is expected to use. This should possibly trigger a migration of the Serverless Runtime to a different edge node.

5.4 Summary of technology developments

In this cycle the Energy Use Case developed two repositories, which are now part of the COGNIT Project's Github. These are the following:

1. `use-case-3-sem-simulator` – Simulator of smart energy meter that provides interface for user applications that can be installed on the meter. The simulator can run predefined scenarios and be integrated with simulators of various appliances.
2. `use-case-3` – Energy Use Case basic demo. Simulates a user application running on a smart energy meter. The application manages various appliances and energy

assets (e.g. photovoltaic or heating devices) according to the decision algorithm, which it offloads to the COGNIT Serverless Runtime.

Both repositories are provided with the BSD 3-Clause licence. For now they are not publicly available, due to the early stage of development, but they are planned to be made public in the near future.

5.5 Plans for the next research cycle M16-M21

Plans for next research cycle are going to focus on following topics:

- **Testing environment**

  Basic plan:
  Integration with EV charger controller for collecting data using OCPP protocol.
  Validation Criteria:
  Ability for collecting data from Electric Vehicle charger. Send data periodically minimum every 5 minutes.

- **COGNIT Integration**

  Basic plan:
  Implement a user app which is using COGNIT Device Runtime in C and run it on the smart energy meter or evaluation board.
  Validation Criteria:
  User app offloads decision algorithm, as well as sends current energy data into provisioned Serverless Runtime.

- **Decision Algorithm**

  Basic plan:
  - Transition from the dummy decision algorithm to an AI one.
  - Add EV charger logic.
  - Add user preferences dynamic change mode to the algorithm.
  - Develop test scenarios for decision algorithm validation.
  - Develop an algorithm for getting optimal decisions based on weather, energy consumption and energy production predictions for the next 24h.
  Validation Criteria:
  - Use the algorithm in a laboratory environment, offload it periodically to the Serverless Runtime with prediction data and get results of computations.
  - Energy self-consumption is quantitatively higher for test scenarios using new version of the algorithm compared to the current simple version.

6. Use Case #4: Cybersecurity

Use Case 4 of the project focuses on Cybersecurity and highlights the utilisation of the COGNIT Framework through the implementation of an anomaly detection scenario within a fleet of rovers (vehicles).

6.1 Current architecture and scenario

Here is a high level diagram of the architecture:

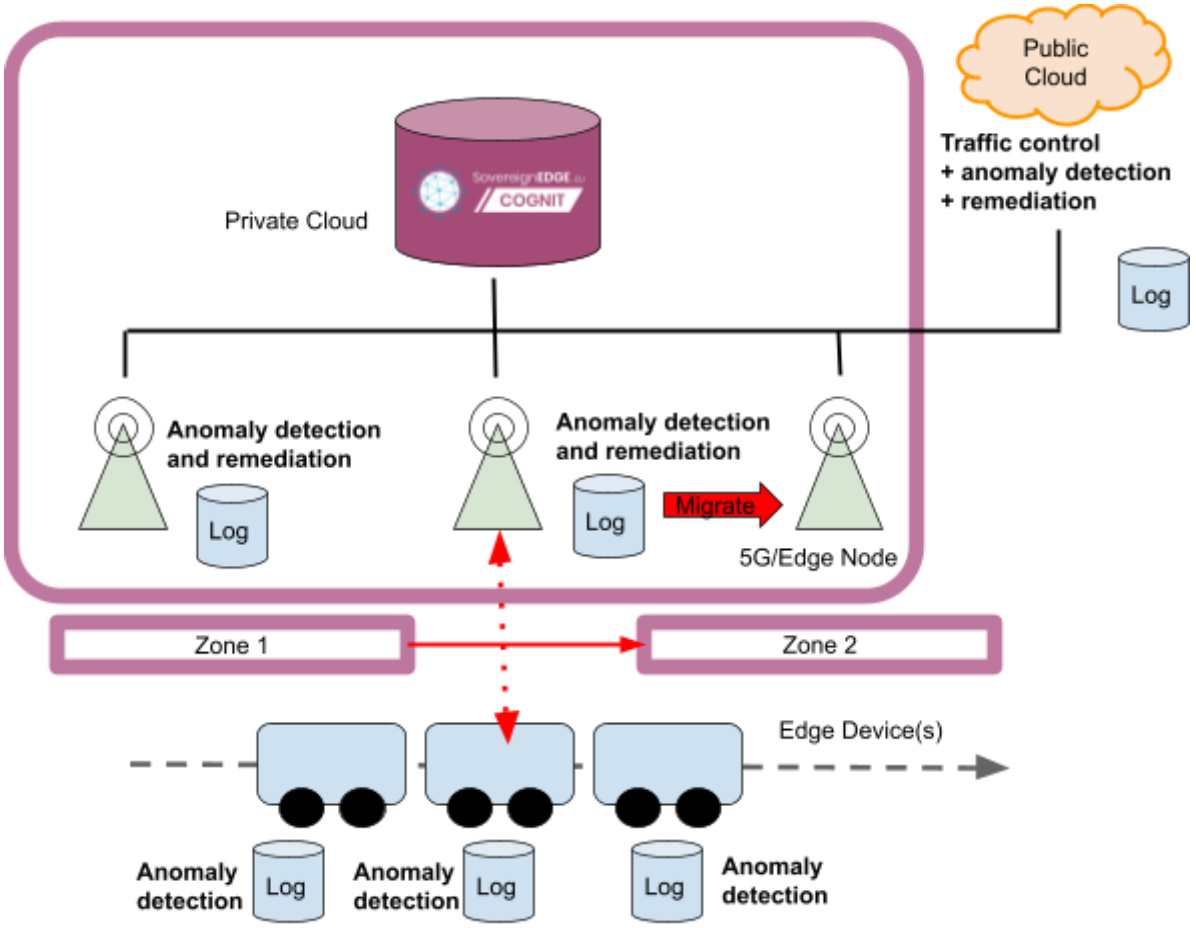


Figure 6.1. High-level architecture for the Cybersecurity Use Case.

The current architecture of the scenario includes the following components:

- **Rover Data Collection**: Rovers collect data, including system logs and metrics such as location, speed, and distance between vehicles.
- **Data Transfer**: The collected data is transmitted to the anomaly detection, which is deployed at the cluster level. This step aims to ensure fast (low latency) and secure transmission of data to the detection system.
- **Anomaly Detection**: Anomaly detection is performed using a Serverless Runtime, with lightweight models running at the edge to reduce latency, and heavier models

running in the cloud. This component analyses incoming data to identify any significant deviations from normal behaviour patterns.

- **Migration Management**: A crucial aspect of Use Case 4 is to demonstrate the COGNIT Framework's ability to manage the migration of Serverless Runtimes based on the itinerary and movement of the rovers. This functionality ensures service continuity and operational efficiency even in dynamic and constantly evolving environments.

6.2 Summary of research done during cycle M10-M15

During the M10-M15 cycle, our team turned to SUSE's OPNI[20] solution, specifically its anomaly detection component (AIOps). We deployed multiple Kubernetes/K3S clusters as well as Rancher for this task. However, from the outset of OPNI deployment, we encountered significant difficulties, requiring multiple attempts to achieve a functional configuration.

OPNI research and deployment

The main challenges we faced included, on the one hand, substantial resource requirements, and on the other hand, bugs that were difficult to identify in the OPNI system. Despite our efforts, we were unable to get the AIOps component to work as intended. Another challenge is that earlier this year we were informed by SUSE that future development support for OPNI was not guaranteed.

Following the OPNI deployment, we explored its operation to determine how best to adapt it to the various components of the framework. However, we found that OPNI's architecture, based on pre-configured agents communicating with an upstream cluster, posed several issues for our use case. Firstly, this configuration did not guarantee the low latency required for rapid and effective anomaly detection at the edge. Moreover, due to the significant infrastructure and resource overhead of OPNI deployment (CPU, memory, storage), we questioned the feasibility and relevance of such deployment in an edge environment. Secondly, the pretrained anomaly detection models available in OPNI were not suitable out of the box for analysing the logs from our use case, and therefore we decided that the best course of action would be to develop new anomaly detection models tailored to the use case.

Given these findings, as well as the low-latency requirements and the need to dynamically migrate runtimes, we decided not to continue to pursue OPNI as a technology for use as general purpose anomaly detection.

In light of this, we explored alternatives that meet the scenario objectives. Ultimately, we decided to develop our own anomaly detection models and are currently in the architecture design phase.

[20] https://github.com/rancher/opni

Anomaly Detection Solutions

In response to the challenges encountered with OPNI solutions, our team has decided to develop our own anomaly detection solution tailored to our specific use case. Leveraging deep learning techniques, we will train custom models capable of accurately detecting anomalies within our rover fleet.

Our anomaly detection solution will use three main types of data:

1. **System Logs**: System logs provide insights into the internal operations and behaviour of the rovers, aiding in the detection of anomalies. For example, we can detect anomalous behaviour such as unauthorised access attempts by monitoring SSH login attempts in the system logs. A sudden surge in failed login attempts or unusual access patterns can signal a potential security breach or unauthorised activity.
2. **GPS Data**: Location data from GPS sensors will enable us to track the spatial movements of the rovers, allowing us to detect deviations from expected routes or locations.
3. **Metrics**: Metrics such as speed and distance between vehicles provide additional contextual information for anomaly detection, such as sudden changes in velocity or unexpected closeness between vehicles.

Our models will be trained to learn the normal behaviour patterns of the vehicles using the provided data. By analysing historical data and learning from past behaviour, the models will develop an understanding of what constitutes typical rover behaviour in various scenarios and environmental conditions.

Once trained, the models will be capable of detecting anomalies in real-time by comparing incoming data streams against learned patterns of normal behaviour. Any deviations from these patterns will be flagged as anomalies, triggering appropriate alerts or actions.

These models should make it possible to identify the vulnerabilities present in the rover's system, and they will also be able to function as an IDS in order to detect intrusion attempts that leave traces in the system logs. Or to check that a rover does not deviate from the planned route using the GPS location logs. They will also be able to detect any errors or introduced into the data collected by distance or speed radars, such as by an attacker attempting to cause an accident.

Anomaly Detection Integration

During the M10-M15 cycle, we also implemented an architecture represented in the diagram below, which illustrates the interaction of our use case with the framework.

We developed a function named “get_authentication_failures”, intended to be executed within the Serverless Runtime. This function uses a regular expression to search for authentication failures within the content of a log file. If authentication failures are detected, the function returns a warning message indicating the details of suspicious connection attempts. Otherwise, it indicates that everything is normal.

```python
def get_authentication_failures(log_content):
   import re
   try:
       authentication_failures = []
       for line in log_content.split('\n'):
           if re.search(r'pam_unix\(sshd:auth\): authentication failure', line):
               authentication_failures.append(line.rstrip())
       if authentication_failures:
           result_message = "Anomaly Detection : Warning ! Connection attempt : \n" +
"\n".join(authentication_failures)
       else:
           result_message = "Anomaly Detection : No message, Everything is fine"
       return result_message
   except Exception as e:
       return "An error occurred while analysing the log file:"  + str(e)
```

Next, we demonstrate how to invoke this function via the framework, by retrieving the content of the log file and passing it as a parameter to the function. We used the call_sync method to execute the function synchronously on the Serverless Runtime, providing the function and the log file content as parameters.

```python
# Example offloading a function call to the Serverless Runtime
# Loading the log file
log_file_path = "./examples/auth.log"
Try:
   with open(log_file_path, "r") as log_file:
        log_content = log_file.read()
except FileNotFoundError:
   print("The specified log file does not exist.")
except Exception as e:
   print("An error occurred while reading the log file:", e)

# call_sync sends to execute sync.ly to the already assigned Serverless Runtime.
# First argument is the function, followed by the parameters to execute it.
result = my_cognit_runtime.call_sync(get_authentication_failures, log_content)
print("Offloaded function result:", result)
```

Finally, the result of the successful execution of the function is displayed by the Device Client Runtime, showing that authentication failures have been successfully detected, if any, and that the system is functioning as expected.

```
device_client_runtime | COGNIT Serverless Runtime ready!

device_client_runtime | [2024-04-12 12:05:52,171] [WARNING]
[_serverless_runtime_client.py::48] Faas execute sync [POST] URL:
http://[2001:67c:22b8:1::e]:8000/v1/faas/execute-sync
```

```
device_client_runtime | Offloaded function result: ret_code=<ExecReturnCode.SUCCESS:
0> res='Anomaly Detection : Warning ! Connection attempt :
- Apr 12 09:22:36 cognit-device-runtime sshd[248279]: pam_unix(sshd:auth):
authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=X.X.X.X
user=atthacker
- Apr 12 10:22:36 cognit-device-runtime sshd[248279]: pam_unix(sshd:auth):
authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=X.X.X.X
user=atthacker' err=None
```

6.3 End-to-end integration with COGNIT Framework

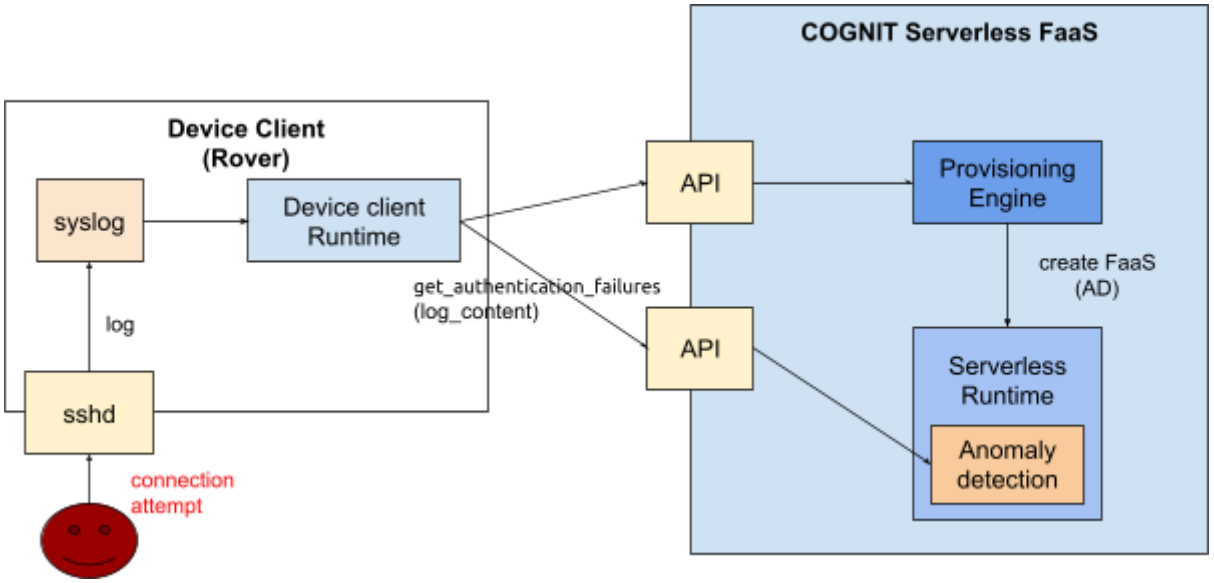


Figure 6.2. Interaction between the Device Runtime and Serverless FaaS.

The cybersecurity case study will use COGNIT services by the creation of the initial Serverless Runtime: a call to the **COGNIT Provisioning Engine** is made to create a Serverless Runtime for the anomaly detection function. The call to the Provisioning Engine provides the necessary data for the Provisioning Engine to identify the most appropriate end platform on which to deploy the Serverless Runtime and create the necessary software stack to deploy the anomaly detection function.

Once the Serverless Runtime has been created, the Device Client can offload anomaly detection to the cloud-edge continuum, performing anomaly detection on the Serverless Runtime by passing system logs as input to the function.

6.4 Summary of technology developments

During this cycle, the cybersecurity use case developed a repository, which is now part of the COGNIT project Github. This is an example of anomaly detection in an authentication log file:

- https://github.com/SovereignEdgeEU-COGNIT/use-case-4/blob/main/examples/dummy_anomly_detection_offload_sync.py

6.5 Plans for the next research cycle M16-M21

As we transition into the next research cycle (M16-M21), our primary focus will be on initiating the development of custom deep learning models tailored to each type of data collected from the rover. These models will form the cornerstone of our anomaly detection solution, empowering us to effectively identify and respond to anomalies within our rover fleet.

During the next cycle, the use case will develop deep learning models to analyse system logs, GPS data, and vehicle metrics to learn the normal behaviour patterns of the rovers. By leveraging COGNIT's capabilities, we aim to ensure that the offloading of our models is optimised for efficiency, scalability, and compatibility within the framework's ecosystem.

In parallel with model development, we will prepare for seamless integration with COGNIT's Data as a Service (DaaS) component. The DaaS component, being an integral part of the COGNIT Framework, will serve as the centralised repository for storing and managing the data required for anomaly detection. We will work on establishing robust communication channels between our anomaly detection system and the DaaS platform to facilitate efficient data retrieval and analysis.

As part of our planning process, we will outline an optimisation strategy to ensure that our anomaly detection solution is well-suited for deployment in edge computing and Serverless Runtime environments. This strategy will involve identifying key performance metrics and architectural considerations to guide the development process.

Over the next few cycles, we also intend to transmit and store data logs to be analysed for anomaly detection using the COGNIT DaaS service that is planned for development in the next cycles. The mobile application will create files in which it will store monitoring data.

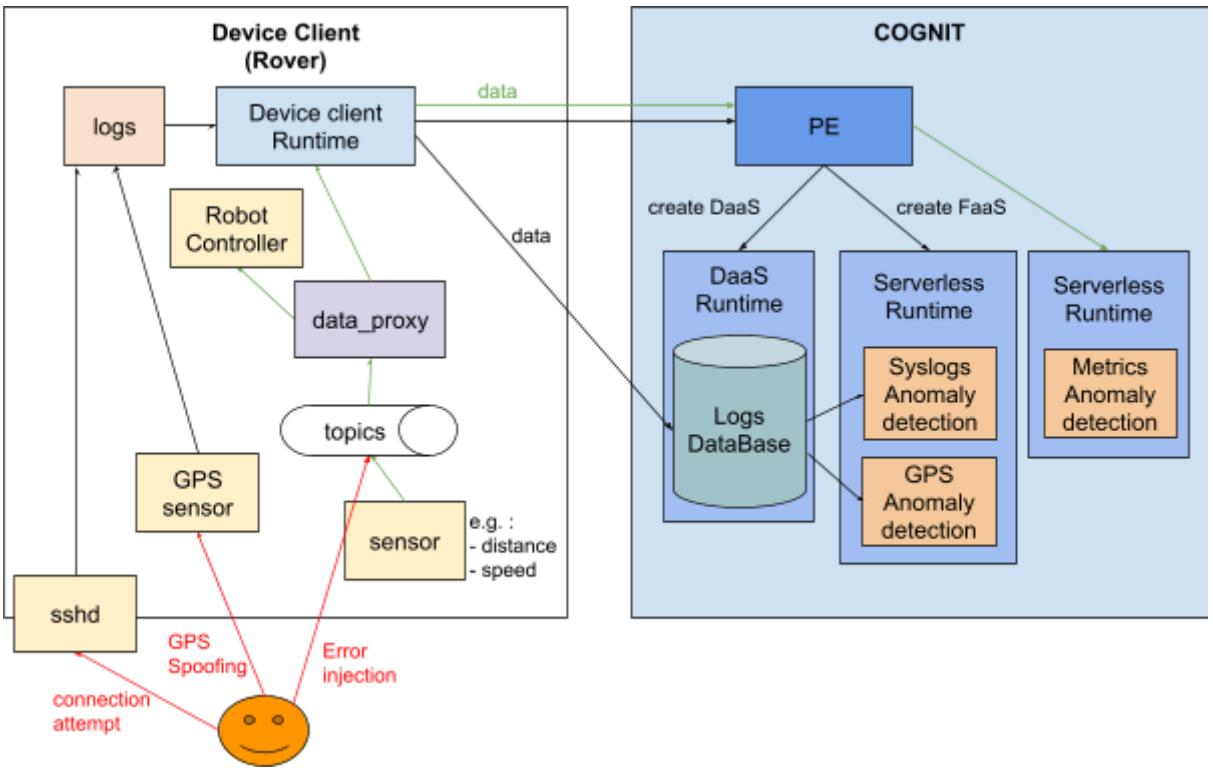


Figure 6.3. Diagram showing the envisioned integration with the COGNIT Framework in the next cycles

We will also plan to use the COGNIT capability to migrate Serverless Runtime from one Edge Cluster to another one in order to avoid deterioration in mobile response time. The COGNIT migration service is transparent for the mobile application. All data required by the Migration Service to plan and perform the Serverless Runtime migration is stored in the COGNIT DaaS available on the whole cloud-edge continuum.

PART II. Software Integration and Verification

7. Software Integration Process and Infrastructure

In this development cycle a new tool, OpsForge,[21] was designed and implemented to facilitate the deployment of the COGNIT Platform as an instance of a Cognitive Serverless Framework for the Cloud-Edge Continuum. This tool is able to create an instance of the COGNIT software stack and deploy it on a target infrastructure, creating a private cloud running on top of resources that can span along the cloud-edge continuum.

This tool receives as inputs the target infrastructure and uses Terraform and Ansible, orchestrated through developed scripts, to perform the following actions:

- Request resources from the target infrastructure.
- Deploy the COGNIT components using Ansible.
- Configure infrastructure aspects: networking, storage and hypervisors.
- Configure COGNIT components to run smoothly on the target infrastructure: for instance, populate OpenNebula with content required by the COGNIT use cases, like specific Serverless Runtime flavours.

Currently two kinds of infrastructures are supported out of the box: Amazon AWS (through the EC2 service), and on premises resources, using a static list of SSH reachable servers. The initial version of OpsForge focused on Amazon AWS deployment.

Once the initial COGNIT Platform has been deployed, the OpenNebula one-provision tool[22] can be used to add computing resources (i.e., hypervisors) to create a private cloud overlay on cloud-edge continuum resources, from a centralised datacenter to the near edge, servicing the FaaS environment to different client devices in different locations with optimal latency.

OpsForge is a command line interface application that runs on a local machine. It has some requirements that need to be met to ensure proper execution of the tool: ruby, terraform, awscli, ansible and a valid ssh key to connect to AWS EC2 instances.

7.1. Deployment in AWS

The deployment will create its own Virtual Private Cloud (VPC), Internet Gateway and subnets with the proper network configuration for the EC2 instances to communicate with each other. The deployment of the COGNIT Platform can target any available AWS region.

OpsForge needs as inputs the means and characteristics of the target infrastructure. In Figure 7.1 you can find the OpsForge input needed to deploy the 1.0 version of the COGNIT Framework on AWS. With this YAML file, the COGNIT software stack can be deployed in the target infrastructure running the following command:

[21] https://github.com/SovereignEdgeEU-COGNIT/cognit-ops-forge
[22] https://docs.opennebula.io/6.8/quick_start/operation_basics/provisioning_edge_cluster.html

```
Unset
       $ ./opsforge deploy <opsforge_template>
```

The output of this command can be seen in Figure 7.2, with information about the endpoints of the different components. After the command finishes, a COGNIT Framework is deployed in AWS as per Figure 7.3, and after adding nodes using one-provision, it is fully functional and a Device Client can start making use of the offload functionality.

```
Unset
infra:
 :aws:
   :ec2_instance_type: t2.medium
   :volume_size: 125
   :region: "us-east-1"
   :ssh_key: "dann1"
   :ssh_key_path: '~/.ssh/id_rsa'
:cognit:
 :engine:
   :port: 6969
   :version: release-cognit-1.0
 :ai_orchestrator:
   :version: release-cognit-1.0
 :cloud:
   :version: 6.8
   :ee_token: "XXXX:XXXX"
   :web_ports:
     :main: 80
     :next_gen: 443
   :extensions:
     :version: release-cognit-1.0
```

Figure 7.1. OpsForge 1.0 Input YAML Template for AWS Infrastructure

```
Unset
Setting up infrastructure on AWS
Infrastructure on AWS has been deployed
Took 16.729304 seconds
Installing Frontend and Provisioning Engine
Frontend and Provisioning Engine installed
Took 74.6311 seconds
Setting up Frontend for Cognit
Frontend ready for Cognit
Took 3.686645 seconds
Took 2.7e-05 seconds
{
  "cloud": "ec2-3-81-149-9.compute-1.amazonaws.com",
```

```
  "engine": "ec2-35-173-132-188.compute-1.amazonaws.com",
  "ai_orchestrator": "ec2-52-90-15-180.compute-1.amazonaws.com"
}

Logs available at ./opsforge.log'
Take a look at AWS cluster provisioning in order to setup your KVM cluster
https://docs.opennebula.io/6.8/provision_clusters/providers/aws_provider.htm
l#aws-provider
```

Figure 7.2. Example of execution output when running OpsForge for AWS infrastructure

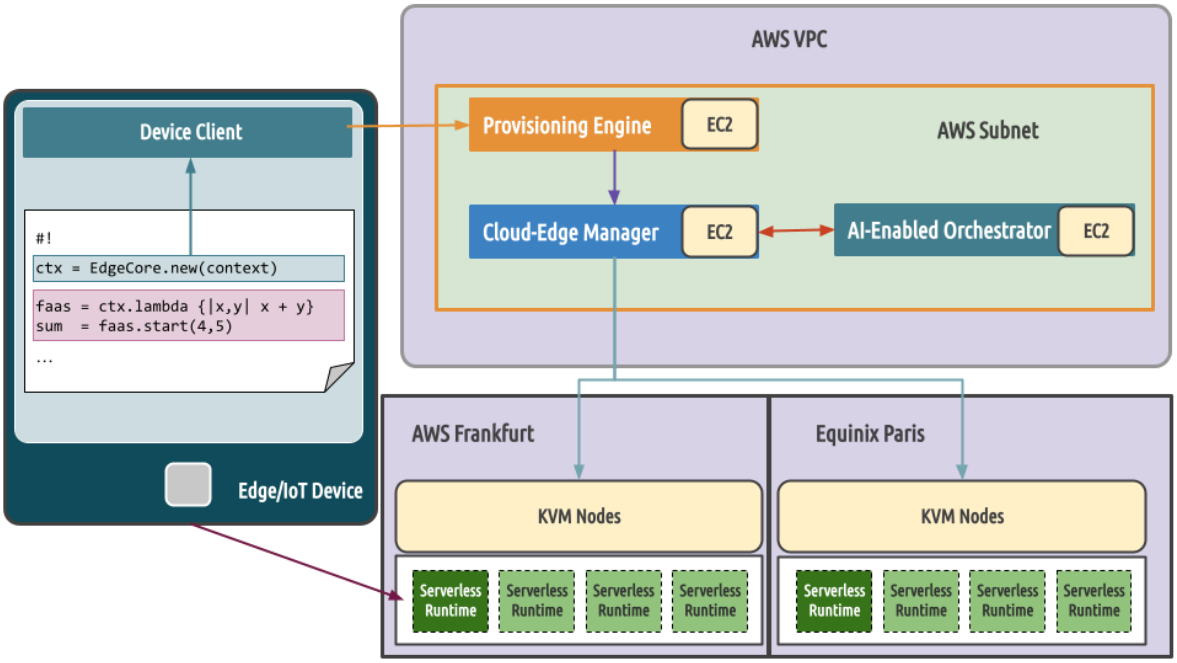


Figure 7.3. COGNIT components installed by OpsForge on AWS infrastructure.

7.2. On-premise deployment

The process of using OpsForge to deploy the COGNIT Platform on premises follows the same steps as described in the AWS section (Section 7.1). There are some light requirements for the on premises hosts, namely they need to have a Ubuntu 22.04 installation and root ssh passwordless access with an authorised SSH key accessible by OpsForge.

When using OpsForge to deploy on specific hosts on premises, there should be no :aws section in the YAML file. It should instead be replaced by a :hosts section where the hostname must be listed for each host that will host one of the COGNIT component (see

Figure 7.4 below). Using the YAML file specified in Figure 7.4, OpsForge can deploy the 1.0 version of the COGNIT Framework in an on-premise environment.

```
Unset
:infra:
  :hosts:
    :cloud: 172.20.0.4
    :engine: 172.20.0.9
    :ai_orchestrator: 172.20.17
:cognit:
  :engine:
    :port: 6969
    :version: release-cognit-1.0
  :ai_orchestrator:
    :version: release-cognit-1.0
  :cloud:
    :version: 6.8
    :ee_token: "XXXX:XXXX"
    :web_ports:
      :main: 80
      :next_gen: 443
    :extensions:
      :version: release-cognit-1.0
```

Figure 7.4. OpsForge 1.0 Input YAML Template for on-premise infrastructure.

Afterwards, hypervisor nodes need to be added to the OpenNebula Cloud Edge manager to achieve a fully functional COGNIT Framework, as described in Figure 7.5.

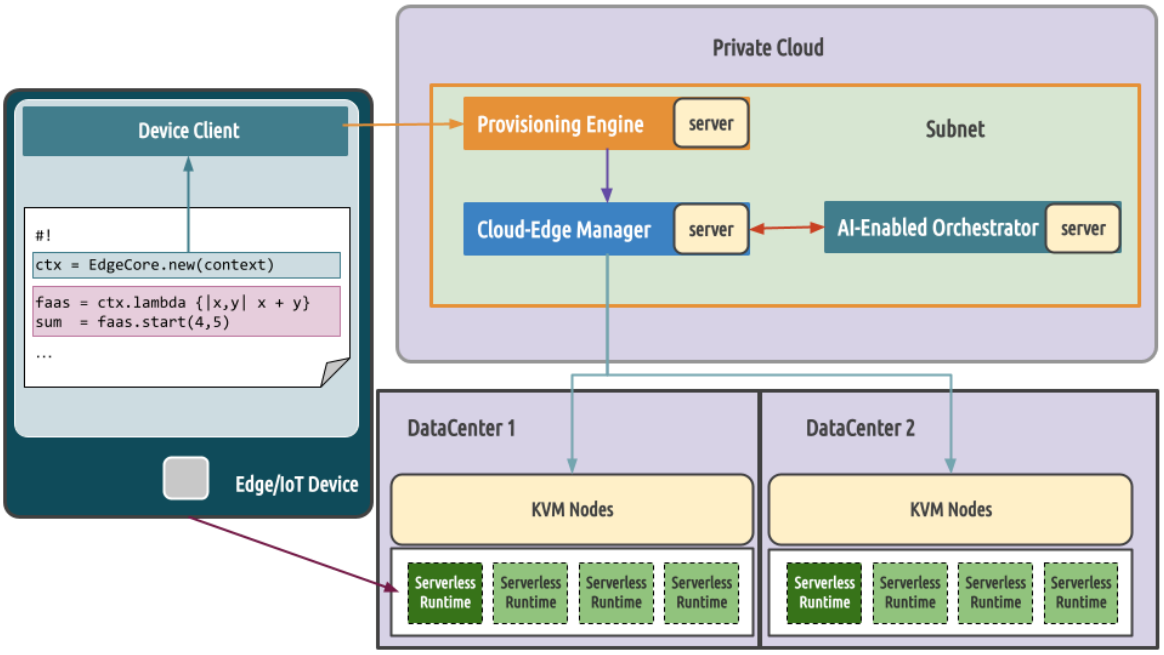


Figure 7.5. COGNIT components installed by OpsForge on-premise hardware.

7.3. Deprovisioning of the COGNIT Platform

Once the COGNIT deployment is no longer needed, the provisioned resources can be deprovisioned by issuing the command “opsforge clean”:

```
Unset
./opsforge clean
Destroying resources created on AWS
COGNIT deployment successfully destroyed
```

7.4 First Version of the COGNIT Software Stack

The list of components deployed by OpsForge that constitutes the 1.0 version of the COGNIT software stack, can be found in Table 7.1. This can be deployed in AWS or on premises as described in Section 7.1 and Section 7.2, respectively.

Name	Documentation	Testing	Installation
Device Client (Python)	Wiki documentation	Test folder	README
Device Client (C)		Test folder	README
OpenNebula	Official doc	Q&A	Install guide
Serverless Runtime	Wiki documentation	Test folder	README
Provision Engine	User guide	GitHub Actions	Admin guide
AI-Enabled Orchestrator	User guide	See docs	Install guide

Table 7.1. Main COGNIT Framework components

At the end of this second development cycle, OpsForge fully automates the deployment of the following components:

- OpenNebula as the Cloud/Edge Manager. Extensions made to OpenNebula in the context project that are not contributed upstream, and which can be found in the opennebula-extensions repository[23] are applied automatically. The OpenNebula services are deployed in a single VM.

[23] https://github.com/SovereignEdgeEU-COGNIT/opennebula-extensions

- Provisioning Engine, using the corresponding COGNIT repository, is deployed on a single VM.
- AI-Enabled Orchestrator, the VM to contain the software is created although at this point it is not deployed automatically.

After the initial deployment a number of manual steps are needed to deploy the other components of the COGNIT software stack:

- AI-Enabled Orchestrator needs to be deployed manually using the VM created by OpsForge. This will be automated at the beginning of the third development cycle.
- OpsForge creates the needed resources in OpenNebula to represent the Serverless Runtime. The creation of the Serverless Runtime image is not automated (there are plans to address this in the third development cycle using SUSE's KIWI technology[24] integrated into the Open Build Service[25]). Currently Serverless Runtimes for the four different flavours corresponding to the other four use cases are defined, using a common Serverless Runtime version. Those Serverless Runtimes can be updated by uploading a new image to OpenNebula and adjusting the Serverless Runtime definition to use the new version.

Afterwards, the OpenNebula one-provision tool can be used to add computing resources, or they can be added manually. Finally, the Device Client software can be used to offload application specific functions to the COGNIT Platform, serving a FaaS paradigm in the cloud edge continuum.

7.5 Testing of the COGNIT components

At the end of this second development cycle, testing of the COGNIT components is performed per component. There are mainly two types of components with respect to testing:

- pre-existing components (such as OpenNebula), which needs to be tested as part of their regular Q&A process.
- COGNIT specific components that need to be tested using dedicated frameworks.

The state of testing per component is reviewed below.

Cloud/Edge Manager

COGNIT Cloud Edge Manager is based on OpenNebula, and as such its testing is performed as part of OpenNebula's Quality and Assurance process.

Provisioning Engine

As introduced in Deliverable D5.2, each commit to the master branch of the Provisioning Engine repository[26] spawns a new GitHub-backed container where:

[24] https://osinside.github.io/kiwi/
[25] https://openbuildservice.org
[26] https://github.com/SovereignEdgeEU-COGNIT/provisioning-engine

1. The Provisioning Engine repository is checked out.
2. All the Provisioning Engine dependencies are installed.
3. The Provisioning Engine is installed.
4. The Provisioning Engine is configured using the appropriate Cloud-Edge Manager endpoint.
5. The Provisioning Engine is launched.
6. A set of smoke tests are run (i.e. to create a new Serverless Runtime, update it, and delete it).
7. The Provisioning Engine is stopped.
8. The Provisioning Engine is uninstalled.

This comprehensive testing process ensures that errors are identified during the development phase as soon as possible. This check is compulsory before a new version of the Provisioning Engine is deployed on the COGNIT infrastructure.

Device Client

There are two repositories linked to this component:

- Python version: Unit tests are available as well as integration tests (against a valid Serverless Runtime, usually running locally where the Device Client is being executed) within a test folder in the corresponding repository. Furthermore, as a knowledge base, the Wiki documentation could be used as a reference for FAQs.
- C version: Regarding this version, unit tests are also provided in a test folder in the corresponding repository.

Serverless Runtime

Unit tests folder is located at this location in the repository. Additionally, there is the Wiki documentation that can be used as a reference for FAQs.

AI-Enabled Orchestrator

The AI-Enabled Orchestrator consists of two main components (please check D4.2 for further details):

- EnvServer has unit tests, see EnvServer documentation.
- MLServer only implements integration tests including the full stack except the EnvServer frontend, see MLServer documentation.

Additionally, the ML models can be also tested, taking into account that this requires correctly formatted data. Please check IDEC and MC2PCA testing documentation for further details on this.

Integration Testing

With the introduction of the OpsForge tool in this second development cycle, the possibility to create end to end integration testing across the whole COGNIT Framework is now feasible. Current Provisioning Engine integration tests will be used as a basis and expanded in order to cover all the other integrated components and functionality of the COGNIT Framework, and also in the future incorporating security tests related to penetration, vulnerability analysis, intrusion detection, and security remediation.

8. Testbed Environment

This section includes an overview of the current state of the COGNIT Testbed environment. The environment is a centrally managed edge-cloud infrastructure as a service platform (IaaS) based on OpenNebula, combined with COGNIT-specific components that implement the serverless platform on top. The central management is deployed at RISE's ICE Datacenter in Luleå (Sweden), to which compute/edge hosts at use case sites around Europe can be connected.

8.1 Virtual Machines

For the central IaaS management and COGNIT software we have currently provisioned five Ubuntu 22.04 VMs on ICE Datacenter. The details of these VMs are shown in Figure 8.1:

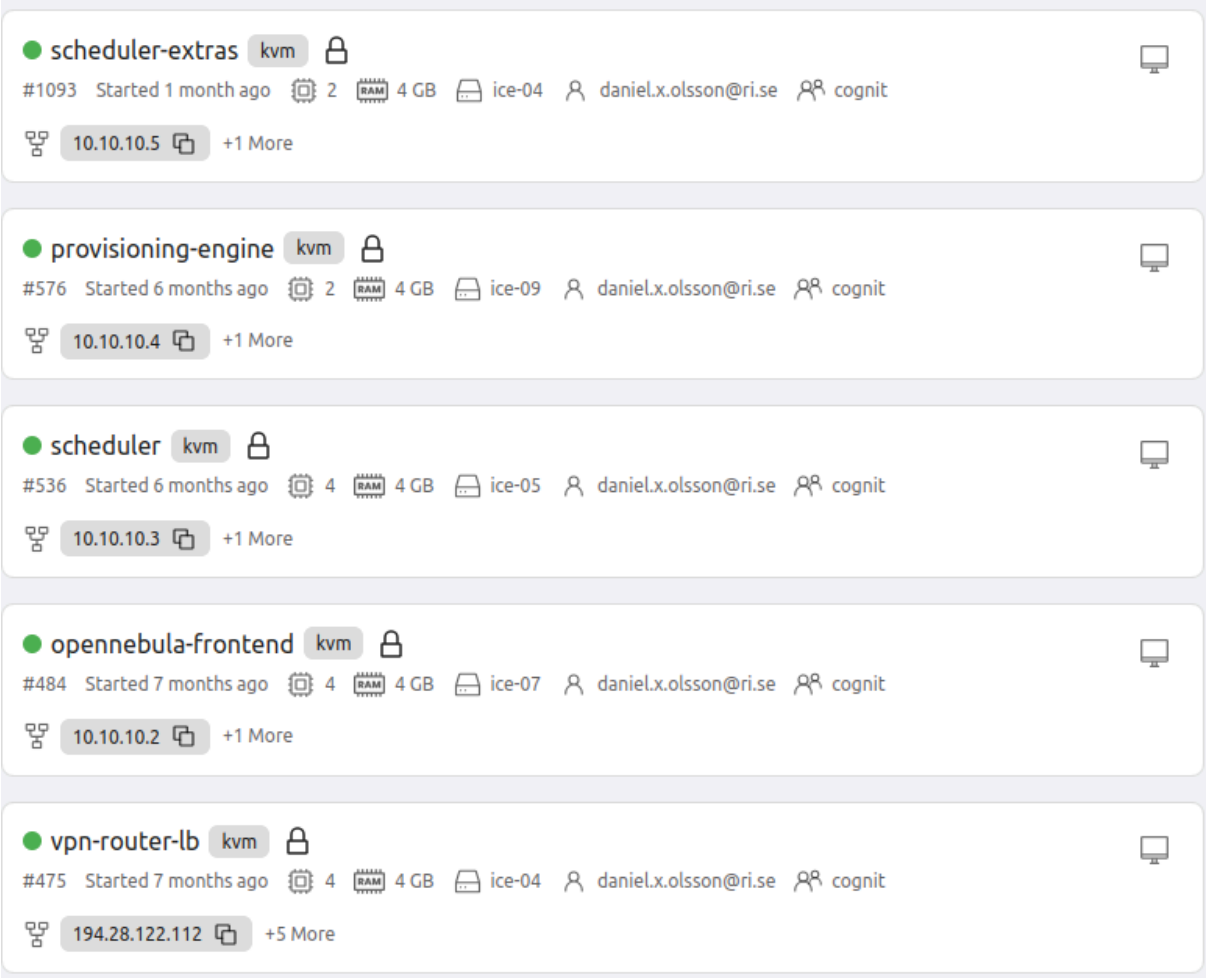


Figure 8.1. Screenshot from ICE Connect with the details of the VMs running the central IaaS management.

8.2 Compute Hosts

Bare-Metal servers at edge sites have been added as compute hosts to the COGNIT testbed, using OpenNebula's cloud edge architecture. The testbed currently integrates resources from two clusters: one cluster (RICE) with two nodes in Luleå (Sweden) and one cluster (dyn#) with four nodes managed by OpenNebula Systems in Madrid (Spain):

Cluster	Name / Link	VPN IP	CPU (vCore)	Memory (GB)	GPU
ice	p02r11srv01	10.22.0.21	24	256	Nvidia GTX 2080ti
ice	p02r11srv15	10.22.0.22	24	256	Nvidia GTX 1080ti
dyn	dyn10	10.22.0.10	16	16	N/A
dyn	dyn11	10.22.0.11	16	16	N/A
dyn	dyn12	10.22.0.12	16	16	N/A
dyn	dyn13	10.22.0.13	16	8	N/A

Figure 8.2. The virtualization hosts from the two cloud-edge clusters managed by the testbed.

8.3 Networking

The network topology is designed to ensure efficient communication between the VMs and the public internet. The 'vpn-router-lb' VM acts as a gateway, connecting the internal network with the public internet. The internal network uses the IP range 10.10.10.0/24 and fd67::/64 for IPv4 and IPv6 respectively.

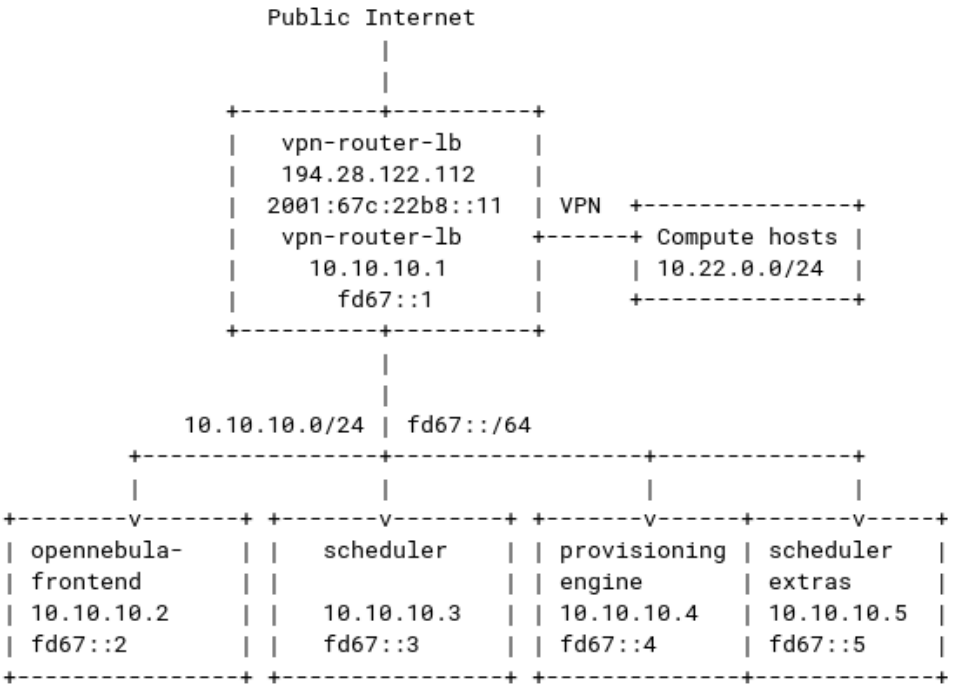


Figure 8.3. Diagram of the network used in the COGNIT testbed.

8.4 Load Balancer

The load balancer is implemented using Nginx Proxy Manager which is Nginx combined with a management web interface. It acts as a load balancer and SSL termination point and certificate handler for all services in the infrastructure. It is deployed in the virtual machine called vpn-router-lb. The following are the configured endpoints that publicly expose various COGNIT services:

- Sunstone: https://cognit-lab.sovereignedge.eu
- XML-RPC API: https://cognit-lab.sovereignedge.eu/RPC2
- Fireedge: https://cognit-lab.sovereignedge.eu/fireedge/sunstone
- OneFlow API: https://cognit-lab-oneflow.sovereignedge.eu
- Provisioning API: https://cognit-lab-pe-api.sovereignedge.eu
- Grafana: https://cognit-lab-grafana.sovereignedge.eu

8.5 COGNIT-LAB Management VPN

The OpenNebula Frontend needs to be able to actively reach all compute hosts, but these might not have public IPs. To solve this, we connect each compute host to a COGNIT-LAB management VPN network. The VPN is implemented using Wireguard.

8.6 Public IPv6 subnets

Every site needs a public IPv6 subnet to be used by Serverless Runtime VMs. Currently only the site in Luleå has provided one, which is 2001:67c:22b8:1::/64.

8.7 IPv4toIPv6 Gateway

Because many developers do not have access to the public IPv6 network from their local computers, we have implemented an IPv4toIPv6 gateway. This was implemented using Wireguard VPN that is terminated at vpn-router-lb.

8.8 Workload tests

All four of the use cases have tested their full end-to-end integration with COGNIT to offload their functions to the COGNIT Testbed. As part of their research work done for Use Case 1 (Smart City), ACISA setup and ran several additional workload tests on the testbed, to assess and validate the performance of their end-to-end integration with COGNIT as the number of offloading requests scale. The results of these initial workload tests are presented below. The idea is to extend this in the third Research & Innovation Cycle (M16-M22) to systematically simulate real-life workload scenarios for all the use cases.

One of their goals in the second Research and Innovation Cycle was to identify tools that would allow them to assess how much workload the COGNIT Framework is able to support in its current state.

After some research they found that Locust (https://locust.io/) would be a testing tool very well suited to our case. It is an open source project that is based on Python and allows a very easy integration of performance tests running Python code. In addition it provides a very convenient GUI to configure the tests, run them and get some statistics from a web console.

For their case, they added a Test class to their code:

```python
class UC1_Test(HttpUser):
    wait_time = between(1, 5)

    @task
    def run_faas(self):
        start_time = time.time()
        result = remote_call()
        total_time = (time.time() - start_time) * 1000

        # Record the execution time as a custom event

        events.request.fire(
            request_type="Cognit remote call",
            name="UC1_FAAS",
            response_time=total_time,
            response_length=0,
            exception=result
        )
```

Where we have some test behaviour configuration, some statistics event generation, and a call to our remote FaaS call (i.e. remote_call).

The following figures show some preliminary results:

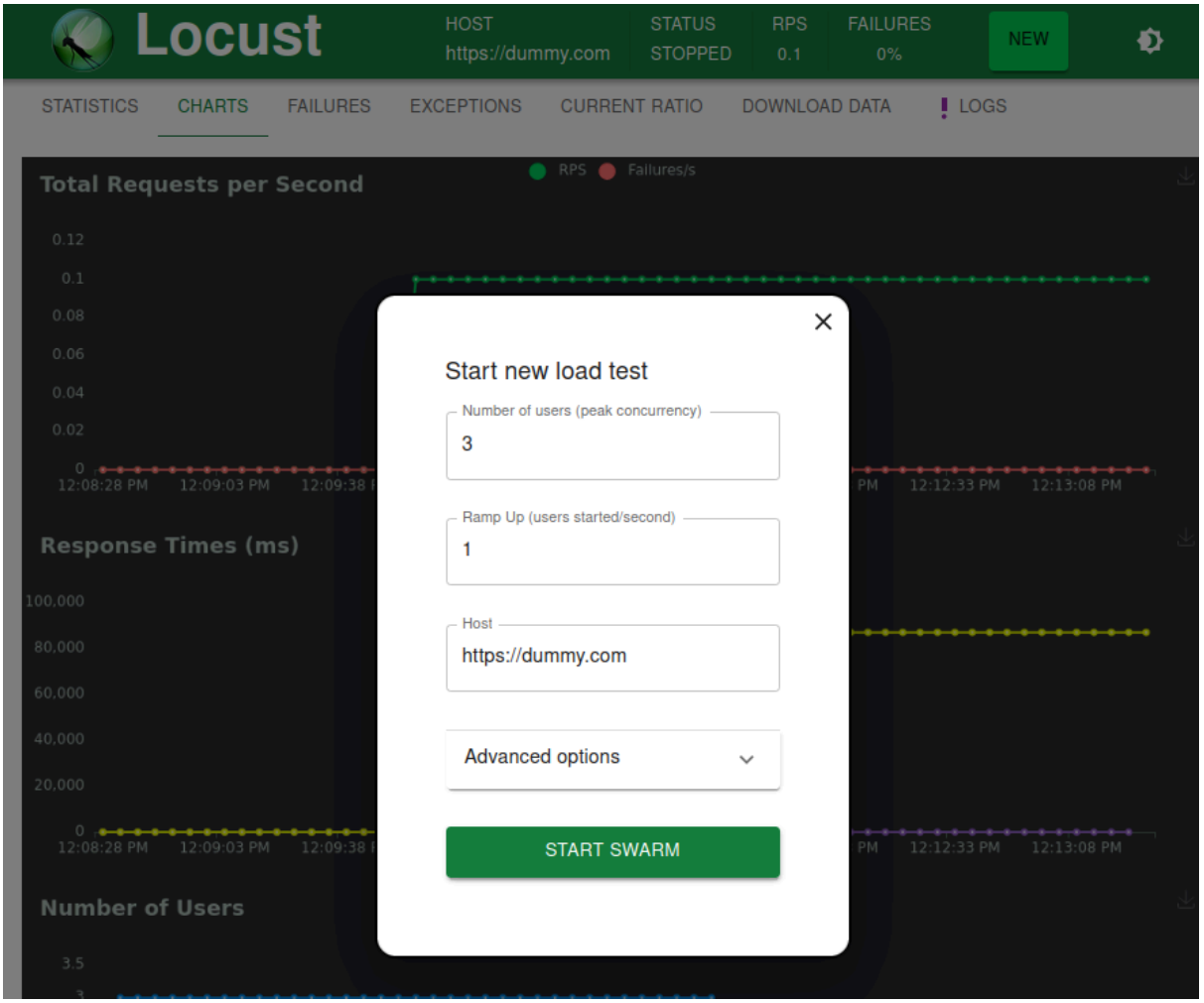


Figure 8.4: Configure and run the tests

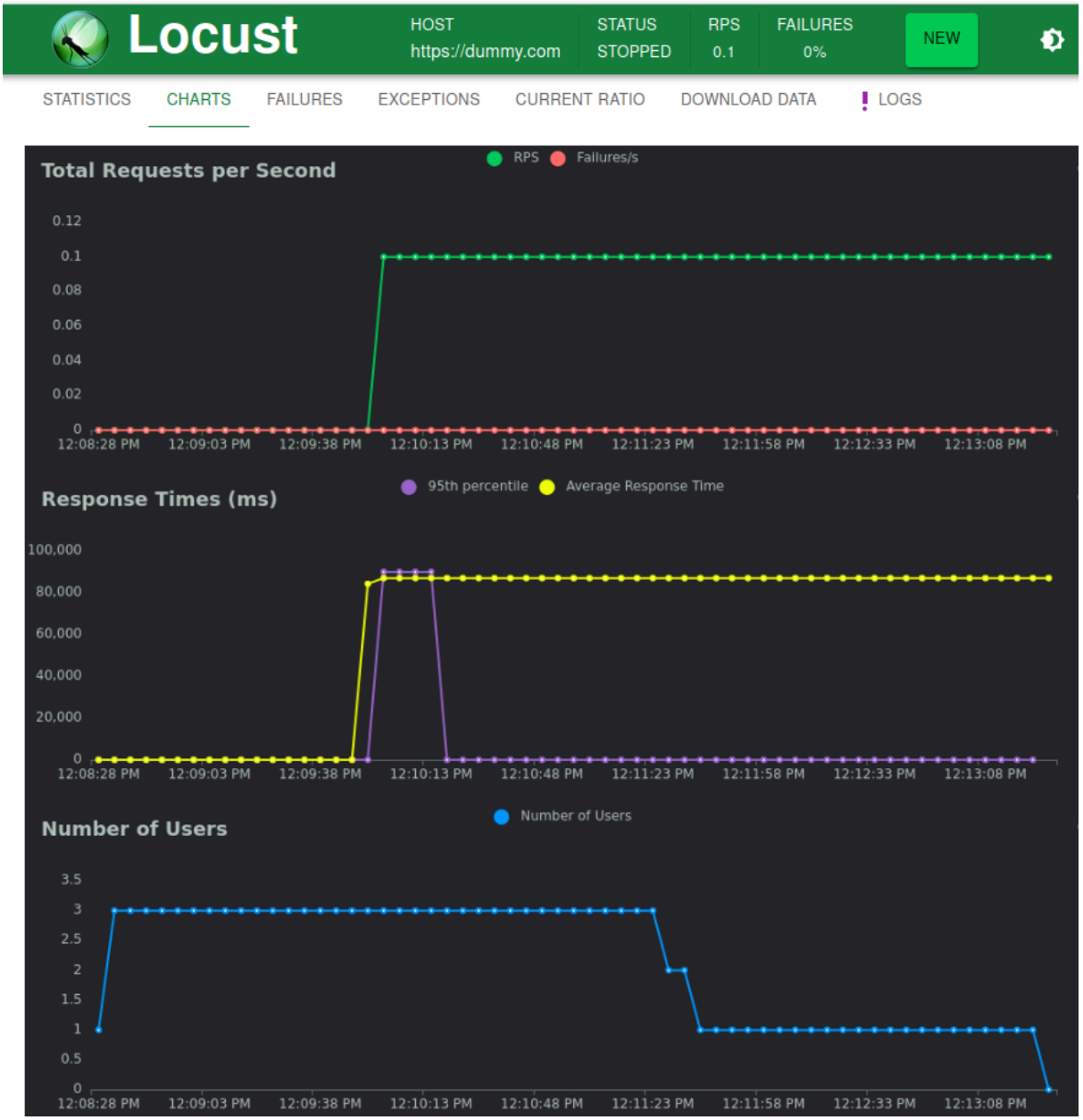


Figure 8.5: Sample of results

Some performance results

The purpose of those performance tests was to generate some workload data, for the team developing the AI-Enabled Orchestrator, since the data gathered by the monitoring (Prometheus) service will be used to develop AI models. During these tests, we found some interesting results from the point of view of the statistics collected by Locust.

In our first tests, we found that at some random pattern, some calls were returning an HTTP 500 error. Once analysed it was determined that some race condition was happening with concurrent requests on the Provisioning Engine, and the problem was quickly fixed by the responsible team.

After this change, we were able to increase the number of concurrent requests without any further issues.

Another interesting result was the increasing time taken to complete the concurrent executions as their number of tasks was increasing, which makes sense because the OpenNebula system has an allocation of resources, which would reach a limit under an increasingly higher workload.

We have been allowing 3 minutes for the execution of the concurrent tasks, and the behaviour of Locust is that all the tasks still running after those 3 minutes would be killed. We added a 2 minute grace period, so that more tasks would have time to finish. The typical execution time for a task is about 1 minute. With 10 concurrent tasks running for 3 minutes, we found that one task didn't have enough time to finish (see Figure 8.6 below):

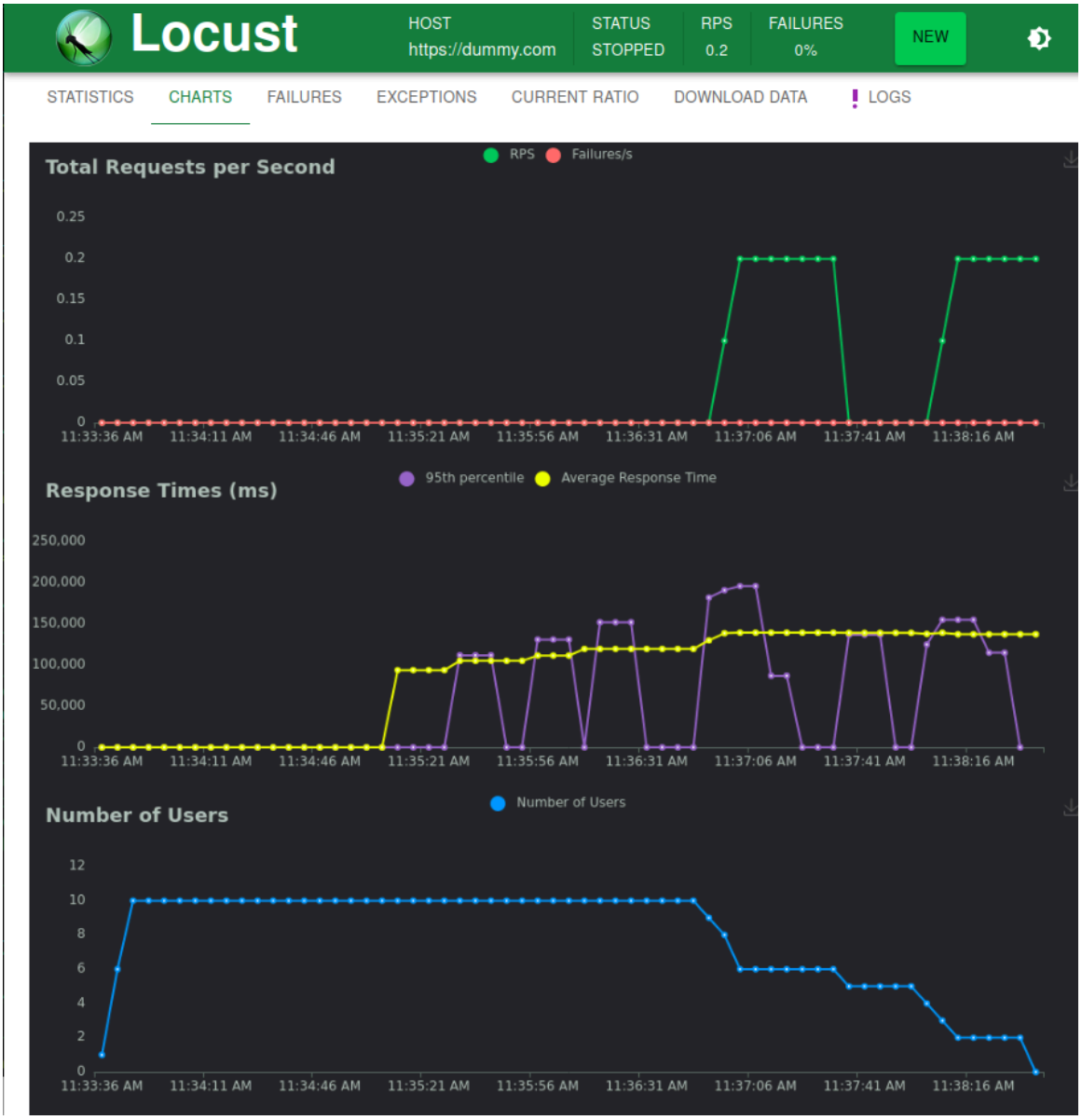


Figure 8.6: Results for 10 concurrent tasks

When we increased concurrency to 20 parallel tasks, we found that 19 tasks didn't have enough time to finish.

Note: with a typical execution time of 1 minute per task, with 20 concurrent tasks and 3 minutes of execution extension, we should have around 60 tasks executed, and from them, 19 didn't have enough time to finish.

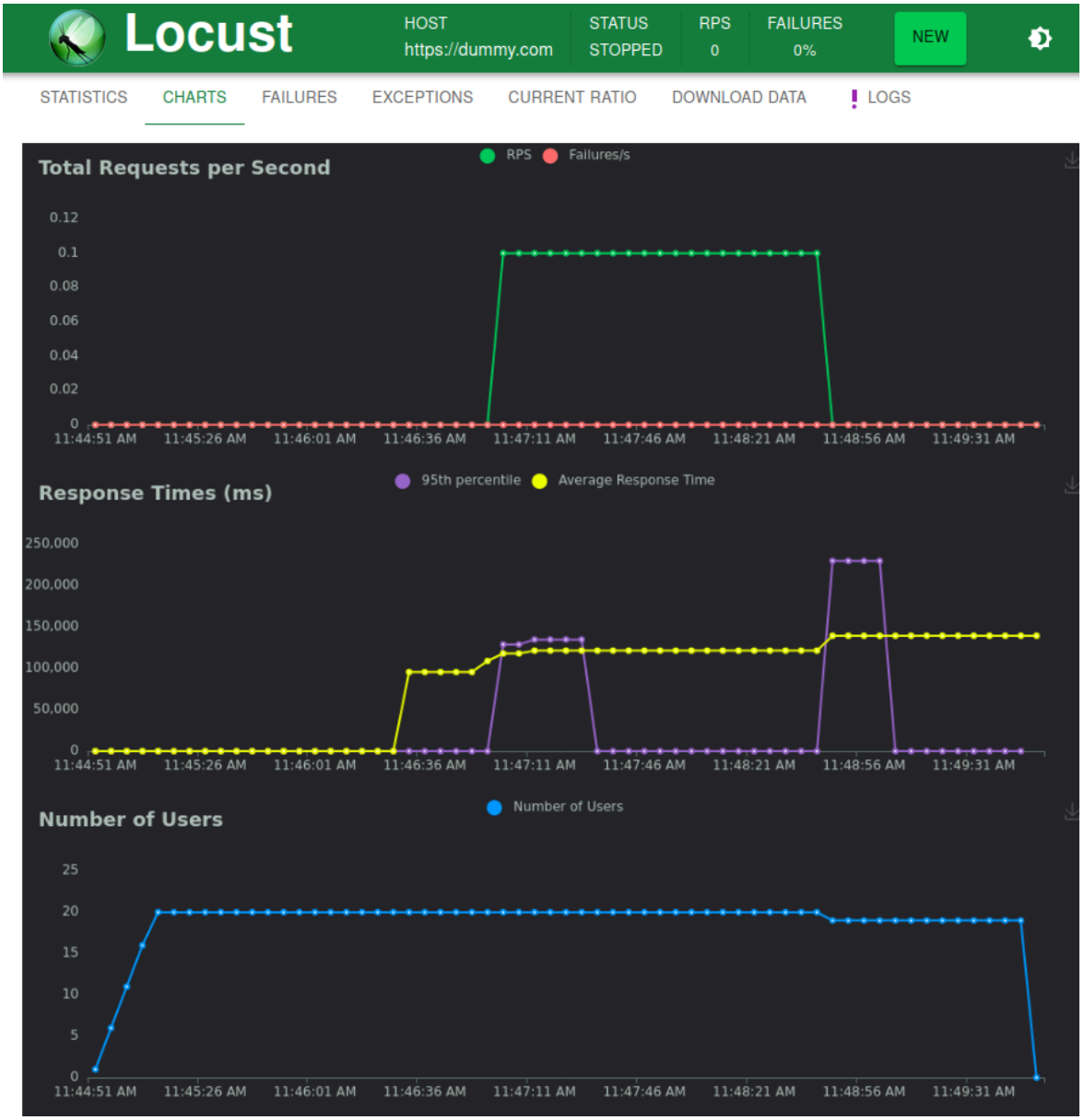


Figure 8.7: Result for 20 concurrent tasks

That is an interesting result that showed that we had reached some capacity limit, and we needed to adjust the resources allocation in case that, in real world conditions, we need to support such a workload or a bigger one.

It could be also a hint to explore different ways to execute the remote calls so that the starting time could be reduced.

8.9 Future plans for the COGNIT Testbed

In the next Research and Innovation Cycle (M16-M22), we are planning to enhance the testbed with GPU access and additional edge compute nodes.

GPU access to Serverless Runtime

Currently GPUs are available in two bare-metal servers, however these are not yet configured to be used by the Serverless Runtime VMs. We will add this support by configuring the hosts to load the vfio-pci kernel driver for the GPUs and let the hypervisor know about the GPUs so that they can be assigned to VMs using PCI passthrough. For the Serverless Runtime to be able to use the assigned GPUs, the runtime VM image must be complemented with Nvidia drivers and tools.

Add additional edge compute nodes

The plan is to add compute nodes at all Use Case sites in the next cycle. When ready, these local nodes will be added as virtualization hosts to the COGNIT testbed.

9. Software Requirements Verification

Possible status are: NOT STARTED | IN PROGRESS | COMPLETED

9.1 Device Client

SR1.1 Interface with Provisioning Engine

Status: IN PROGRESS

Description: Users are able to download and build the Device Runtime Python library. Following the instructions[27] of the Project's GitHub repository, a user can configure the Device Runtime to point to a valid Provisioning Engine endpoint and communicate with the requested Serverless Runtime to offload the execution of a Python function. All the library functionalities can be tested standalone by executing the unit tests provided on the GitHub repository[28].

Completed Verification Scenarios:

- [VS1.1.1] The Device Client asks the Provisioning Engine for a Serverless Runtime with certain characteristics and it receives the ID of the created Serverless Runtime.
- [VS1.1.2] The Device Client asks the Provisioning Engine for information about the created Serverless Runtime.
- [VS1.1.3] The Device Client requests the Provisioning Engine to update the features of a Serverless Runtime. The Device Client requests again information about the Serverless Runtime to verify that the Serverless Runtime has been modified.
- [VS1.1.4] The Device Client requests the Provisioning Engine to delete a Serverless Runtime. The Device Client requests information about the Serverless Runtime to verify that it no longer exists.

SR1.2 Interface with Serverless Runtime

Status: IN PROGRESS

Description: Currently, the Device Client can communicate with the implemented Serverless Runtime component to upload a function, execute it, and retrieve the result.

Completed Verification Scenarios:

- [VS1.2.2] The Device Client uploads a function to the Serverless Runtime and receives an acknowledgement.

[27] https://github.com/SovereignEdgeEU-COGNIT/device-runtime-py/blob/main/README.md
[28] https://github.com/SovereignEdgeEU-COGNIT/device-runtime-py/tree/main/cognit/test

- [VS1.2.3] The Device Client requests the execution of a function to the Serverless Runtime and receives the result of the execution.

Pending Verification Scenarios:

- [VS1.2.1] The Device Client uploads data from the device to the Serverless Runtime and receives an acknowledgement.
- [VS1.2.4] The Device Client requests transfer data from external resources to the Serverless Runtime and receives an acknowledgement.

SR1.3 Programming languages

Status: IN PROGRESS

Description: The current implementation of the Device Client only supports Python. The tested verification scenarios for the Python language are described above, at the SR1.2 section.

Completed Verification Scenarios:

- [VS1.3.2] Test previously described validation scenarios implemented in Python language.

Pending Verification Scenarios:

- [VS1.3.1] Test previously described validation scenarios implemented in C language — *Only VS1.1.3 equivalent is pending.*

SR1.4 Low memory footprint for constrained devices

Status: IN PROGRESS

Pending Verification Scenarios:

- [VS1.4.1] Test validation scenarios described above on a device with less than 500kB of RAM.

SR1.5 Security

Status: NOT STARTED

Pending Verification Scenarios:

- [VS1.5.1] The Device Client asks the Provisioning Engine for a Serverless Runtime with the data encrypted and signed and the request is accepted.
- [VS1.5.2] The Device Client asks the Provisioning Engine for a Serverless Runtime without the data encrypted or signed and the request is refused.

9.2 Serverless Runtime

SR2.1 Secure and Trusted FaaS Runtimes

Status: IN PROGRESS

Description: The current COGNIT Framework is able to orchestrate on-demand VMs containing Serverless Runtimes through the Provisioning Engine. Currently the access to this API is not secured. In addition, the execution of the FaaS functions can be tested standalone. This component has been developed with a set of unit tests[29] that can be executed to validate that the component is able to execute and return the result of Python functions.

Completed Verification Scenarios:

- [VS2.1.1] Build and instantiate a FaaS Runtime image for Python language and test the execution of a function using a secure communication channel.
- [VS2.1.2] Build and instantiate a FaaS Runtime image for C language and test the execution of a function using a secure communication channel.

SR2.2 Secure and Trusted DaaS Runtimes

Status: NOT STARTED

Pending Verification Scenarios:

- [VS2.2.1] Build and instantiate a DaaS Runtime image for SQL DB (e.g. MariaDB) and test uploading and copying data using a secure communication channel.
- [VS2.2.2] Build and instantiate a DaaS Runtime image for Object Storage (e.g. MinIO) and test uploading and copying data using a secure communication channel.

9.3 Provisioning Engine

SR3.1 Provisioning Interface for the Device to manage Serverless Runtimes

Status: IN PROGRESS

Description: Provisioning Engine tests have been extended to incorporate Serverless Runtime update tests, and as such fulfil the verification scenario 3.1.3. These tests are specially relevant since they can cause actions in the AI-Enabled Orchestrator. While these scheduling actions are not tested, the new integration tests check that the appropriate rescheduling flag is set on update of a Serverless Runtime, so we can ensure the AI-Enabled Orchestrator is properly informed.

[29] https://github.com/SovereignEdgeEU-COGNIT/serverless-runtime/tree/main/app/test

Completed Verification Scenarios:

- [VS3.1.1] A YAML file with the device requirements is provided to the Provisioning Engine and it returns the Serverless Runtime ID.
- [VS3.1.2] Query the Provisioning Engine to return the status of a Serverless Runtime identified by its ID.
- [VS3.1.3] A YAML file with the updated device requirements is provided to the Provisioning Engine that updates the associated Serverless Runtime.
- [VS3.1.4] Delete a Serverless Runtime providing its ID.

9.4 Cloud-Edge Manager

SR4.1 Provider Catalog

Status: NOT STARTED

Pending Verification Scenarios:

- [VS4.1.1] Listing the providers belonging to the Provider Catalog.
- [VS4.1.2] Filtering the providers according to a desired latency threshold on a geographic area.
- [VS4.1.3] Filtering the providers according to a cost per hour threshold.
- [VS4.1.4] Filtering the providers according to energy consumption per hour threshold.
- [VS4.1.5] Filtering the providers according to some specific hardware characteristics (e.g. GPUs, Trusted Execution Environments).

SR4.2 Edge Cluster Provisioning

Status: NOT STARTED

Pending Verification Scenarios:

- [VS4.2.1] A YAML file containing the information about the provision is provided to the Cloud-Edge Manager that creates a new Edge Cluster.
- [VS4.2.2] Query the Cloud-Edge Manager to return the status of an Edge Cluster identified by its ID.
- [VS4.2.3] Query the Cloud-Edge Manager to scale up/down the number of hosts of an Edge Cluster identified by its ID.
- [VS4.2.4] Query the Cloud-Edge Manager to delete an Edge Cluster identified by its ID.

SR4.3 Serverless Runtime Deployment

Status: IN PROGRESS

Description: A new test for VS4.3.2 was introduced so the new attributes READY_SCRIPT and READY_SCRIPT_PATH are properly tested.

Completed Verification Scenarios:

- [VS4.3.1] A YAML file containing the information about the deployment is provided to the Cloud-Edge Manager that creates a new Serverless Runtime.
- [VS4.3.2] Query the Cloud-Edge Manager to return the status of a Serverless Runtime identified by its ID.
- [VS4.3.4] Query the Cloud-Edge Manager to update the deployment of the Serverless Runtime identified by its ID.
- [VS4.3.5] Query the Cloud-Edge Manager to delete a Serverless Runtime identified by its ID.

Pending Verification Scenarios:

- [VS4.3.3] Query the Cloud-Edge Manager to scale up/down the resources (CPU, memory and disks) of a Serverless Runtime identified by its ID.

SR4.4 Metrics, Monitoring, Auditing

Status: IN PROGRESS

Description: Development is still in process, so there are currently no integration tests that stress the Verification Scenarios. We will be adding integration testing to check the different metrics are available, now they are checked indirectly in the COGNIT testbed.

Pending Verification Scenarios:

- [VS4.4.1] Create an Edge Cluster and deploy a Serverless Runtime and check the metrics collected for a certain period of time.

SR4.5 Authentication & Authorization

Status: IN PROGRESS

Description: Development is still in process, so there are currently no integration tests that stress the Verification Scenarios.

Pending Verification Scenarios:

- [VS4.5.1] Test the creation of new users and groups.
- [VS4.5.2] Assign ACLs to designated users and test the creation of new Edge Clusters and Serverless Runtimes.
- [VS4.5.3] Communicate with Provisioning Engine using a secure channel.

9.5 AI-Enabled Orchestrator

SR5.1 Building Learning Model

Status: IN PROGRESS

Description: Development is still in process, so there are currently no integration tests that stress the Verification Scenarios. However, a machine learning server has been set up to store and train models, where a number of models are developed for two tasks. The first task is to characterise the workloads to complement the second task on interference-aware scheduling.

Pending Verification Scenarios:

- [VS5.1.1] List instances from Devices to Applications to System for metrics to be collected.
- [VS5.1.2] Correlate and represent features that are ready to take as input to the Model.
- [VS5.1.3] Feedback-aware performance check when training the model on represented features.
- [VS5.1.4] Assess the ability in terms of AUROC score for each task (e.g. scheduling).

SR5.2 Smart Deployment of Serverless Runtimes

Status: IN PROGRESS

Description: Development is still in process, so there are currently no integration tests that stress the Verification Scenarios.

Pending Verification Scenarios:

- [VS5.2.1] Users Quality of Service (QoS) / Quality of Experience (QoE) will check for each AI-enabled Orchestrator decision for deployment of Serverless Runtimes.

SR5.3 Scheduling Mechanisms

Status: IN PROGRESS

Description: The refactor of the OpenNebula Scheduler to request placement suggestions from an external module is currently tested in the context of the upstream OpenNebula QA process. These tests need an external scheduler implementing a REST API conforming to the requirements of the AI-enabled Orchestrator; for this end a mockup AI-enabled Orchestrator has been implemented in the QA process of OpenNebula, implementing a simple round-robin scheduling algorithm.

Completed Verification Scenarios:

- [VS5.3.1] The Cloud-Edge Manager must allocate Virtual Machines to hosts when an external scheduler is configured.
- [VS5.3.2] The external scheduler must receive a request whenever there are Virtual Machines in pending states.

Pending Verification Scenarios:

- [VS5.3.3] The external scheduler must receive a request for replacement whenever there are Virtual Machines in running states.

9.6 Secure and Trusted Execution of Computing Environments

SR6.1 Advanced Access Control

Status: IN PROGRESS

Description: Development is still in process, so there are currently no integration tests that stress the Verification Scenarios.

Pending Verification Scenarios:

- [VS6.1.1] Define a security policy based on geographic zone attributes.
- [VS6.1.2] Check enforcement of new security policy when edge device moves closer from one edge node than another.

SR6.2 Confidential Computing

Status: IN PROGRESS

Description: Development is still in process, so there are currently no integration tests that stress the Verification Scenarios.

Pending Verification Scenarios:

- [VS6.2.1] Deploy a function on a host that provides confidential computing capability.

- [VS 6.2.2] Check that the function is executed inside the host trusted execution environment (TEE).

SR6.3 Federated Learning

Status: NOT STARTED

Pending Verification Scenarios:

- [VS6.3.1] Perform training of the ML algorithm without exchanging local data.
- [VS6.3.2] Check that the redistributed models for inference do not contain private data.

10. Conclusions and Next Steps

On the basis of the first two versions of the Use Cases Scientific Report (Deliverables D5.1 and D5.2), this third version provides an overview of the overall status of the integration of the use cases with the COGNIT Framework. It provides further details about the research and technology developments that have been achieved by the use cases in the Second Research & Innovation Cycle (M10-M15). Finally, this report provides an update on the integration process and infrastructure for the first version of the integrated COGNIT software stack (see Deliverable D5.7) and its demonstration in the testbed environment of the Project (see Deliverable D5.10), and on the progress of the software requirement verification tasks per component.

This report complements the Project's global overview provided by Deliverable D2.3, as well as the component-specific research and development activities reported in Deliverables D3.2, D3.7, D4.2, and D4.7.

Additional incremental versions of this report will be released at the end of each research and innovation cycle (i.e. M21, M27, M33).