A Cognitive Serverless Framework for the Cloud-Edge Continuum

D3.4 COGNIT FaaS Model - Scientific Report - d

Version 1.0

30 April 2025

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 public providers and on-premises data centers in the Cloud-Edge Continuum. The main goal of this project is the automatic and intelligent adaptation of those resources 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 describes the research and development carried out in WP3 “Distributed FaaS Model for Edge Application Development” during the Fourth Research & Innovation Cycle (M22-M27), providing details on the status of a number of key components of the COGNIT Framework (i.e. Device Client, COGNIT Frontend, and Edge Cluster) as well as reporting the work related to supporting the Secure and Trusted Execution of Computing Environments.

Copyright © 2023 SovereignEdge.Cognit. All rights reserved.

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

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

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:	WP3. Distributed FaaS Model for Edge Application Development
Nature:	R: Report
Dissemination Level:	PU: Public
Version:	1.0
Contractual Date of Delivery:	31/03/2025
Actual Date of Delivery:	30/04/2025
Lead Author:	Idoia de la Iglesia (Ikerlan)
Authors:	Monowar Bhuyan (UMU), Malik Bouhou (CETIC), Aritz Brosa (Ikerlan), Cristina Cruces (Ikerlan), Martxel Lasa(Ikerlan), Jean Lazarou (CETIC), Fátima Fernández (Ikerlan), Aitor Garciandia (Ikerlan), Torsten Hallmann (SUSE), , Philippe Massonet (CETIC), Nikolaos Matskanis (CETIC), Deins Darquennes (CETIC), Mikel Irazola (Ikerlan), Álvaro Puente (Ikerlan), Thomas Ohlson Timoudas (RISE), Paul Townend (UMU), Iván Valdés (Ikerlan), Alejandro Mosteiro (OpenNebula), Mikalai Kutouski (OpenNebula), Michal Opala (OpenNebula), Marco Mancini (OpenNebula).
Status:	Submitted

Document History

Version	Issue Date	Status[1]	Content and changes
0.1	22/04/2025	Draft	Initial Draft
0.2	24/04/2025	Peer-Reviewed	Reviewed Draft
1.0	30/04/2025	Submitted	Final Version

Peer Review History

Version	Peer Review Date	Reviewed By
0.2	23/04/2025	Yashwant Singh Patel (UMU)
0.2	24/04/2025	Antonio Álvarez (OpenNebula)

Summary of Changes from Previous Versions

First Version of Deliverable D3.4

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

Executive Summary

This is the fourth “COGNIT FaaS Model - Scientific Report” that has been produced in WP3 “Distributed FaaS Model for Edge Application Development”. It describes in detail the progress of the software requirements stated in deliverable D2.4 that have been active during the Fourth Research & Innovation Cycle (M22-M27) in connection with these main components of the COGNIT Framework:

Device Client

- **SR1.1** Interface with COGNIT Frontend
  Implementation of the communication of the Device Client with the COGNIT Frontend.
- **SR1.2** Interface with Edge Cluster
  Implementation of the communication of Device Client with the Edge Cluster.
- **SR1.3** Programming languages
  Support for different programming languages.
- **SR1.4** Low memory footprint for constrained devices.
- **SR1.5** Security
  Device Runtime must be secured.
- **SR1.6** Collecting Latency Measurements
  Latency measurements against Edge Clusters should be acquired by the Device Client.

COGNIT Frontend

- **SR2.1** COGNIT Frontend
  Provides an entry point for devices to communicate with the COGNIT Framework for offloading the execution of functions and uploading global data.

Edge Cluster

- **SR3.1** Edge Cluster Frontend
  The Edge Cluster must provide an interface (Edge Cluster Frontend) for the Device Client to offload the execution of functions and to upload local data that is needed to execute the function.
- **SR3.2** Secure and trusted Serverless Runtimes
  The Serverless Runtime is the minimal execution unit for the execution of functions offloaded by Device Clients.

Secure and Trusted Execution of Computing Environments

- **SR6.1** Advanced Access Control
  Implement policy-based access control to support security policies on

geographic zones that define a security level for specific areas.

- **SR6.2** Confidential Computing
  Enable privacy protection for the FaaS workloads at the hardware level using Confidential Computing (CC) techniques.

This deliverable has been released at the end of the Fourth Research & Innovation Cycle (M27), and in M33, at the end of the Fifth Research and Innovation Cycle D3.5 will be released and, being the final version, will be standalone.

Table of Contents

Abbreviations and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CFE	COGNIT Frontend Engine
CC	Confidential Computing
DaaS	Data as a Service
ECFE	Edge Cluster Frontend Engine
FaaS	Function as a Service
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	Javascript Object Notation
MITM	Man In the Middle attack
MTM	Microsoft Threat Modeling
REST	Representational State Transfer
SLA	Service Level Agreement
SR	Serverless Runtime (with no number)
SRx	Software Requirement (with a number associated, e.g.: SR1.1)
SSL	Secure Sockets Layer
TLS	Transport Layer Security
VM	Virtual Machine
VPN	Virtual Private Network

1. Device Client

The Device Client uses the COGNIT library, which exposes the `DeviceRuntime` class. This release improves upon previously developed methods from this class and enhances the system's internal operation. In addition, new methods have been included to extend the functionality of the DeviceRuntime. As of this release, the `DeviceRuntime` class provides five methods through which the client device can interact with the COGNIT Framework.

The first two methods are the `init()` and `stop()` functions. In previous versions, `init()` just initialised the communication with COGNIT by means of authentication and exchange of requirements. Now, `init()` launches a thread where the Device Runtime State Machine, developed in the previous version, runs by its own means. In other words, once the Device Runtime State Machine is launched, it will not only authenticate and exchange requirements, it will be prepared to handle every type of situation that the communication will require. By contrast, the `stop()` method shutdowns the Device

Runtime State Machine, meaning that the user will not be able to offload functions once this method is called.

The `call()` and `call_async()` methods from the `DeviceRuntime` class can be leveraged by the user to offload functions into COGNIT by passing the function and its arguments. The `call()` function blocks the calling thread until it receives the results from COGNIT whereas `call_async()` function will call the provided callback once the result arrives. Both functions enqueue function information into a buffer called the `call_queue`. The Device Runtime State Machine, responsible for managing communication between the Device Runtime and the Cognit framework, runs on a separate thread from the one issuing offloading commands launched or stopped by the `init()` and `stop()` functions, respectively. Once the Device Runtime State Machine reaches the `READY` state, it processes the enqueued functions sequentially, offloading them to the Cognit Framework.

Although the `call_async()` function allows users to initiate non-blocking executions, the Device Runtime State Machine itself processes functions synchronously, meaning it will not execute a new function until it receives the result from the previous one. This ensures orderly execution while maintaining an asynchronous interface for the user. In order to provide a full asynchronous function offloading service, it will be needed between the Device Client and COGNIT Framework to have a mechanism to poll the status of all the different functions offloaded, as well as an endpoint to offload functions without waiting for the result. As of now, the Device Client is only able to reach the endpoint where the functions are executed following the queue order.

Finally, the `update_requirements()` function will notify the Device Runtime State Machine that the application requirements that are used for the communication between the Device Runtime and COGNIT have changed. This will trigger some actions in the Device Runtime State Machine to readapt the communication between the Device Client and COGNIT.

It is also worth noting that the Device Runtime State Machine is also prepared to send periodically to the COGNIT Framework the latency between the Device Runtime and Edge Cluster Frontend. However, it is not fully integrated in the system as it needs to be adapted to be fully compliant with Edge Cluster Frontend's needs to handle properly these metrics, that's why the concerned Software Requirement (SR1.6) is in progress and not completed, this will be tackled in the next development cycle.

The Application Programming Interface (API) definition for the `DeviceRuntime` can be summarized in the following methods:

Description	Method	Parameters	Return Type
Initializes the state machine that handles the communication with the COGNIT Framework with specific requirements.	`init()`	A dict python object containing the application requirements.	bool
Stops the state machine that handles the communication with the COGNIT Framework.	`stop()`	None	bool
Updates the current requirements that are used during the communication with COGNIT Framework.	`update_requirements()`	A dict python object containing the new application requirements.	bool
Offloads a function blocking the calling thread until the results are given by the COGNIT framework.	`call()`	Function to be offloaded (Callable type) Parameters to be used in the function. (Tuple type, optional)	ExecResponse object containing the result of the execution or the error in case it was not executed correctly.

Offloads a function without blocking the calling thread. The callback will be called once the results are given by the COGNIT framework	`call_async()`	Function to be offloaded (Callable type) Callback to be called when the function execution finishes (Callable type with the structure) Parameters to be used in the function. (Tuple type, optional)	bool

Table 1.1. API definition of the `DeviceRuntime` class

System architecture

In this new version, the Device Runtime State Machine operates in a separate thread. As illustrated in Figure 1.1, unlike the previous version, where interactions occurred directly with the Device Runtime State Machine, this version introduces a handler that manages all transitions and events that influence the behaviour of the state machine.

The primary events managed by the handler are:

- Initialization of the State Machine: Upon initialization, a thread is launched where the state machine is continuously evaluated to determine the appropriate state transition.
- Shutdown of the State Machine: The loop executed by the thread is terminated by setting a flag that controls the running state. This flag allows the handler to exit or re-enter the loop.
- Changing Requirements: The handler can receive commands to modify the state machine's requirements. When this occurs, the handler updates a flag that indicates whether there are pending requirements to be processed. Once this parameter changes, the Device Runtime State Machine transitions to the state responsible for handling the requirement modification.

When calling the `call()` and `call_async()` functions, the Device Runtime library does not interact with the Device Runtime State Machine handler. These functions convert the information provided by the user into a `Call` Object and enqueue it to the `call_queue` buffer. The Device Runtime State Machine, once in the ready state (which requires prior

initialization), retrieves these `Call` Objects and uses them to offload the functions to COGNIT.

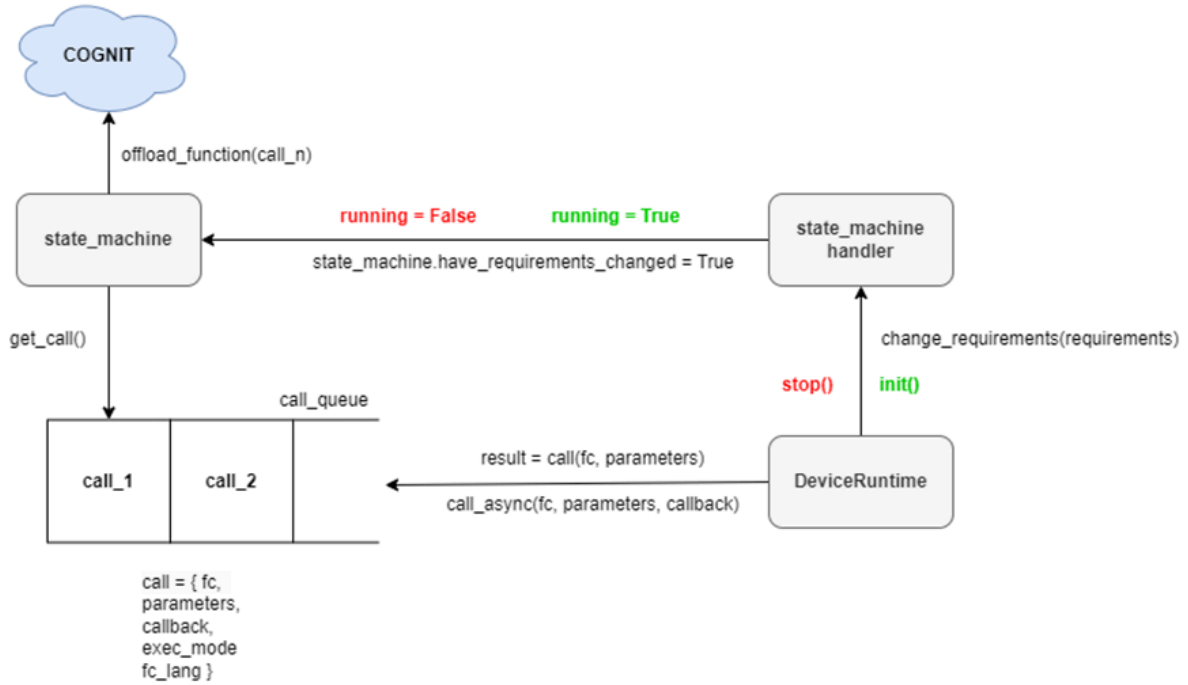

Figure 1.1. Internal system description

Components of the Device Runtime State Machine

The Device Runtime State Machine has not changed from previous versions. It is composed of a set of four states. Each of these states represents a particular situation in the communication process between the Device Client and the COGNIT framework. The different states work as follows:

- `INIT`. This is the initial state of the communication process, where the user has not yet been authorised on the Frontend.
- `SEND_INIT_REQUEST`. Once the user is authorised, they are permitted to upload offloading requirements for their functions. This state handles that process.
- `GET_ECF_ADDRESS`. After successfully uploading the requirements, the system transitions to this state, where it waits for and requests the address of the Edge Cluster (ECF) that will handle the user's requests.
- `READY`. If all prior steps are completed, the state machine enters the `READY` state, indicating it is prepared to offload the user's client functions to the most suitable virtual machine based on the requirements.

Although the states and transitions between them have not changed, the way the `READY` state offloads the functions changes as the way of receiving the function changes.

The transition from one state to another does not follow a linear path, and it always depends on the particular situation of the communication. These situations are tracked by the state machine using variables that are checked before a transition is produced. In this

manner, the path to the states will be different depending on the value of these variables. Some of the different situations that are tracked are:

- A successful or not authentication from the user.
- The correct uploading of the requirements to the COGNIT Framework.
- A continuous track of the state of the communication session with the Frontend clients.
- The obtention of a correct address to the Edge Cluster attendant.

The only difference between previous versions is that on top of the state machine, there is now a state machine handler that oversees the launch and management of the transitions in the Device Runtime State Machine.

Data Model of the Device Runtime State Machine

When calling the `call()` and `call_async()` functions, the Device Runtime library does not interact with the Device Runtime State Machine handler. These functions convert the information provided by the user into a `Call` Object and enqueue it to the `call_queue` buffer. The Device Runtime State Machine, once in the `READY` state (which requires prior initialization), retrieves these `Call` Objects and uses them to offload the functions to COGNIT.

Three different data structures are used in communication with the Device Runtime State Machine:

1. `ExecResponse` Model: Used to communicate the function execution result from COGNIT. It is the data structure retrieved when calling `call()` via the function result and `call_async()` via the argument of the callback function.
2. `Scheduling` Model: This structure is used to share requirements with the Device Runtime State Machine when the `update_requirements()` or `init()` functions are called.
3. `Call` Model: The newest model incorporated in this communication. A `Call` object of this model is created every time `call()` or `call_async()` is called. By passing this function information to the queue, it is then retrieved by the Device Runtime State Machine to offload and execute functions.

An overview of these three models is shown as follows:

Attribute	Description	Fields	Type
`ExecResponse`	Response of a generic execution, with its return code, result and error if applicable.	`ret_code`: `ExecReturnCode` `res`: str (Optional) `err`: str (Optional)	Inherits from pydantic[2]'s BaseModel
`Scheduling`	String describing the policy applied to scheduling. e.g.: "energy, latency" will optimise the placement according to those two criteria.	`POLICY`: str `REQUIREMENTS`: str	Inherited from pydantic's BaseModel

`Call`	Internal object model expected by `call_queue` to offload functions.	`function`: Callable
		`fc_lang`: `FunctionLanguage`
		`callback`: Callable (optional)
		`mode`: `ExecutionMode`
		`params`: List[Any]
			Inherited from pydantic's BaseModel

Table 1.2. Data Model defining the Device Client's interaction with the Device Runtime State Machine.

1.1. [SR1.1] Interface with COGNIT Frontend

Description

The COGNIT Frontend Client is an integral component of the Device Runtime, facilitating interaction with the COGNIT Frontend Engine.

This client provides several key functionalities. Firstly, it supports the uploading, updating, and deletion of user-defined application requirements. Secondly, it enables the uploading of functions. Lastly, it facilitates the retrieval of the most optimal COGNIT Edge Cluster Frontend Engine (ECFE) endpoint for task offloading.

The COGNIT Frontend Client employs an internal flag `has_connection` to indicate the current connection status, allowing other modules within the Device Runtime to check this flag. Additionally, it maintains an internal variable that maps uploaded functions to the IDs

[2] https://docs.pydantic.dev/latest/

assigned by COGNIT, thereby minimising the processing overhead associated with re-uploading existing functions.

Data model

The data model of the interaction with the COGNIT Frontend Engine defines all the fields expected by the COGNIT Frontend Engine for requests and responses.

The `UploadFunctionDaaS` data model follows the structure defined in `ExecSyncParams` of the previous version.

Attribute	Description	Fields	Type
`Scheduling`	Object containing the information of the application requirements.	FLAVOUR: str MAX_LATENCY: int (optional) MAX_FUNCTION_EXECUTION_TIME: float (optional) MIN_ENERGY_RENEWABLE_USAGE: int (optional) GEOLOCATION: str (required if MAX_LATENCY is defined, optional otherwise)	Inherited from pydantic's BaseModel
`FunctionLanguage`	String defining the language of the offloaded function.	`PY` = "PY" `C` = "C"	Enum
`UploadFunctionDaaS`	Object containing the information (language, function, and hash) about the function to be uploaded.	`LANG`: FunctionLanguage `FC`: str `FC_HASH`: sr	Inherited from pydantic's BaseModel

`EdgeClusterFrontendResponse`	Object containing the information about the optimal Edge Cluster obtained from the COGNIT Frontend Engine.	`ID`: int `NAME`: str `HOSTS`: list[int] `DATASTORES`: list[int] `VNETS`: list[int] `TEMPLATE`: dict	Inherited from pydantic's BaseModel

Table 1.3. Data model of the COGNIT Frontend Client component**.**

API & Interfaces

The COGNIT Frontend Client is composed of several private methods, as depicted in Table 1.4, which are abstracted from the user and are used to interact with the COGNIT Frontend Engine.

Description	Method	Parameters	Return Type
Used to delete the application requirements using the id stored as a class variable.	`_app_req_delete`	None	Bool indicating if the app requirement has been deleted.
Get the application requirements using the id stored as a class variable.	`_app_req_read`	None	`Scheduling` type object containing the information about the application requirements.
Update the application requirements using the id stored as a class variable.	`_app_req_update`	`new_reqs`: `Scheduling`	Bool indicating if the app requirement has been updated.

Authenticates the device with the ECFE to get a valid biscuit token. Uploads the initial application requirements to the DaaS gateway.	`_authenticate`	None	`biscuit_token` : str
Check if GEOLOCATION field has a value for the cases that MAX_LATENCY is defined	`_check_geolocation_valid`	`reqs`: `Scheduling`	`is_valid`: bool
Get the endpoint of the optimal Edge Cluster.	`_get_edge_cluster_address`	None	String containing the endpoint.
Serialize and upload the function. Add the uploaded function id to the class variable map.	`upload_funtion_to_daas`	`func`: Callable	Int
Perform the upload of the serialised function.	`_upload_fc`	`fc`: `UploadFunctionDaas`	func_id: int

Getter for the has_connection flag.	`get_has_connection`	None	Bool indicating if the CFE Client has a connection with the COGNIT Frontend.
Perform the authentication and upload the initial app requirements	`init`	`initial_reqs`: `Scheduling`	Bool indicating the status of the initialization.
Setter for has_connection flag.	`set_has_connection`	`new_value`: bool	None

Table 1.4. API definition of the Cognit Frontend Client component.

1.2 [SR1.2] Interface with Edge Cluster

Description

The Edge Cluster Client is the component from the Device Runtime that manages the entire communication process with the Edge Cluster. The Edge Cluster serves as an intermediary between the Device Client and the Serverless Runtimes (via the Edge Cluster) that run across the Cloud-Edge Continuum. Therefore, to interact with these Serverless Runtimes, the Edge Cluster Client must be capable of executing certain directives from the Edge Cluster, which will, in turn, affect the Serverless Runtimes.

As well as in the previous deliverable, there is only one directive that interacts with the Edge Cluster. This directive can be accessed through the REST API incorporated into the Edge Cluster if a correct authentication was properly done in the COGNIT Frontend. This directive allows the synchronous retrieval of the result of a function execution, previously uploaded, by supplying the arguments and the identification of the corresponding function.

This endpoint is used not only in the `call()` function but also in the `call_async()` function. This is the reason why `call_async()` functions preserve the order of execution and only provide an asynchronous interface for the user. Future works can focus on a different endpoint for this type of execution.

Architecture & components

As well as it occurs in the COGNIT Frontend, the Edge Cluster Frontend employs an internal flag has_connection to indicate the current connection status, allowing other modules within the Device Runtime to check this flag.

The following table summarize the methods developed for the Edge Cluster Frontend Client:

Description	Method	Parameters	Return Type
Executes a function with `func_id` in the COGNIT Framework	`execute_function()`	`func_id`: int `app_req_id`: int `exec_mode`: ExecutionMode `callback`: Callable `params_tuple`: tuple `timeout`: int (optional)	`ExecResponse`
Sends location and latency metrics to the Edge Cluster Frontend	`send_metrics()`	`location`: str `latency`: int	None
Setter for `has_connection` flag.	`set_has_connection`	`new_value`: bool	None

Table 1.5. API definition of the Edge Cluster Client component.

Data Model

Table 1.6 describes the data model followed by the Edge Cluster Client in order to achieve a successful communication with the Edge Cluster. In other words, the following attributes define all the fields expected by the Edge Cluster:

Attribute	Description	Fields	Type
FunctionLanguage	String describing the language of the offloaded function.	PY = “PY” C = “C”	Enum
ExecutionMode	String describing how the function is going to be executed.	SYNC = “sync” ASYNC = “async”	Enum
ExecReturnCode	String describing if the execution of the function went successfully or not.	SUCCESS = 0 ERROR = -1	Enum
ExecResponse	Information obtained after a synchronous execution of an offloaded function.	ret_code: ExecReturnCode res: str err: str	Inherited from pydantic’s BaseModel

Table 1.6. Data Model followed by the Edge Cluster Client

1.3 [SR1.3] Programming languages

Description

In this development cycle a new version of the Device Runtime for the C language has been developed implementing the new COGNIT architecture. The main difference with the Python version is the serialization of the data. As the C Device Runtime must be able to offload Python functions, a serialization method which enables the correct understanding between the C client and the Python Serverless Runtime had to be implemented. The solution implemented for this requirement has been Protobuf[3].

Architecture & components

Protobuf

Protocol Buffers (Protobuf) is a data serialization framework developed by Google that offers a mechanism to serialize structured data. It is language-neutral and

[3] https://protobuf.dev/

platform-neutral, making it useful for communications between heterogeneous services. Figure 1.2 summarizes how Protobuf works.

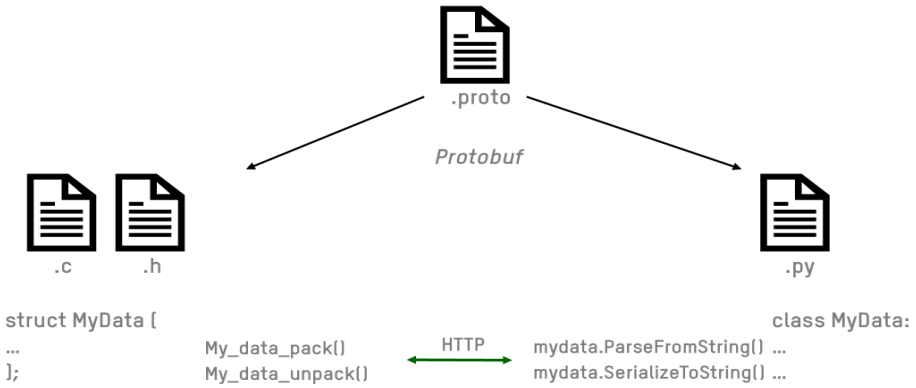

Figure 1.2. Protobuf functioning

The necessary data structures (“messages”) are defined in a .proto file, specifying the fields the messages will contain with an ID, types, and labels that indicate whether a field is optional, required (in earlier versions), or repeated.

Once the .proto file is defined, the Protocol Buffers compiler is used to generate code in the required target programming language. This generated code includes classes and methods for serializing the data to a binary format and deserializing it back into in-memory objects.

In the context of this project we decided to use a specific implementation of Protobuf, which is nanopb[4] is a C implementation of Protocol Buffers specifically designed for resource-constrained devices. Some of its key advantages for resource constrained environments are the optimized memory usage with lightweight structures and the flexibility to use dynamic or static memory.

Classes and Methods

The C Device Runtime offers the following classes and methods to manage the COGNIT library and interact with the COGNIT Platform.

[4] https://github.com/nanopb/nanopb

Class	Description
cognit_config_t	A struct which holds the global configuration of the library, which includes the config to access the COGNIT instance.
cognit_frontend_cli_t	Stores the endpoints of the COGNIT Frontend
edge_cluster_frontend_cli_t	Stores the endpoints of an Edge Cluster Frontend
scheduling_t	Represents the requirements of the application
e_status_code_t	Represents the status code for an offloading. Possible values: ERROR, SUCCESS
device_runtime_t	It stores all the previous structures and needs to be provided to the Device Runtime module

faas_t	Stores the function and parameters to be offloaded

Table 1.7. C Device Runtime classes

Description	Method	Parameters	Return Type
Enables the developer to configure the endpoint, credentials and requirements to connect to the COGNIT Platform instance.	**device_runtime_init**	**device_runtime_t** **cognit_config_t** **scheduling_t** **faas_t**	**e_status_code_t**
Adds the function code string to the FaaS structure	**addFC**	**faas_t** **char***	**void**
Adds a variable as a parameter to the FaaS structure	**addXVar**	**faas_t**	**void**
Adds an array as a parameter to the FaaS structure	**addXArray**	**faas_t**	**void**
Performs the offloading of the function to the Cognit platform and the execution	**device_runtime_call**	**device_runtime_t** **scheduling_t** **faas_t**	**e_status_code_t**

with the parameters		**void****	

Table 1.8. C Device Runtime methods

API & interfaces

Due to the fact that the API and interfaces are the same as in the Python version (shown in D3.3 of M21), we are omitting the same information here for the sake of conciseness.

Device client usage example

Also available on GitHub repository:

```C/C++
#include <stdio.h>
#include "cognit_http.h"
#include <curl/curl.h>
#include <stdlib.h>
#include <string.h>
#include <device_runtime.h>
#include <unistd.h>
#include <cognit_http.h>
#include <logger.h>
#include <ip_utils.h>

// Function to be offloaded.
char* fc_str = "def my_calc(operation, param1, param2):\n"
               "    if operation == \"sum\":\n"
               "        result = param1 + param2\n"
               "    elif operation == \"multiply\":\n"
               "        result = param1 * param2\n"
               "    else:\n"
               "        result = 0.0\n"
               "    return result\n";

size_t handle_response_data_cb(void* data_content, size_t size,
size_t nmemb, void* user_buffer)
{
    size_t realsize           = size * nmemb;
    http_response_t* response = (http_response_t*)user_buffer;

    if (response->size + realsize >=
sizeof(response->ui8_response_data_buffer))
    {
        COGNIT_LOG_ERROR("Response buffer too small");
        return 0;
```

```
    }

    memcpy(&(response->ui8_response_data_buffer[response->size]),
data_content, realsize);
    response->size += realsize;
    response->ui8_response_data_buffer[response->size] = '\0';

    return realsize;
}

int my_http_send_req_cb(const char* c_buffer, size_t size,
http_config_t* config)
{
    CURL* curl;
    CURLcode res;
    long http_code             = 0;
    struct curl_slist* headers = NULL;
    memset(&config->t_http_response.ui8_response_data_buffer, 0,
sizeof(config->t_http_response.ui8_response_data_buffer));
    config->t_http_response.size = 0;

    curl_global_init(CURL_GLOBAL_DEFAULT);

    curl = curl_easy_init();
    if (curl)
    {
        // Set the request header
        headers = curl_slist_append(headers, "Accept:
application/json");
        headers = curl_slist_append(headers, "Content-Type:
application/json");
        //headers = curl_slist_append(headers, "charset: utf-8");

        if (config->c_token != NULL)
        {
            char token_header[MAX_TOKEN_LENGTH] = "token: ";
            strcat(token_header, config->c_token);
            headers = curl_slist_append(headers, token_header);
        }

        if (curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers) !=
CURLE_OK
            // Configure URL and payload
            || curl_easy_setopt(curl, CURLOPT_URL, config->c_url) !=
CURLE_OK
            // Set the callback function to handle the response data
            || curl_easy_setopt(curl, CURLOPT_WRITEDATA,
(void*)&config->t_http_response) != CURLE_OK
```

```
            || curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION,
handle_response_data_cb) != CURLE_OK
            || curl_easy_setopt(curl, CURLOPT_TIMEOUT_MS,
config->ui32_timeout_ms) != CURLE_OK
            || curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 0L) !=
CURLE_OK
            || curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 0L) !=
CURLE_OK)
        {
            COGNIT_LOG_ERROR("[http_send_req_cb] curl_easy_setopt()
failed");
            return -1;
        }

        // Find '[' or ']' in the URL to determine the IP version
        // TODO: fix ip_utils to obtain
http://[2001:67c:22b8:1::d]:8000/v1/faas/execute-sync
        // as IP_V6
        if (strchr(config->c_url, '[') != NULL
            && strchr(config->c_url, ']') != NULL)
        {
            if (curl_easy_setopt(curl, CURLOPT_IPRESOLVE,
CURL_IPRESOLVE_V6) != CURLE_OK)
            {
                COGNIT_LOG_ERROR("[http_send_req_cb]
curl_easy_setopt()->IPRESOLVE_V6 failed");
                return -1;
            }
        }

        if (strcmp(config->c_method, HTTP_METHOD_GET) == 0)
        {
            if (curl_easy_setopt(curl, CURLOPT_HTTPGET, 1L) !=
CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_USERNAME,
config->c_username) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_PASSWORD,
config->c_password) != CURLE_OK)
            {
                COGNIT_LOG_ERROR("[http_send_req_cb]
curl_easy_setopt()->get() failed");
                return -1;
            }
        }
        else if (strcmp(config->c_method, HTTP_METHOD_POST) == 0)
        {
            if (curl_easy_setopt(curl, CURLOPT_POST, 1L) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST,
```

```
"POST") != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE,
size) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_POSTFIELDS,
c_buffer) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_USERNAME,
config->c_username) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_PASSWORD,
config->c_password) != CURLE_OK)
            {
                COGNIT_LOG_ERROR("[http_send_req_cb]
curl_easy_setopt()->post() failed");
                return -1;
            }
        }
        else if (strcmp(config->c_method, HTTP_METHOD_PUT) == 0)
        {
            if (curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "PUT")
!= CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE,
size) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_POSTFIELDS,
c_buffer) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_USERNAME,
config->c_username) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_PASSWORD,
config->c_password) != CURLE_OK)
            {
                COGNIT_LOG_ERROR("[http_send_req_cb]
curl_easy_setopt()->put() failed");
                return -1;
            }
        }
        else if (strcmp(config->c_method, HTTP_METHOD_DELETE) == 0)
        {
            if (curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST,
"DELETE") != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_USERNAME,
config->c_username) != CURLE_OK
                || curl_easy_setopt(curl, CURLOPT_PASSWORD,
config->c_password) != CURLE_OK)
            {
                COGNIT_LOG_ERROR("[http_send_req_cb]
curl_easy_setopt()->post() failed");
                return -1;
            }
        }
        else
```

```
        {
            COGNIT_LOG_ERROR("[http_send_req_cb] Invalid HTTP
method");
            return -1;
        }

        // Make the request
        res = curl_easy_perform(curl);

        curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &http_code);
        COGNIT_LOG_INFO("HTTP err code %ld ", http_code);

        // Check errors
        if (res != CURLE_OK)
        {
            long http_code = 0;
            curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE,
&http_code);
            COGNIT_LOG_ERROR("curl_easy_perform() failed: %s",
curl_easy_strerror(res));
            COGNIT_LOG_ERROR("HTTP err code %ld ", http_code);
        }

        // Clean and close CURL session
        curl_easy_cleanup(curl);
    }

    config->t_http_response.l_http_code = http_code;

    // Clean global curl configuration
    curl_global_cleanup();
    curl_slist_free_all(headers);

    return (res == CURLE_OK) ? 0 : -1;
}

cognit_config_t t_config = {
    .cognit_frontend_endpoint   =
"https://cognit-lab-frontend.sovereignedge.eu",
    .cognit_frontend_usr        = "", // Put your username here.
    .cognit_frontend_pwd        = "", // Put your password here.
};

// Set your own App requirements.
scheduling_t app_reqs = {
    .flavour                    = "FaaS_generic_V2", // Put a
Flavour that your username is allowed to use.
    .max_latency                = 100,               // Max latency
```

```
required in miliseconds.
    .max_function_execution_time = 3.5,              // Max
execution time required in seconds.
    .min_renewable               = 85,               // Minimal
renewable energy resources required in percentage.
    .geolocation                 = "IKERLAN ARRASATE/MONDRAGON 20500"
};

// Set your new App requirements.
scheduling_t new_reqs = {
    .flavour                     = "FaaS_generic_V2", // Put a
Flavour that your username is allowed to use.
    .max_latency                 = 80,               // Max latency
required in miliseconds.
    .max_function_execution_time = 8.5,              // Max
execution time required in seconds.
    .min_renewable               = 50,               // Minimal
renewable energy resources required in percentage.
    .geolocation                 = "IKERLAN ARRASATE/MONDRAGON 20500"
};

int main(int argc, char const* argv[])
{
    device_runtime_t t_my_device_runtime;
    faas_t t_faas;
    float* exec_response;
    e_status_code_t ret;

    device_runtime_init(&t_my_device_runtime, t_config, app_reqs,
&t_faas);

    addFC(&t_faas, fc_str);

    addSTRINGParam(&t_faas, "sum");
    addINT32Var(&t_faas, 8);
    addFLOATVar(&t_faas, 3.5);

    ret = device_runtime_call(&t_my_device_runtime, &t_faas,
new_reqs, (void**)&exec_response);    if (ret == E_ST_CODE_SUCCESS)
    {
        COGNIT_LOG_INFO("Result: %f", *exec_response);
    }
    else
    {
        COGNIT_LOG_ERROR("Error offloading function");
    }
    return 0;
}
```

1.4 [SR1.4] Low memory footprint

Description

In order to comply with this Software Requirement, an analysis of the memory consumption of the C client was carried out. One of the measures taken to reduce the memory usage was the choice of using nanopb instead of protobuf-c for the serialization of the data. Some tests were done to analyse the memory usage of both libraries and the results obtained can be summarized by the following data:

- Size of the code of protobuf-c[5] + code generated by protobuf-c with the defined messages = 98KB

- Size of the code of nanopb + code generated by nanopb with the define messages = 45KB

Regarding the memory usage in execution, a program was implemented in C which performed data serialization using both libraries. A series of arrays were defined for serialization and the memory used in the process was checked. The results were:

- Memory usage with protobuf-c: It varies depending on the number of parameters defined, as it uses dynamic memory.

- Memory usage with nanopb: 7 KB

As a result, in the tests performed with the C client, the memory usage of the whole library was.

- Code size: 200 KB

- Memory usage: 120 KB

1.5 [SR1.5] Security

Description

In this development cycle, the integration of the Biscuit token has been integrated to enhance the authorization procedures of COGNIT. This integration would also allow an

[5] https://github.com/protobuf-c/protobuf-c

even more advanced authorization scheme with the inclusion of a Keycloak server that makes use of the token.

Architecture & components

In order to authenticate against the COGNIT Frontend, and hence to the whole COGNIT Framework, included in the COGNIT Frontend client's (see SR1.1 of this component) private API there is a specific method that allows the whole Device Client to be authenticated to the platform.

Furthermore, in the Device Runtime State Machine implemented on this version, there are mechanisms to ensure all time that the Device Client is authenticated and otherwise make sure that it tries to authenticate before performing any other action towards the COGNIT platform.

API & interfaces

Method	Description	Arguments	Return Type
`_authenticate`	Authenticates against COGNIT Frontend**.**	HTTPBasicAuth with username and password gathered from the configuration file.	Token formatted in a String. Returns None (nothing in Python) in case of authentication error.

Table 1.9. Authentication API with a single method

1.6 [SR1.6] Collecting Latency Measurements

Description

One of the requirements for the Device Runtime is to be able to periodically send the latency between the Edge Cluster Frontend and the Device Client. The Edge Cluster uses this metric to optimise resource allocation, improve load balancing, and ensure a smoother user experience by dynamically adjusting its behaviour based on network conditions.

Architecture & components

The Device Runtime State Machine is the component responsible for establishing communication with the Edge Cluster Frontend. Once connected, it launches a thread that periodically transmits the experienced latency and the Device Client's location at user-defined intervals.

With respect to the previous deliverable, the Edge Cluster Frontend Client has also a method called `send_metrics()`. This method can be used in the future to communicate with the Edge Cluster Frontend to provide the latency between these two components. It also communicates the location of the Device Client. However, this endpoint is not currently developed in the Edge Cluster Frontend, so it cannot be used.

Description	Method	Parameters	Return Type
Sends location and latency metrics to the Edge Cluster Frontend	`send_metrics()`	`location`: str `latency`: int	bool

Table 1.10. API definition of the Edge Cluster Client for latency and location delivery

Device client (Python version) usage examples

Python

```python
import sys
import time
sys.path.append(".")

from cognit import device_runtime

# Functions used to be uploaded
def suma(a: int, b: int):
      return a + b

def mult(a: int, b: int):
      return a * b

# Workload from (7. Regression Analysis) of
#
https://medium.com/@weidagang/essential-python-libraries-for-machine-
learning-scipy-4367fabeba59

def ml_workload(x: int, y: int):
      import numpy as np
      from scipy import stats

      # Generate some data
      x_values = np.linspace(0, y, x)
      y_values = 2 * x_values + 3 + np.random.randn(x)

      # Fit a linear regression model
```

```
      slope,     intercept,     r_value,     p_value,     std_err     =
stats.linregress(x_values, y_values)

      # Print the results
      print("Slope:", slope)
      print("Intercept:", intercept)
      print("R-squared:", r_value**2)
      print("P-value:", p_value)

      # Predict y values for new x values
      new_x = np.linspace(5, 15, y)
      predicted_y = slope * new_x + intercept

      return predicted_y

# Execution requirements, dependencies and policies
REQS_INIT = {
      "FLAVOUR": "SmartCity_ice_V2",
}

REQS_NEW = {
      "FLAVOUR": "SmartCity_ice_V2",
      "MAX_FUNCTION_EXECUTION_TIME": 15.0,
"MAX_LATENCY": 45,
      "MIN_ENERGY_RENEWABLE_USAGE": 75,
      "GEOLOCATION": "IKERLAN ARRASATE/MONDRAGON 20500"
}

def get_result(result):
    print("************************************************")
      print("Sync result: " + str(result))
    print("************************************************")
      return result

try:

      # Instantiate a device Device Runtime
      my_device_runtime                                              =
device_runtime.DeviceRuntime("./examples/cognit-template.yml")
      my_device_runtime.init(REQS_INIT)

      # Synchronous offload and execution of a function
      result = my_device_runtime.call(suma, 17, 5)

    print("------------------------------------------------")
      print("Sum sync result: " + str(result))
    print("------------------------------------------------")

      # Update the requirements
    my_device_runtime.update_requirements(REQS_NEW)

      # Offload asynchronously a function
      my_device_runtime.call_async(suma, get_result, 100, 10)
```

```
      # Offload and execute a function
      result = my_device_runtime.call(mult, 2, 3)

    print("-------------------------------------------------")
      print("Multiply sync result: " + str(result))
    print("-------------------------------------------------")

      # Let's offload a function with wrong parameters
      result = my_device_runtime.call(mult, "wrong_parameter", "3")

    print("-------------------------------------------------")
      print("Wrong result: " + str(result))
      print("-------------------------------------------------")

      # More complex function
      # Offload and execute ml_workload function
      start_time = time.perf_counter()
      result = my_device_runtime.call(ml_workload, 10, 5)


      end_time = time.perf_counter()


print("-------------------------------------------------------------")
      print("Predicted Y: " + str(result))
      print(f"Execution time: {(end_time-start_time):.6f} seconds")

print("-------------------------------------------------------------")

except Exception as e:
      print("An exception has occured: " + str(e))
      exit(-1)
```

2. Edge Cluster

2.1 [SR3.1] Edge Cluster Frontend

Description

The Edge Cluster Frontend is the entry point for offloading functions from the Device Client in each Edge Cluster. It acts as a load balancer to handle the redirection to the correct Serverless Runtime running within the particular Edge Cluster. As stated in the D2.4 Framework architecture document, requirement SR3.1 is satisfied by this component. In this cycle, the architecture of the Edge Cluster Frontend has been changed in order to improve the load balancing and scalability of the COGNIT Framework. The changes that have been implemented in this development cycle are related only to the implementation of the internal architecture of the Edge Cluster Frontend (reported in the following section) without any change to the API used by the device client. The Data Model and API & Interfaces of the Edge Cluster Frontend are the same as reported in Deliverable D3.3.

Architecture

The Edge Cluster Frontend key functionality is to proxy the connection from the Device Clients to a particular Serverless Runtime running in the Edge Cluster to which the Edge Cluster Frontend is bound to. Its main goal will be coherently load balancing the requests coming from the different devices according to the different flavours of Serverless Runtimes.

As shown in Figure 2.1, the Edge Cluster Frontend when receives a request from a device client, creates a new thread that manages the connection with the device itself, sends the request to a queue (‘Execution Request Queue’) corresponding to the Serverless Runtime flavour required by the device, and waits for the result from a temporary queue associated to an Exchange (‘Result Exchange’) before sending the result to the device client.

Each Serverless Runtime will wait for requests available on the ‘Execution Request Queue’ corresponding to its flavour and once there is a request to be served, the selected Serverless Runtime will be in charge of executing the function and will send the result to a temporary queue corresponding to the ‘Result Exchange’ once the function has been executed.

Each Edge Cluster Frontend thread is a producer for the ‘Execution Request Queue’ and a consumer for the result queues. For the result, the thread will consume the result that is related to its request (identified by a request id).

Each Serverless Runtime is a consumer for the ‘Execution Request Queue’ (the dispatch of the requests to a Serverless Runtime is based on a round robin strategy) and a producer for the result queue.

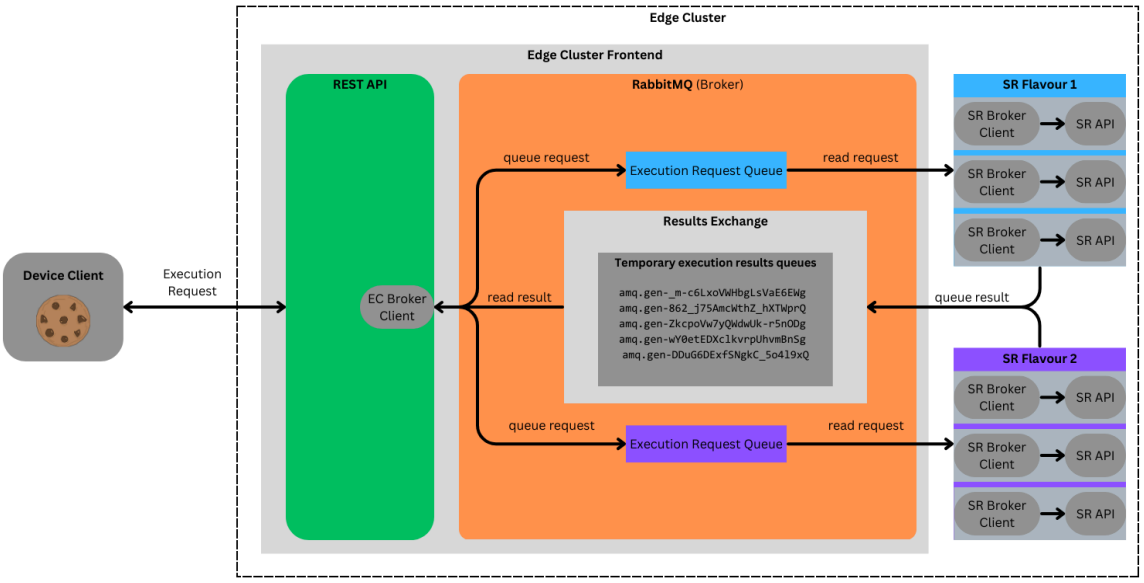

Figure 2.1. Interaction between Device Client and Edge Cluster.

Data model

The same as reported in Deliverable D3.3

API & Interfaces

The same as reported in Deliverable D3.3

2.2 [SR3.2] Secure and Trusted Serverless Runtimes

Description

The current development cycle has introduced new Prometheus metrics to support AI-Enabled Orchestrator decision-making. Additionally, some of the previous metrics have been updated, and all the metrics have been renamed so that they all start with the string "*sr_*", making it easy to identify the metrics coming from the Serverless Runtimes in the COGNIT Prometheus Server. Specifically, the following implementations have been done:

Change 1: Update previous "func_exec_time":

- Modify metric name to "sr_last_func_exec_time"
- Remove label "param_l_0"
- Remove label "param_l_1"
- Add a new label "total_param_size", which shows the total size of the input parameters of the function, in Bytes.

Change 2: Add a new metric gauge for function execution status

- Metric name: "sr_func_status"

- It reflects the current status of the running function. An enum like *{id ↔ state}* map has been implemented to translate a specific state to an integer, so that it can be represented using a Prometheus gauge.
- Labels:
    - “func_hash”: Hash of the function
    - “vm_id”: ID of the Virtual Machine
    - “total_param_size”: represents the total size of the input parameters of the function, in Bytes.
- Possible values:
    - “RUNNING”: 1.0
    - “IDLE”: 0.0

Change 3: Add a new metric counter for executed function number

- Metric name: “sr_func_executed_total”
- Counter that counts the number of functions that have started being executed (regardless of the function result).
- Labels:
    - “vm_id”: ID of the Virtual Machine

Change 4: Add a new metric counter for successful execution number

- Metric name: “sr_func_succeeded_total”
- Counter that counts the number of functions that have been successfully executed.
- Labels:
    - “vm_id”: ID of the Virtual Machine

Change 5: Add a new metric counter for failed execution number

- Metric name: “sr_func_failed_total”
- Counter that counts the number of functions that have failed to be executed.
- Labels:
    - “vm_id”: ID of the Virtual Machine

Change 6: Add new metric histogram for function execution time

- Metric name: “sr_histogram_func_exec_time_seconds”
- Records the execution time of the offloaded functions. This metric is an evolution of the previously defined gauge metric: “sr_last_func_exec_time”. However, the previous gauge metric is still used in this development cycle.
- Labels:
    - “function_outcome”: Either “success” or “error”.
    - “le”: A label of the Prometheus histograms that defines the bucket range.
    - “vm_id”: ID of the Virtual Machine
- Buckets (possible values):
    - time < 1s
    - time < 5s
    - time < 10s

- time > 10s

Change 7: Add new metric histogram for function input size

- Metric name: “sr_histogram_func_input_size_bytes”
- Records the input size of the function parameters
- Labels:
  - “function_outcome”: Either “success” or “error”.
  - “le”: A label of the Prometheus histograms that defines the bucket range.
  - “vm_id”: ID of the Virtual Machine
- Buckets (possible values):
  - size < 1KB
  - size < 1MB
  - size < 1GB
  - size > 1GB

In this development cycle, the Serverless Runtime had to be modified to support the new C Device Runtime. As mentioned, the messages received from the C client will be serialized with Protobuf, so the following changes were made:

- The “LANG” key is checked. A “C” value indicates that the message is coming from a C client
- If the request is coming from a C client, the function and parameters are deserialized by using Protobuf. The files to deserialize and be able to load these data into Python classes are generated by the process described in the section of SR1.3.
- Once the function and the parameters are deserialized the execution of the function is done, and the result is serialized with Protobuf to be returned to the C client.

Architecture & Components

No changes have been made from an architectural point of view.

Data Model

The data model of the Serverless Runtime remains the same as in the previous development cycle

API & Interfaces

There are no changes in the API & Interface of the Serverless Runtime in this development cycle.

3. Secure and Trusted Execution of Computing Environments

3.1 Threat Model

In deliverable D3.3, section 4, a threat assessment has been performed on the framework architecture by creating its threat model. A Microsoft tool called Microsoft Threat Modeling (MTM) has been used for the modelling work.

10 threats had been identified based on the model, and with the help of the MTM tool. These threats had been related to different categories like Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. Attacks can be as different as Collision, Replay, Data Repudiation by COGNIT Frontend, Weak Authentication Scheme, Interruption of Data Flow, Process Crash or Stop, Elevation of Privilege using Remote Code Execution.

This section considers how some of these threats can be exploited through attack flows targeting specific assets of the cybersecurity use case. More specifically, we consider the attacks that can impact the Serverless Runtime. This section will quantify, among all the threats identified in deliverable D2.5, the FaaS threats. It also helps evaluate the associated risks by outlining the attack scenario

Inside the COGNIT Architecture, the Serverless Runtime is responsible for the execution of the function provided by the client. With this central property, it will be the target for a threat analysis. This one will consist of evaluating how a possible attack can impact such a module, and through which specific threats, as listed previously. Figure 3.1 summarizes the COGNIT architecture and the interactions between the Device Client and the COGNIT Framework.

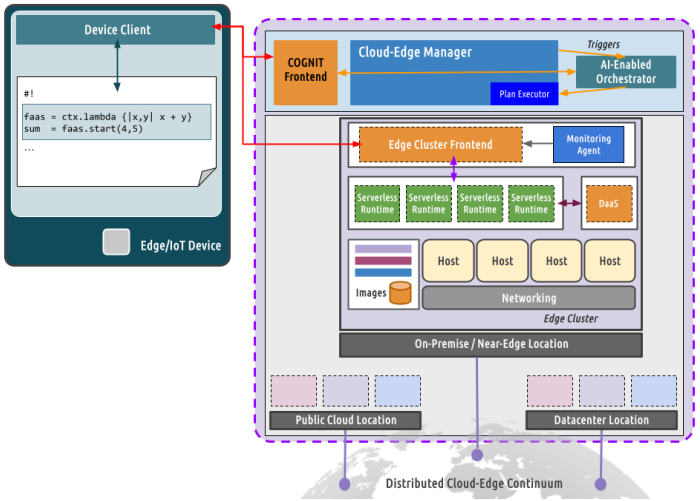

Figure 3.1. Device Client interaction with the COGNIT Platform.

The Device Client could be a source of threats, and the same for a successful implementation of a Man In the Middle attack (MITM) in the communications between the

Device Client and the COGNIT Framework. Figure 3.2 presents the introduction of a MITM attack.

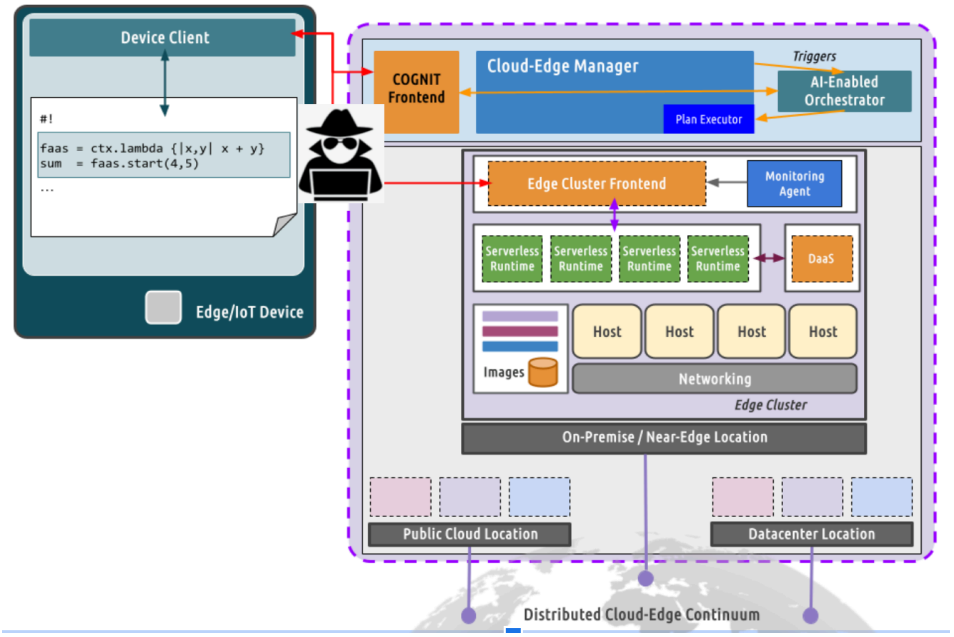

Figure 3.2. Man in the Middle attack

Our approach describes an attack flow, represented in Figure 3.2. It starts from a successful “Man In the Middle" attack between the device and the COGNIT Framework. With this attack between the Device Client and the COGNIT Frontend, exploiting misconfigured VPN or TLS settings can lead to bypass encryption**,** and stealing valid authentication access tokens. The attacker is therefore able:

- to replay the token, to impersonate a legitimate device client, and to access restricted APIs; it can intercept API requests, with a risk of session hijacking and credential theft, allowing unauthorized function execution; the attacker can **spoof a legitimate function’s identity** to access confidential data or invoke other functions; the attacker can **spoof legitimate event data** to manipulate serverless processing;
- to disrupt data communication coming from the client, like metrics for measuring latency against the different edge clusters, **disrupting the AI-enabled orchestrator** to make decisions about the provisioning of Edge Clusters that have to satisfy latency requirements from the application;
- to disrupt the data uploading that can be used by the device functions; unvalidated input can lead to **data leaks**, or **function hijacking**;
- to disrupt the data uploading, that **can lead the serverless function** to process a large dataset or to handle unexpected input sizes; the function is then able to **exceed memory limits of the Serverless runtime** and to crash it, disrupting service continuity;

- to disrupt the data uploading, that can also lead to Denial of Service (DoS), by sending **extremely large inputs**, causing **resource exhaustion** or even crashes, by **forcing repeated failures** and **exhausting retry mechanisms**;
- to disrupt the uploading of application requirements that will be stored and used subsequently by the AI-Enabled Orchestrator for optimising the resources needed by the devices to offload functions related to the application;
- to install a backdoor inside the Serverless runtime, linking to the device client;
- The attacker **spoofs the identity of a trusted service**, tricking the function into accepting malicious requests.

- **Serverless functions rely on event-driven inputs** (API calls, webhooks, message queues), which, if not validated, can be exploited for injection attacks; **Attackers can send malformed or malicious payloads** to bypass security controls and manipulate serverless execution.

- The attacker injects crafted event payloads to execute unauthorized actions.
- A function **relies on another function's response to determine its execution** but lacks safeguards against recursion; the attacker **crafts responses that trigger an endless loop**, leading to resource exhaustion (Denial of Service) of the cluster for other processes.

The following picture synthesizes these consequences of a successful MITM attack, with impact on the Serverless, and the Artificial Intelligence (AI) orchestrator. It models the kill chain on these two assets.

Within this diagram, blue rectangles represent actions expressing the steps of the attack. In this case, the first step is a successful insertion of a MITM. The rectangles in orange represent the asset targeted by a specific previous action.

Or connectors in red indicate that an asset can be followed by different steps of the attack process. Conversely, they also indicate that different actions can concern the same asset.

Grey rectangles indicate the final result and incidence for each specific branch of a kill chain.

Figure 3.3. Attack flow diagram modeling the kill chain in the case of a Man In The Middle Attack.

Figure 3.3 depicts an end-to-end attack flow scenario impacting the Serverless Runtime services, and the AI Orchestrator.

Concerning the Serverless Runtime, the attack starts with compromised communications between the mobile device and the edge cluster with a MITM attack:

1. A valid authentication token is replayed to impersonate a legitimate Device Client, leading to possible disruption of data uploading, or disruption of the uploading of application requirements, or disruption of data communications.
2. Disruption of data uploading or uploading of application requirements can affect serverless processing.

3. Spoofing legitimate function identity, spoofing legitimate event data, and sending large dataset inputs, are actions for disrupting the normal behaviour of the serverless runtime.
4. This can lead to manipulation of serverless processing, that leads to a SLA violation on anomaly detection service.
5. It can also lead to more severe consequences for the serverless, like a crash, a Denial-of-service, a trigger of endless loop leading to resource exhaustion, a forcing of repeated failures exhausting retry mechanisms. Extremely large inputs can also cause resource exhaustion. All of these are able to lead to a failure of anomaly detection.
6. The disruption of data communication coming from the client, like metrics for measuring latency against the different edge clusters affects the AI orchestrator to make decisions about the provisioning of edge clusters that have to satisfy latency requirements from the application.

The impact can lead to loss of service availability, resulting from a crash, repeated failure, or denial of services (DoS). It can also lead to a degradation of service resulting from a manipulation of the functionalities and serverless execution, leading to a deterioration of the normal expected results.

3.2 [SR6.1] Advanced Access Control

Description

This was implemented in the previous cycle, for more details, see Deliverable 3.3

3.3 [SR6.2] Confidential Computing Requirement

Description

The Confidential Computing requirement aims to provide a countermeasure at the edge to be able to process confidential or private data in an edge environment that is exposed to multiple threats and is not inherently trusted.

By leveraging hardware-based Trusted Execution Environments, Confidential Computing enables sensitive data and code to be processed in isolated, secure enclaves that protect against unauthorized access, even from privileged system software. This ensures that critical operations at the edge can maintain confidentiality and integrity, despite operating in potentially vulnerable or untrusted locations. This is particularly relevant in the scenarios that COGNIT Project is examining, where data is collected and processed close to the source, at remote edge nodes, and where centralized security controls may not be feasible or sufficient.

In this context, COGNIT has integrated Confidential Computing capabilities into the framework and initiated the deployment of CC-enabled components within the testbeds. These deployments aim to validate the practical use of CC enclaves for secure data and workload execution at the edge. The Cybersecurity use case will serve as the main scenario

for testing these capabilities, assessing the effectiveness of CC in protecting sensitive operations and data flows.