A Cognitive Serverless Framework for the Cloud-Edge Continuum

D3.5 COGNIT FaaS Model - Scientific Report - e

Version 2.0

1 March 2026

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 standalone document comprehensively describes the research and development carried out across the different tasks in WP3 “Distributed FaaS Model for Edge Application Development” , being the final delivery of WP3. It provides details about key components of the COGNIT Framework such as the Device Client, COGNIT Frontend, and Edge Cluster. Additionally, it reports the work related to supporting the Secure and Trusted Execution of Computing Environments.

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:	WP3. Distributed FaaS Model for Edge Application Development
Nature:	R: Report
Dissemination Level:	PU: Public
Version:	1.0
Contractual Date of Delivery:	30/09/2025
Actual Date of Delivery:	01/03/2026
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	19/12/2025	Draft	Initial Draft
0.2	21/12/2025	Peer-Reviewed	Reviewed Draft
1.0	31/12/2025	Submitted	Final Version
1.1	24/02/2026	Draft	Initial Draft
1.2	26/02/2026	Peer-Reviewed	Reviewed Draft
2.0	01/03/2026	Submitted	Final Version

Peer Review History

Version	Peer Review Date	Reviewed By
0.2	21/12/2025	Antonio Álvarez (OpenNebula)
1.2	25/02/2026	Antonio Álvarez (OpenNebula)
1.2	26/02/2026	Daniele Mingolla (OpenNebula)

Summary of Changes from Previous Versions

Second Version of Deliverable D3.5

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

Executive Summary

This is the fifth “COGNIT FaaS Model - Scientific Report” that has been produced in WP3 “Distributed FaaS Model for Edge Application Development”. As this is the final deliverable for this work package, it will describe in detail all the final work carried out throughout the project (M4-M33) and not the changes with respect to the last deliverable (D3.4) about the following 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.
  Support for low memory footprint on 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.

- **SR6.3** Federating learning
  Enhance privacy of AI workloads that have confidentiality requirements preventing the exchange of information for training.

This deliverable has been released at the end of the Fifth Research & Innovation, which contains 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
CRUD	Create, Read, Update, Delete
DaaS	Data as a Service
DoS	Denial of Service
ECFE	Edge Cluster Frontend Engine
FaaS	Function as a Service
FIFO	First In First Out
HOTP	HMAC-based One-Time Password
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	Javascript Object Notation
LDAP	Lightweight Directory Access Protocol
MITM	Man In the Middle attack
MTM	Microsoft Threat Modeling
OIDC	OpenID Connect
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
TOTP	Time-based One-Time Password
VM	Virtual Machine
VPN	Virtual Private Network
XML-RPC	XML Remote Procedure Call
YAML	Yaml Ain’t a markup language

1. Introduction

COGNIT is a platform that allows its users to execute functions across the edge–cloud continuum in a simple and intuitive way, hiding the underlying complexity. Functions are executed in a serialised manner in a highly dynamic context, where both environmental conditions (network latency, resource availability, device mobility, etc.) and user requirements (performance, energy efficiency, cost, security) change.

COGNIT incorporates adaptive offloading mechanisms, so that functions can be executed in different points on the continuum to optimise the execution experience depending on the context. To support this capability, COGNIT is structured around the following essential components:

- The **Device Client** resides on the user's device. Through the COGNIT library, the user can offload functions to the COGNIT Framework without worrying about communication protocols, resource discovery or dependency management. The library exposes a simple interface that abstracts all the complexity allowing the user to focus solely on defining the requirements needed and the functions to offload.

- The **COGNIT Frontend** acts as the entrypoint to the COGNIT Framework. This component is responsible for authenticating users and authorising access to resources. It receives both the function and the execution requirements. With this information, the COGNIT Frontend will offer to the Device Client the most suitable resource to ensure optimal execution.

- The **Edge Cluster** is the distributed environment that offers computing capabilities in the cloud-edge continuum via Serverless Runtimes. Edge Clusters are made up of different nodes that balance load, isolate executions and ensure fault tolerance.

- **Serverless Runtimes** are the execution units deployed within the Edge Cluster. Each runtime is responsible for receiving, instantiating and executing functions in an isolated environment.

This work package focuses primarily on the design and development of these components, ensuring their seamless integration to provide a transparent user experience capable of adapting to changing environments in real time.

2. Device Client

The Device Client uses the COGNIT library[2], a library built from scratch, through which the Device Runtime class can be extracted. Initializing this class correctly and providing a set of valid requirements, the Device Client will be able to offload functions using a user-friendly interface. Using the COGNIT library, the Device Client does not have to deal with the complexity of the standards and communication with the COGNIT Framework.

This section provides a comprehensive description of the Device Client, detailing its design and role as the component that abstracts the complexity of the COGNIT Framework for the end user. The chapter is organized to first describe the general system architecture in Section 2.1, including its internal components and the state machine that manages communications. Subsequently, the report reviews how each of the defined software requirements (from SR1.1 to SR1.6) is individually implemented, offering technical details on the interfaces, programming language support, and security measures adopted.

2.1. System Architecture

The COGNIT library is composed of a set of components, each designed to fulfil a specific role enabling efficient function offloading across the COGNIT cloud–edge continuum and satisfy the defined software requirements. Figure 2.1 shows the main components that make up the COGNIT library. These components can be summarized as follows:

- **Device Runtime**. The class used by the Device Client to offload functions.
- **State Machine**. A state machine in which every state will represent a particular situation in the communication with the COGNIT Framework.
- **State Machine Handler**. Handles the state transitions of the State Machine.
- **Latency Calculator**. Measures the latency of the available Edge Cluster Frontends.
- **`call_queue`**. A queue where the functions wait to be offloaded.
- **`sync_result_queue`**. A queue that blocks the thread until the result is obtained from the COGNIT Framework.

[2] See D3.10 for more details on the library repository and versioning.

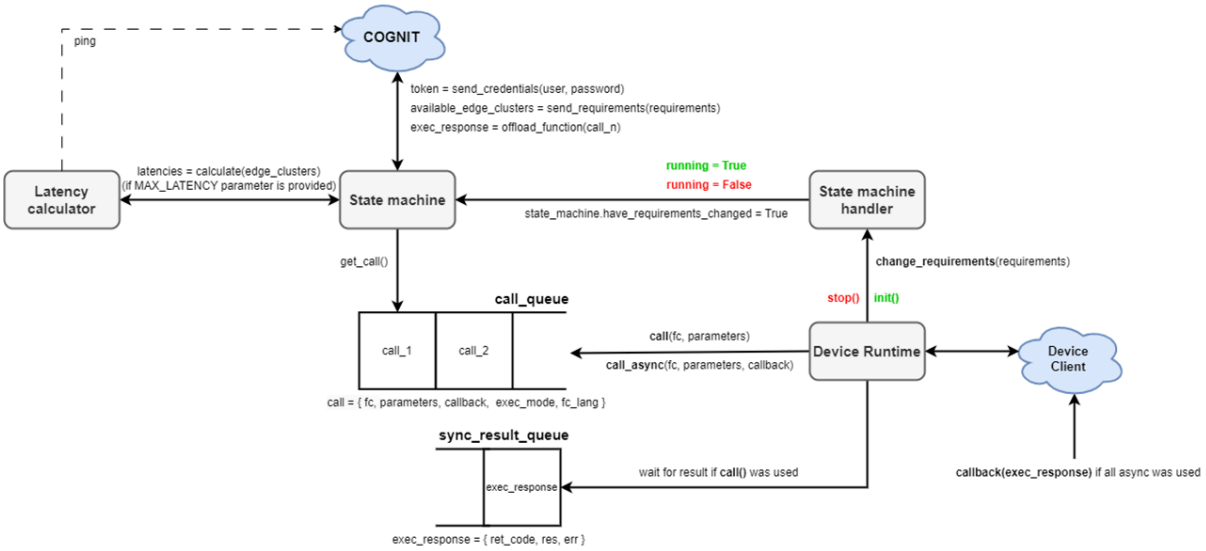

Figure 2.1. COGNIT library architecture

2.1.1. Device Runtime

The Device Runtime class provides a set of methods that allows the Device Client to offload functions with a set of user-defined requirements. In other words, the Device Runtime is the class provided by the COGNIT library that acts as an interface to the Device Client so it can communicate seamlessly with the COGNIT edge-cloud continuum. This interface abstracts the complexity to the Device Client as the communication details will be handled in the background by the State Machine. The Device Runtime available methods are summarized in Table 2.1.

Description	Method	Parameters	Return Type
Initializes the state machine.	`init`	`reqs: Scheduling`	`bool`
Stop the state machine. Deletes de app requirements id generated in the COGNIT Framework.	`stop`	`None`	`bool`
Updates the current requirements that are used	`update_requirements`	`reqs: Scheduling`	`bool`

during the communication with the COGNIT Framework.			

Offloads a function blocking the calling thread until the results are given by the COGNIT Framework.	`call`	`function: Callable`
		
		`*params: Tuple`
		
		`timeout: int (optional)`
			`ExecResponse` object containing the result of the execution or the error in case it was not executed correctly.

Table 2.1. API definition of the Device Runtime class

More technical information about the Device Runtime class can be found here.

The State Machine runs in an independent thread and uses a queue for communication. The Device Runtime enqueues functions to be offloaded into a queue of `Call` objects, called `call_queue`, while the State Machine retrieves these objects to generate the corresponding HTTPS request for offloading.

Each `Call` object stores the necessary information to offload a function, specifically:

- The serialized function to be sent and parameters, which must match the number and type of the function's arguments.

- A callback for asynchronous executions (from the user's perspective). This allows the thread that issued the offloading request to remain unblocked. When using the `call_async` method of the Device Runtime, the user can enqueue a function without blocking. Once the result becomes available, the specified callback is invoked to deliver it.

  Conversely, if the call method is used, no callback is required, since the result is returned directly once it is placed in the `sync_result_queue`. In this case, the call method blocks the thread until the result is available.

- The execution mode, which specifies whether the function should run in parallel with others or not. Although the architecture is designed to support it in the future, currently, only sequential execution is supported. This means that the next call will not be offloaded until the previous one has finished.

- The function language, since both C and Python device clients are supported. The user must indicate whether the function to be offloaded is written in C or Python.

To instantiate a Device Runtime object, a configuration file must be provided. This file should contain two key-value pairs: `credentials` and `api_endpoint`.

- The `credentials` parameter should include the user and password, separated by a colon (:). These credentials are used for authentication with the COGNIT Frontend.

- The `api_endpoint` parameter specifies the URL where the COGNIT Frontend is accepting requests.

This configuration allows the Device Runtime to securely authenticate and communicate with the COGNIT Frontend. This file must follow the structure shown in the following example:

```
api_endpoint: "https://cognit-lab-frontend.sovereignedge.eu"
credentials: "****:****"
```

Figure 2.2. Device Runtime configuration file example

The path to this file has to be provided when instantiating a Device Runtime object. Thereafter, `init` can be called with the user-defined requirements. These requirements can be created by the Device Client creating a dictionary. However, this dictionary must comply with the `Scheduling` model. The `Scheduling` model defines the available requirements that the user can define. The most important ones are:

- `FLAVOUR`: It is a string that describes the flavour of the runtime. It is mandatory to set this requirement as it is used to define in which type of serverless runtime the functions will be executed in terms of dependencies and computing resources.

- `ID`: Unique identifier assigned to each device. This field enables device-specific assignment caching and is used as part of the composite key for database lookups. This field is mandatory and must be of type string.

- `PROVIDERS`: It is an optional field that provides a list of cloud provider identifiers used to restrict cluster selection.

- `IS_CONFIDENTIAL`: It is an optional requirement and indicates if Confidential Computing is required for the execution of the functions. By default this requirement is set to false.

- `GEOLOCATION`: It is also a mandatory requirement as it is used to select the nearest Edge Cluster Frontend for the user application. This parameter is composed of a dictionary with two keys ("latitude" and "longitude") that have to be filled with

float values. This requirement communicates the current geolocation of the Device Client.

- `MAX_FUNCTION_EXECUTION_TIME`: Maximum allowed execution time for functions in seconds. This field is optional and has to be of type float.

- `MAX_LATENCY`: It is an integer optional parameter that defines the maximum latency that the user application will tolerate when offloading a function. Currently, this requirement measures the latency of the Edge Cluster that connects to. It also has the capacity to select the Edge Cluster Frontend with the least latency measured if more than one Edge Cluster Frontends are received.

Once initialized, the State Machine begins retrieving `Call` objects from the `call_queue` which represent an object with all the parameters needed for offloading a function. `call` and `call_async` are the methods given to the Device Client to offload functions as they will convert the parameters into `Call` objects and enqueue them to the `call_queue`. `call` method will expect from the user the function to be executed along with the parameters to execute it. Similarly, `call_async` will expect the same parameters as the `call` method but it also expects a callback where the result will be given. Whereas the call method blocks the thread until the result is returned to the `sync_result_queue`, `call_async` will not block the thread.

The State Machine can be stopped by invoking the `stop` method. Once stopped, to restart the Device Runtime object, the `init` method will have to be called again. When stopping the Device Runtime, a DELETE HTTP request is sent to a specific URL to remove the application requirements generated during the device runtime. This allows the COGNIT Framework to release the resources associated with this application, thereby optimizing resource usage.

Additionally, the requirements can be updated at any time after initialization by calling `update_requirements`. Depending on the requirements provided, a different Edge Cluster will be selected to offload the functions.

2.1.2. State Machine

The State Machine manages the lifecycle of the communication with the COGNIT Framework. The use of a State Machine to handle communication abstracts this complexity from the user, ensuring seamless integration of the library and making it easy to use. Each of the states represents a particular situation in the communication process between the Device Client and the COGNIT Framework. The different states work as follows:

- `INIT`. In this state, the user has not yet been authenticated on the COGNIT Frontend. Upon successful authentication, the COGNIT Frontend issues a Biscuit token, which is returned to the Device Runtime. For all subsequent requests, this Biscuit token is included in the Authorization header as a Bearer token. This

mechanism ensures that the Device Runtime does not need to repeat the authentication process for each request, relying instead on the validity of the issued token.

- `SEND_INIT_REQUEST`. Once the user is authorised, the state machine will order the upload of the requirements to COGNIT Frontend. This state handles that process.

- `GET_ECF_ADDRESS`. After successfully uploading the requirements, the system transitions to this state, where it requests the address of the Edge Cluster Frontend that will handle the user's offloading requests. The Edge Cluster received will be the one that best fits the requirements sent by the user. If the requirement `MAX_LATENCY` is activated, the latency from the Edge Cluster Frontend will be measured.

  If there were more than one edge cluster that met the previously defined requirements, this requirement would also determine which Edge Cluster Frontend to connect to based on latency.

  As a potential future improvement, latency could be incorporated more explicitly into the decision-making process. It is important to note that latency can only be measured from the client's perspective and not from the server's, since the server is unaware of the client's address. For this reason, there may be scenarios in which leveraging client-side latency information would be beneficial.

  In such cases, both the measured latency and the selected edge cluster could be sent to the Cognit Frontend. This additional information would help reinforce the orchestrator's knowledge and enable more informed and accurate decisions in subsequent connection requests.

- 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 selected Edge Cluster Frontend. The State Machine will be offloading functions, one by one, that are enqueued in the `call_queue`.

  Additionally, a periodic task is initiated in this state to consult the COGNIT Frontend every 600 seconds. This process verifies if a more suitable Edge Cluster is available according to the current application requirements and updated system conditions. If the requirements are modified by the Device Client before the timer expires, the periodic task is automatically reset to align with the new configuration.

  The execution of the periodic task is done by the CallbackTimer class whose API reference can be found here. A summary of this class can be found in the following table:

Description	Method	Parameters	Return Type
Starts the timer A new thread is launched that periodically executes the callback function.	`start`	None	None
Stops the timer. The background thread is terminated gracefully.	`stop`	None	None
Executes the callback function at regular intervals. This method runs inside the background thread and should not be called directly.	`_execute`	None	None

Table 2.2. API definition of the Callback Timer class

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 Handler 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 COGNIT Frontend and Edge Cluster Frontend.
- The obtention of a correct address to the Edge Cluster Frontend attendant.

As it was said previously, when calling the `call` and `call_async` functions, the Device Runtime library does not interact with the 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 State Machine, once in the `READY` state (which requires prior initialization and requirement upload), retrieves these `Call` objects and uses them to offload the functions to the COGNIT Framework.

Three different data structures are used in the State Machine whether is to communicate with the COGNIT Framework or the State Machine Handler:

- `ExecResponse`: Used to communicate the function execution result from the COGNIT Framework to the State Machine. It is the data structure retrieved when calling the `call` function and `call_async` through the argument of the callback function.

- `Scheduling`: This model is used to share the requirements from the Device Runtime to the State Machine through the State Machine Handler when the `update_requirements` or `init` functions are called. Thanks to this model, the State Machine is able to upload the requirements to the COGNIT Frontend with the required format.

- `Call`: This model is created every time `call` or `call_async` is called. The function stored as a `Call` object is then retrieved by the State Machine from the `call_queue` to offload functions to the COGNIT Framework.

The content of these models can be summarized in the following table:

Attribute	Description	Field	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's BaseModel.

`Scheduling`	Object containing the information of the application requirements.	`FLAVOUR: str (required)`
		
		`ID: str (required)`
		
		`PROVIDERS: List[str] (optional)`
		
		`GEOLOCATION: Geolocation (required)`
		
		`MAX_LATENCY: int (optional)`
		
		`MAX_FUNCTION_EXECUTION_TIME: float (optional)`
			Inherited from pydantic's BaseModel.

		`MIN_ENERGY_RENEWABLE_USAGE: int (optional)`	

`Geolocation`	Object containing the geolocation of the device client	`Latency: float`
		
		`Longitude: float`
			Inherited from pydantic's BaseModel.

`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's BaseModel.

Table 2.3. Data Model defining the Device Runtime interaction with the State Machine

More technical information about the Device Runtime models can be found here.

2.1.3. State Machine Handler

So the State Machine can be executed in a separate thread, changes in its behaviour are triggered by the State Machine Handler. This component handles the state in which the State Machine is by setting the conditions depending on the inputs of the Device Client to the Device Runtime (by calling `init`, `stop` or `update_requirements`) or the communication status between the State Machine and the COGNIT Framework.

Therefore, any transition or input to the State Machine will be handled first by the State Machine Handler. This also includes overseeing shutting down or awaking the State Machine. This State Machine management is done through methods that can be summarized in the following table:

Description	Method	Parameters	Return Type
Method that updates the current requirements and triggers the State Machine to switch to the state in which it has	`change_requirements`	`func: Callable` `args: Any` [Bundled as positional arguments]	`bool`

to upload the new requirements.			
Method that launches an infinite loop waiting for changes in the state based on device client inputs or communication status.	`run`	`Requirements: Scheduling`	`None`
Method that stops the State Machine.	`stop`	`None`	`None`
Method that evaluates the state of the different inputs that must be considered to switch to a state different than the current one.	`evaluate_conditions`	`None`	`None`

Table 2.4. API definition of the State Machine Handler class

More technical information about the State Machine Handler methods can be found here.

2.1.4. Latency Calculator

The Latency Calculator is a lightweight component designed to measure the network latency for a given address. Within the COGNIT Framework, this component is activated whenever the MAX_LATENCY requirement is included in the application's scheduling model. In the current architecture, the AI orchestrator assumes sole and exclusive responsibility for determining the optimal Edge Cluster for each request. Therefore, the Latency Calculator is primarily used to measure and verify the real-time latency of the optimal Edge Cluster Frontend address provided by the COGNIT Frontend.

Beyond its current implementation, the Latency Calculator is designed to support more advanced orchestration scenarios. For instance, the measurements collected could be sent back to the COGNIT Frontend to provide the AI orchestrator with real-time, client-side network conditions, enabling even more accurate placement decisions. Additionally, in cases where the framework might provide multiple candidate clusters, this component could be used to perform a final client-side selection by identifying the endpoint with the lowest measured latency, ensuring the best possible performance for the user application.

To measure latency, the Latency Calculator relies on the execution of ping commands, which allow it to estimate round-trip time (RTT) between the Device Client and a given address. This approach provides a lightweight and widely supported mechanism for continuously assessing network performance and comparing latency across different endpoints. The following table summarizes the methods that Latency Calculator class has:

Description	Method	Parameters	Return Type
Calculates the latency to a given IP.	`calculate`	`ip: str`	`float`
Gets the latency of each of the provided edge clusters.	`get_latency_for_clusters`	`edge_clusters : list[str]`	Dictionary containing the latency of each edge cluster.

Table 2.7. API definition of the Latency Calculator component

More technical information about the Latency Calculator class can be found here.

2.2. [SR1.1] Interface with the COGNIT Frontend

2.2.1. Description

So far it has been discussed a communication process that occurs within the State Machine without going into any detail. This communication is what ultimately allows the functions to offload. This communication is achieved through two components: COGNIT Frontend and Edge Cluster Frontend. The COGNIT library has two modules called COGNIT Frontend Client and Edge Cluster Frontend Client which handle the communication with both elements.

The COGNIT Frontend Client is an integral component of the COGNIT library, 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 (not execution) of functions. Lastly, it facilitates the retrieval of the most optimal COGNIT Edge Cluster Frontend endpoint for task offloading.

The COGNIT Frontend Client employs an internal flag `has_connection` to indicate the current connection status, allowing the State Machine to check this flag. Additionally, it maintains an internal variable that maps uploaded functions to the IDs assigned by

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

The steps involved in the communication between the COGNIT Frontend and the State Machine (using the COGNIT Frontend Client) are:

1. The State Machine will get the credential from the path to the configuration file passed in the initialization of the Device Runtime. COGNIT Frontend Client will authenticate against the COGNIT Frontend using these credentials. The State Machine will receive a token which is then attached to all subsequent requests to keep the session active. This process will not start until the `init` method from Device Runtime class is not invoked.

2. After a successful authentication, State Machine will switch the state where it will invoke the COGNIT Frontend Client to upload the requirements provided from the `init` method. If these requirements are uploaded successfully, an "app requirements id" is given in return. At least, the flavour and geolocation parameters have to be sent in these requirements.

3. The "app requirements id" is used by the COGNIT Frontend Client to ask the COGNIT Frontend for the optimal Edge Cluster Frontend. The latency of the Edge Cluster Frontend received can be measured if the `MAX_LATENCY` is activated.

4. Every time requirements are updated by the device client, steps two to four are repeated. These steps also represent the `SEND_INIT_REQUEST` and `GET_ECF_ADDRESS` states. Note that, if no requirements are updated in a 600-second-period, the State Machine will request again for an Edge Cluster Frontend address again. These allow the system to check whether a better Edge Cluster Frontend is available for the previously submitted requirements.

Once the Edge Cluster is selected, the State Machine will switch from `GET_ECF_ADRESS` to `READY` so it can offload functions through the Edge Cluster Frontend Client. Before a function is offloaded, it must be stored in the COGNIT distributed object storage by the State Machine using the COGNIT Frontend Client.

2.2.2. 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 following table summarized the models used in the communication with COGNIT Frontend:

Attribute	Description	Fields	Type

Scheduling	Object containing the information of the application requirements.	FLAVOUR: str (required) GEOLOCATION: Geolocation (required) MAX_LATENCY: int (optional) MAX_FUNCTION_EXECUTION_TIME: float (optional) MIN_ENERGY_RENEWABLE_USAGE: int (optional)	Inherited from pydantic's BaseModel.
Geolocation	Object containing the geolocation of the device client.	Latency: float Longitude: float	Inherited from pydantic's BaseModel.
FunctionLanguage	Enum defining the language of the offloaded function.	PY = "PY" C = "C"	Enum
UploadFunction	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

Table 2.5. Data model of the COGNIT Frontend Client component.

More technical information about the COGNIT Frontend Client models can be found here.

2.2.3. API & Interfaces

The COGNIT Frontend Client is composed of several methods, as depicted in Table 2.6, which are abstracted from the user as they are used in the State Machine.

Description	Method	Parameters	Return Type
Uploads the requirements to the COGNIT Frontend.	`init`	`reqs: Scheduling`	`bool` indicating that the requirements were successfully uploaded.
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.
Gets 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.
Updates 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 against COGNIT Frontend to get a valid Biscuit Token.	`_authenticate`	`None`	`str`
Gets the available edge cluster addresses	`get_edge_cluster_frontends_available`	`None`	A dictionary containing all the available edge clusters addresses.

Serializes and uploads the function. Adds the uploaded function id to the class variable map.	`upload_function_to_daas`	`func: Callable`	`int`
Performs 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.
Setter for `has_connection` flag.	`set_has_connection`	`new_value: bool`	`None`
Gets the edge cluster address that complies best with the requirements.	`_get_edge_cluster_address`	`None`	`str`

Table 2.6. API definition of the COGNIT Frontend Client component.

More technical information about the COGNIT Frontend Client class can be found here.

2.3. [SR1.2] Interface with the Edge Cluster Frontend

2.3.1. Description

The Edge Cluster Frontend Client is the component from the COGNIT library that manages the entire communication process with the Edge Cluster Frontend. 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. The Serverless Runtime is responsible for executing functions in the COGNIT Framework, while the Edge Cluster Frontend distributes the workload across multiple Serverless Runtimes using RabbitMQ.

There is a single endpoint available to interact with the Edge Cluster Frontend. This endpoint supports synchronous function execution, meaning that a request triggers the execution of the function and the response immediately returns its result. Internally, the function is first uploaded to the COGNIT distributed object storage, after which its returned ID, along with the required parameters, is sent to the Edge Cluster Frontend.

2.3.2. 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

Triggers the execution of a function described by its ID in a certain mode using certain parameters for its execution.	`execute_function`	`func_id: int`
		
		`app_req_id: int`
		
		`exec_mode: ExecutionMode`
		
		`callback: Callable`
		
		`params_tuple: tuple`
			`ExecResponse`

		`timeout: int (optional)`	
Serializes the parameters to be sent in the request	`get_serialized_params`	`params_tuple: tuple`	`list`
Setter for `has_connection` flag.	`set_has_connection`	`new_value: bool`	`None`
Getter for `has_connection` flag.	`get_has_connection`	`None`	`bool`

Table 2.8. API definition of the Edge Cluster Frontend Client component.

More technical information about Edge Cluster Frontend Client class can be found here.

2.3.3. Data Model

Table 2.9 describes the data model followed by the Edge Cluster Client in order to achieve a successful communication with the Edge Cluster Frontend. In other words, the following attributes define all the formats the client expects from the Edge Cluster Frontend:

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 2.9. Data Model followed by the Edge Cluster Frontend Client

More technical information about the Edge Cluster Frontend models can be found here.

2.4. [SR1.3] Programming languages

2.4.1. Description

In addition to the Python version of the COGNIT library, there is also a C version of the library to support use cases where the use of Python language is not feasible. The global functionality of this library is very similar to the Python version. Thus, in this section we will focus on describing the structures and functions of the library with which the Device Client will interact, and then we will detail the specifics of this particular implementation.

The main difference lies in the serialization of the data to be sent to the Edge Cluster Frontend. As the C COGNIT library must be able to offload Python functions, a language agnostic serialization method is required, enabling the correct understanding between the C Device Client and the Serverless Runtime. The solution implemented for this requirement has been Protobuf[3]. For more detailed technical information, please refer to the GitHub wiki at the following link.

2.4.2. Architecture & components

The main components of the C version COGNIT library are the following:

- **Device Runtime**. The module used by the Device Client to interact with the library.
- **State Machine**. The C COGNIT library also implements a state machine to manage communications with the platform. This state machine follows the same logic as the Python library, implementing the same states and transitions described in Section 2.1.2.
- **COGNIT Frontend Client.** This module is responsible for the interaction with the Cognit Frontend. This module follows the same API and data model described in Section 2.2.

[3] https://protobuf.dev/

- **Edge Cluster Frontend Client.** This module is responsible for the interaction with the Edge Cluster Frontend. This module follows the same API and data model described in Section 2.3.

- **Protobuf Parser.** This module is responsible for offering the capabilities of serializing and deserialing Protobuf messages to enable the offloading of functions and parameters to the Cognit platform.

The C library offers the following classes and methods to manage the interaction with the COGNIT Platform.

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 modul.e
`faas_t`	Stores the function and parameters to be offloaded.

Table 2.10. C Device Runtime structures

Description	Method	Parameters	Return Type

Enables the developer to configure the endpoint, credentials and requirements to	`device_runtime_init`	`device_runtime_t`
		`cognit_config_t`
		`scheduling_t`
			`e_status_code_t`

connect to the COGNIT Platform instance.		`faas_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 with the parameters.	`device_runtime_call`	`device_runtime_t`
		`scheduling_t`
		`faas_t`
		`void**`
			`e_status_code_t`

Table 2.11. C Device Runtime methods

2.4.3. Protobuf Parser

Protocol Buffers (Protobuf) is a data serialization framework developed by Google that offers a mechanism to serialize structured data. It is language-neutral and platform-neutral, making it useful for communications between heterogeneous services. Figure 2.3 summarizes how Protobuf works.

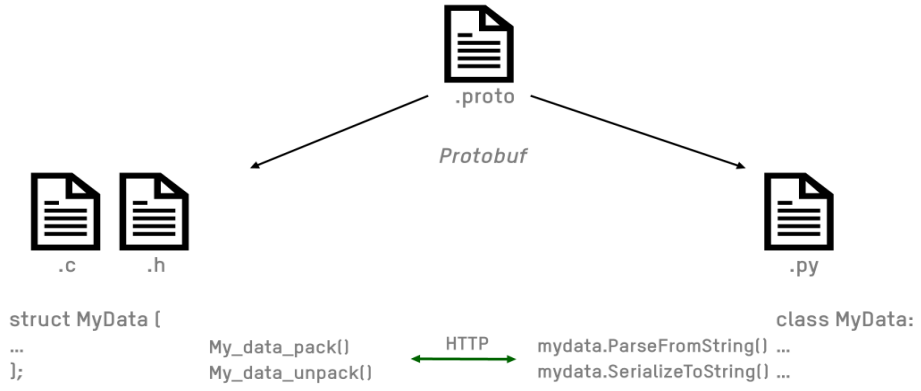

Figure 2.3. Protobuf functioning

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.

To make the COGNIT library self-contained and versatile for different scenarios, it was necessary to establish a set of pre-defined messages and integrate the code to serialize/deserialize these messages in the library. In the context of this project, the Device Client shall send two types of serialized messages: the function to offload and the actual request of executing the function with the set of parameters of the specific execution. In addition, the response of the execution of the function will be received from the platform and shall be deserialized. Thus, the following messages were defined in the *.proto* file:

- **MyFunc.** A message which includes a string type element, where the code of the function is introduced to be serialized.

- **MyParam.** This message is meant to gather a parameter to perform the execution of the function. The MyParam message permits introducing an array of any type supported by the Protobuf library.

- **FaasResponse.** This message defines an array (8 elements by default) of `MyParam` elements, which are the elements returned by the function. Despite the request, which must be sent as an array of serialized parameters, the response needs to be a single serialized element. This forces to define this message, where the parameters will be introduced and the will be serialized as a whole.

Once these messages were defined, the compilation of the *.proto* file generated the code implementing the C code for the COGNIT library and the Python code for the Serverless Runtime so that they can interact by using these messages. In the COGNIT Library, these messages are managed by the Protobuf Parser, which offers the `addFC()`, `addXParam()` and `addXArray()` functions to introduce the user functions and parameters into these messages and serializes/deserializes them.

In the context of this project we decided to use a specific implementation of Protobuf, which is nanopb[4]. It is a C implementation of Protocol Buffers specifically designed for

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

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.

2.5. C Device client usage example

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 milliseconds.
    .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 milliseconds.
    .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;
}
```

2.6. [SR1.4] Low memory footprint

2.6.1. 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: 35 KB

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

- Code size: 463 KB
- Memory usage: 135 KB

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

2.7. [SR1.5] Security

2.7.1. Description

The device client is designed with strong security guarantees, incorporating both authentication and confidentiality mechanisms.

To initialise the client, valid credentials (username and password) must be supplied through a secure configuration file. These credentials are required to establish a trusted session with the COGNIT Frontend. Without successful authentication, the client will be denied access to the services provided by the COGNIT Framework.

To avoid repeated credential exchanges on every request, the system implements a token-based authentication mechanism. After the initial login, the client receives a Biscuit token issued by the COGNIT Frontend. This token is subsequently included in the Authorization header of each request (as `Authorization: Bearer <token>`), allowing the client to access services without re-authenticating with username and password. Tokens are time-limited and can be refreshed securely to maintain session continuity.

All communication between the device client, the COGNIT Frontend, and the Edge Cluster Frontend is performed exclusively over HTTPS (TLS 1.2/1.3). This ensures:

- Confidentiality. End-to-end encryption prevents eavesdropping and data leakage.
- Integrity. Cryptographic protections prevent tampering with messages in transit.

This layered approach credentials for initial authentication, token-based session management, and TLS for secure transport ensures that the device client can operate securely while minimising overhead and exposure.

2.7.2. Architecture & components

The components responsible for ensuring authentication policies are COGNIT Frontend and Edge Clusters and their corresponding clients in the Device Runtime. When Device Runtime is initialised via the `init` method, credentials are sent to COGNIT Frontend via the COGNIT Frontend Client. The token received, if authentication is successful, is used for subsequent requests to COGNIT Frontend and Edge Cluster Frontend, which are responsible for verifying its authenticity.

Data confidentiality and integrity are ensured by the same elements as in authentication. Clients make HTTPS requests to their corresponding servers (COGNIT Frontend and Edge Cluster Frontends).

2.7.3. API & interfaces

As it can be seen in Table 2.12, the method involved in the authentication process is the `_authenticate` method from the COGNIT Frontend Client. This method will upload the user credentials and, if they are valid, a Biscuit token will be received. This token is then

added in the subsequent requests made to the Edge Cluster Frontend and COGNIT Frontend.

Attribute	Description	Fields	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 2.12. COGNIT Frontend single authentication method

2.8. [SR1.6] Collecting Latency Measurements

2.8.1. Description

This requirement states: “Latency measurements against Edge Clusters should be acquired by the Device Client.”

One of the key requirements for the Device Runtime is the ability to measure the latency between the Device Client and the Edge Cluster Frontend. These latency measurements could potentially be used by the COGNIT Framework to optimize resource allocation, improve load balancing, and enhance the user experience by dynamically adjusting its behavior according to current network conditions.

2.8.2. Architecture & components

As explained in Section 2.1, `MAX_LATENCY` is an optional parameter that can be included in the requirements. Along with this tag, a maximum tolerable latency must be specified. However, it is important to note that the COGNIT Framework treats this as a best-effort service: while it will attempt to minimize latency, it cannot guarantee compliance with the specified maximum.

This process relies on latency measurements collected by the Device Runtime through its Latency Calculator component. As it was explained in section 2.1.4, the Latency Calculator is only activated when the `MAX_LATENCY` requirement is included. The `MAX_LATENCY` requirement is used to measure the latency of the optimal Edge Cluster Frontend provided by the COGNIT Framework. If future works determine cases where more than one optimal Edge Cluster Frontends can be provided by the COGNIT Framework, this requirement could potentially take the final decision based on the current network conditions selecting the Edge Cluster Frontend with the lowest latency.

Future tasks could also include this latency measured by the device Client as a new input to the AI orchestrator so that it takes network conditions into account when providing an optimal Edge Cluster Frontend. As of now, the geolocation is used as a good proxy for this latency because the closer an Edge Cluster Frontend is, the more likely it is to have the lowest latency.

As mentioned in previous sections, if the requirements remain unchanged for a 600-second time slot, the optimal Edge Cluster Frontend address will be requested again. This allows the system to account for potential changes in system conditions that may result in a different Edge Cluster Frontend being considered optimal. This will allow the Device Runtime to measure the latency of the optimal Edge Cluster Frontend received again and potentially selecting the one with the lowest latency if more than one were received.

2.9. Python Device client usage example

Python

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

from cognit import device_runtime

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

def delayed_sum(a: int, b: int):
    import time
    time.sleep(50)
    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 = {
    "ID": "device1",
    "FLAVOUR": "Energy",
    "GEOLOCATION": {
        "latitude": 43.05,
        "longitude": -2.53
    }
}

REQS_NEW = {
    "ID": "device1",
    "FLAVOUR": "Nature",
    "MAX_FUNCTION_EXECUTION_TIME": 15.0,
    "MAX_LATENCY": 45,
    "GEOLOCATION": {
        "latitude": 43.05,
        "longitude": -2.53
    }
}

REQS_ML = {
    "ID": "device1",
    "FLAVOUR": "EnergyTorch",
    "MAX_FUNCTION_EXECUTION_TIME": 15.0,
    "MAX_LATENCY": 45,
    "GEOLOCATION": {
            "latitude": 43.05,
            "longitude": -2.53
        }
}

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(sum, 17, 5)

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

    # Synchronize offload but timeout after 10 seconds
    result = my_device_runtime.call(delayed_sum, 50, 25, timeout=10)

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

    # Update the requirements
    are_updated = my_device_runtime.update_requirements(REQS_NEW)

    if (are_updated):

        print("Requirements: "+ str(REQS_NEW) + " UPDATED!")

    else:

        print("Requirements: "+ str(REQS_NEW) + "NOT UPDATED!")

    # Offload asynchronously a function
    my_device_runtime.call_async(sum, 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("-----------------------------------------------")

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

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

```
    # Update the requirements
    are_updated = my_device_runtime.update_requirements(REQS_ML)

    if (are_updated):

        print("Requirements: "+ str(REQS_ML) + " UPDATED!")

    else:

        print("Requirements: "+ str(REQS_ML) + "NOT UPDATED!")

    # 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("-------------------------------------------------------")

    time.sleep(5)

    # Offload and execute a function
    result = my_device_runtime.call(mult, 9, 12)

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

    # Stop device runtime
    my_device_runtime.stop()

except Exception as e:

    print("An exception has occurred: " + str(e))
    exit(-1)
```

3. COGNIT Frontend

3.1. [SR2.1] COGNIT Frontend

3.1.1 Description

The COGNIT Frontend is the primary entry point for all Device Clients requesting access to the COGNIT Framework for offloading computational tasks through the Function-as-a-Service (FaaS) paradigm. As specified in Software Requirement SR2.1, this component acts as a centralized gateway that mediates all interactions between edge devices and the distributed COGNIT infrastructure.

The COGNIT Frontend serves several functions within the COGNIT ecosystem. It manages device authentication and authorization using Biscuit token-based cryptography, handles the lifecycle of application requirements that specify device computational needs, facilitates the upload and management of functions to be executed in the Cloud-Edge continuum, and provides Edge Clusters to the devices where to offload functions.

3.1.1 Architecture & Components

The COGNIT Frontend is implemented as a FastAPI application. The architecture is modular and consists of several components that work together to provide the complete functionality required by the COGNIT Framework.

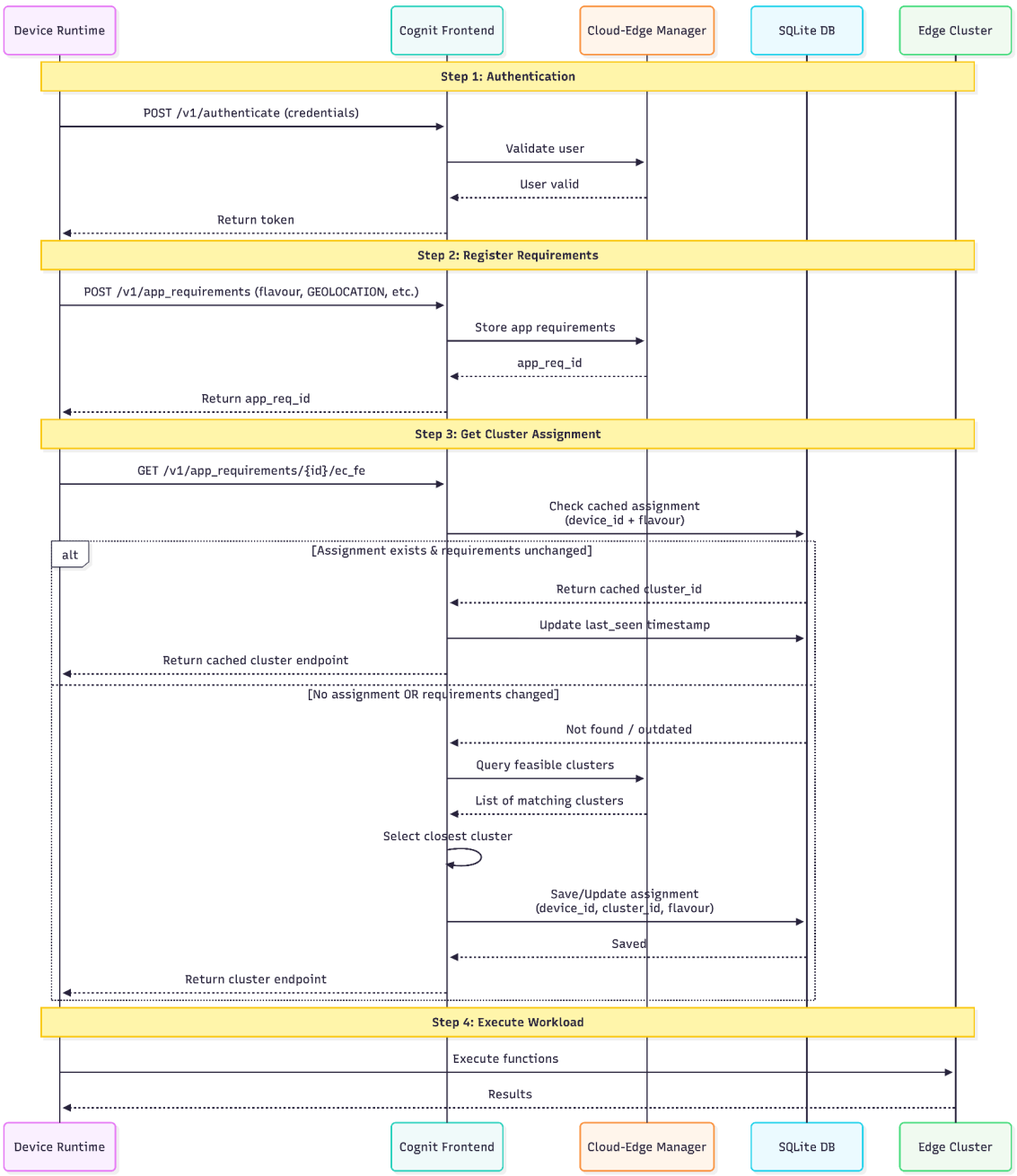

Figure 3.1. COGNIT Frontend interaction flow showing authentication, requirement management, and cluster assignment with caching.

The COGNIT Frontend is composed of the following core modules.

The **REST API** serves as the communication interface between Device Clients and the COGNIT Frontend. Built on FastAPI, it provides high-performance asynchronous request handling with automatic OpenAPI documentation generation. All endpoints are secured through Biscuit token verification, ensuring that only authenticated devices can access COGNIT services.

The **Authentication Manager** handles the complete authentication lifecycle using Biscuit token-based public-key cryptography. When a device authenticates with valid Cloud-Edge Manager credentials, the Authentication Manager generates a Biscuit token containing the device's identity and permissions. The private key used for token generation is regenerated on each service restart for enhanced security, while the corresponding public key is exposed via a dedicated endpoint for token verification by other COGNIT components (such as Edge Cluster Frontends).

The **Application Requirements Handler** manages the complete CRUD lifecycle of application requirements. Application requirements define the constraints and preferences for function execution, including the Serverless Runtime flavour, maximum latency tolerance, geolocation information, confidential computing needs, provider restrictions. The handler interfaces with the Cloud-Edge Manager's document pool to persist these requirements as structured documents that can be referenced throughout the device's session.

The **Function Handler** enables devices to upload function code to the COGNIT Framework. Functions are serialized, base64-encoded, and hashed for integrity verification before being stored in the Cloud-Edge Manager document pool. Each function is assigned a unique identifier that can be referenced during execution requests. This approach allows functions to be uploaded once and executed multiple times across different Edge Clusters without redundant data transfers.

The **Assignment Manager** maintains a local SQLite database that caches device-to-cluster assignments based on a composite key of device ID and flavour. When a device requests an Edge Cluster assignment, the manager first checks if a valid cached assignment exists and whether the application requirements have changed. If the requirements are unchanged, the cached cluster is returned immediately, avoiding expensive cluster selection computations. If no assignment exists or requirements have changed, the manager queries the Cloud-Edge Manager for feasible clusters, selects the optimal one the closest based on geolocation, and caches the new assignment. The database also tracks timestamps for automatic cleanup of stale assignments. The AI-Enabled Orchestrator (defined in WP4) actively uses this database to asynchronously recompute and update device-to-cluster assignments based on evolving infrastructure conditions, workload patterns, and optimization objectives. This asynchronous update model ensures that the COGNIT Frontend can always provide low-latency responses to device requests by reading current assignments from the database, while the AI-Enabled Orchestrator continuously refines these assignments in the background to maintain optimal workload distribution across the cloud-edge continuum.

The **Cloud-Edge Manager Client** provides an abstraction layer over the OpenNebula XML-RPC API. It handles authentication, document management operations, cluster queries, and general API call error handling. The client translates high-level operations into appropriate OpenNebula API calls and processes responses into formats consumable by other COGNIT Frontend components.

The **Global DaaS** provides a global data storage service accessible from all Edge Clusters. Global DaaS is implemented as MinIO, an S3-compatible object storage system deployed with the COGNIT Frontend. This service enables devices to upload data that needs to be accessible regardless of which Edge Cluster executes functions, ensuring data persistence even when devices are reassigned to different Edge Clusters. The COGNIT Framework does not provide stateful function execution or automatic state management; applications are responsible for managing their own data and state. Devices can upload application data to Global DaaS for use by functions executing across the cloud-edge continuum.

3.1.3 Data Model

The COGNIT Frontend data model defines the structure of information exchanged between Device Clients and the framework, as well as the internal representation of state maintained by the service.

The **Application Requirements Object** model captures all requirements specified by the application running on the Device Client. The requirements guide cluster selection and function placement decisions:

Attribute	Description	Type
ID	Unique identifier of the device. This field enables device-specific assignment caching and is used as part of the composite key for database lookups.	Mandatory String
IS_CONFIDENTIAL	Indicates whether the function execution requires Confidential Computing hardware (e.g., AMD SEV-SNP, Intel TDX).	Optional Boolean (Default: False)

PROVIDERS	Restricts cluster selection to specific cloud providers (e.g., ["DC1", "DC2", "AWS-Zone-1"]).	Optional List of Strings
FLAVOUR	Specifies the Serverless Runtime flavour identifier. Each flavour corresponds to a pre-configured runtime image with specific libraries, tools, and capabilities.	String
MAX_LATENCY	Maximum acceptable network latency in milliseconds between the device and the Edge Cluster.	Optional Integer
MAX_FUNCTION_EXECUTION_TIME	Maximum allowed execution time for functions in seconds.	Optional Float
MIN_ENERGY_RENEWABLE_USAGE	Minimum percentage of renewable energy that should power the selected cluster, supporting sustainability objectives.	Optional Integer

GEOLOCATION	Geographic coordinates of the device (latitude and longitude). Used for proximity-based cluster selection when latency constraints are defined.	Optional Object

The **Edge Cluster Frontend Object** model represents the information about an assigned Edge Cluster that is returned to the Device Client:

Attribute	Description	Type
ID	Unique identifier of the cluster in the Cloud-Edge Manager cluster pool.	Integer
NAME	Name of the cluster.	String
HOSTS	List of hypervisor host IDs belonging to the cluster.	List of Integers
DATASTORES	List of datastore IDs available in the cluster.	List of Integers
VNETS	List of virtual network IDs configured in the cluster.	List of Integers
TEMPLATE	Additional cluster metadata including the Edge Cluster Frontend endpoint URL, geographic location, capabilities (e.g., confidential computing support), and provider information.	Dictionary

The **Function Object** model defines the structure of functions uploaded to the COGNIT Framework:

Attribute	Description	Type
LANG	Programming language of the function. Supported values are "PY" (Python) and "C" (C language).	FunctionLanguage Enum
FC	The function code serialized and encoded in base64 format for safe transmission.	String
FC_HASH	SHA-256 hash of the function code, serving as both an integrity check and a unique function identifier.	String

The device-to-cluster assignment cache is maintained in a SQLite database with the following schema:

```SQL
CREATE TABLE device_cluster_assignment (
    device_id TEXT NOT NULL,
    cluster_id INTEGER NOT NULL,
    flavour TEXT NOT NULL,
    last_seen TIMESTAMP NOT NULL,
    app_req_id INTEGER NOT NULL,
    app_req_json TEXT NOT NULL,
    estimated_load REAL DEFAULT 1.0,
    PRIMARY KEY (device_id, flavour)
)
```

The composite primary key (`device_id, flavour`) is essential to avoid conflicts between different Use Cases that may have device IDs in common. This design ensures that each Use Case can maintain its own cluster assignment independently, even when multiple Use Cases share the same device identifier. The `app_req_json` field stores a

JSON representation of the complete application requirements, allowing change detection by comparing incoming requirements with cached ones. The `last_seen` timestamp is updated on each cluster query, enabling automated cleanup of stale assignments. The `estimated_load` field stores the load estimation and forecasting (based on observations and predictions) of the computational load this device is expected to generate. This load estimation is actively used by the AI-Enabled Orchestrator to inform cluster assignment decisions and workload balancing strategies across the COGNIT infrastructure.

3.1.4 API & Interfaces

The COGNIT Frontend exposes a RESTful API that defines the interface through which Device Clients consume the functionality of the COGNIT Framework. All endpoints (except authentication and public key retrieval) require a valid Biscuit token to be provided in the request header. The API is fully documented using OpenAPI 3.0 specification and is accessible at the `/docs` endpoint of the running service.

Action	Verb	Endpoint	Request Body	Response
Authenticates to COGNIT Frontend.	POST	`/v1/authenticate`	HTTP Basic Auth with username and password in the Authorization header.	Status 201 (Created) with the generated Biscuit token as the response body.
Gets token public key.	GET	`/v1/public_key`	None	Status 200 (OK) with the public key string that can be used to verify tokens issued by this COGNIT Frontend instance.
Uploads application requirements.	POST	`/v1/app_requirements`	JSON representation of the AppRequirements model (with Biscuit token in header).	Status 200 (OK) with the created application requirements document ID.

Action	Verb	Endpoint	Request Body	Response
Updates application requirements.	PUT	`/v1/app_requirements/{id}`	JSON representation of the AppRequirements model (with Biscuit token in header).	Status 200 (OK) on successful update.
Gets application requirements.	GET	`/v1/app_requirements/{id}`	None (with Biscuit token in header).	Status 200 (OK) with the AppRequirements JSON document.
Deletes application requirements.	DELETE	`/v1/app_requirements/{id}`	None (with Biscuit token in header).	Status 204 (No Content) on successful deletion.
Get assigned Edge Cluster.	GET	`/v1/app_requirements/{id}/ec_fe`	None (with Biscuit token in header).	Status 200 (OK) with a list containing the EdgeClusterFrontend object for the optimal cluster. Returns 404 (Not Found) if no clusters match the requirements.
Upload function.	POST	`/v1/daas/upload`	JSON representation of the ExecSyncParams model (with Biscuit token in header).	Status 200 (OK) with the created function document ID.

Table 3.1. COGNIT Frontend API Specification

The `/v1/app_requirements/{id}/ec_fe` endpoint implements assignment logic that leverages the database cache for improved performance:

1. **Retrieve Application Requirements:** The endpoint first retrieves the full application requirements document from the Cloud-Edge Manager using the provided ID.
2. **Cache Lookup:** The device identifier is extracted from the requirements, and the database is queried for an existing assignment matching the device ID and flavour combination.
3. **Assignment Validation:** If a cached assignment exists, the endpoint compares the cached application requirements JSON with the current requirements. If they match exactly, the cached cluster ID is reused, the last_seen timestamp is updated, and the cluster information is returned immediately.
4. **Cluster Selection:** If no cached assignment exists or the requirements have changed, the endpoint queries the Cloud-Edge Manager for all clusters that satisfy the requirements (matching flavour, meeting latency constraints based on geolocation, supporting confidential computing if required, belonging to allowed providers). The list of feasible clusters is sorted by proximity to the device's geolocation, and the closest cluster is selected.
5. **Cache Update:** The new assignment is inserted or updated in the database with the current timestamp and the full requirements JSON.
6. **Response:** The selected cluster's information is retrieved and returned to the Device Client in the EdgeClusterFrontend format.

3.1.5 Configuration

The COGNIT Frontend is configured through a YAML configuration file located at `/etc/cognit-frontend.conf`. The configuration parameters control the service behavior, connectivity to external systems, and operational characteristics:

Attribute	Description	Default
`host`	IP address to which the COGNIT Frontend will bind to listen for incoming requests.	`0.0.0.0`
`port`	Port number on which the service will listen.	1338
`one_xmlrpc`	URL of the OpenNebula XML-RPC endpoint for Cloud-Edge Manager communication.	Required

Attribute	Description	Default
`log_level`	Logging verbosity level for the Uvicorn server (options: `debug`, `info`, `warning`, `error`).	`info`
`db_path`	File system path to the SQLite database for assignment caching.	`/var/lib/cognit-frontend/assignments.db`
`db_cleanup_days`	Number of days after which inactive device assignments are automatically purged from the database.	`30`

Table 3.2 COGNIT Frontend Configuration Parameters

4. Edge Cluster

4.1. [SR3.1] Edge Cluster Frontend

4.1.1 Description

The Edge Cluster Frontend serves as the entry point for function execution requests within each Edge Cluster in the COGNIT Framework. As specified in Software Requirement SR3.1, this component acts as a dispatcher that receives function execution requests from Device Clients and coordinates their execution across the pool of Serverless Runtimes deployed in the Edge Cluster.

The Edge Cluster Frontend addresses a fundamental challenge in distributed serverless computing: efficiently routing execution requests to available computational resources while maintaining security, isolation, and performance guarantees. Rather than implementing a load balancer based on static metrics, the Edge Cluster Frontend leverages a queue-based architecture using RabbitMQ as the underlying message broker. This design decouples the request submission from actual execution, enabling asynchronous processing, automatic load distribution, and resilient handling of transient failures.

When the Edge Cluster Frontend receives a function execution request, it validates the authorization token, retrieves the function and application requirements, and publishes an execution request to a flavour-specific queue. Serverless Runtime instances configured for that flavour act as queue consumers, each processing one execution at a time. This

approach provides natural load balancing as idle Runtimes automatically pick up pending requests, while busy Runtimes continue processing their current tasks without being interrupted.

The Edge Cluster Frontend provides scaling capabilities that enable dynamic adjustment of Serverless Runtime cardinality based on workload demands. It exposes a scaling endpoint that can be invoked by the AI-Enabled Orchestrator to increase or decrease the number of active Runtimes. The scale-down logic implements a multi-phase algorithm that prioritizes terminating idle Runtimes while gracefully draining busy Runtimes by unbinding them from the message queue and waiting for in-flight executions to complete before termination.

4.1.2 Architecture & Components

The Edge Cluster Frontend is implemented as a FastAPI application. The architecture consists of two primary subsystems: the REST API for synchronous request handling and the Broker subsystem for asynchronous message-based communication with Serverless Runtimes.

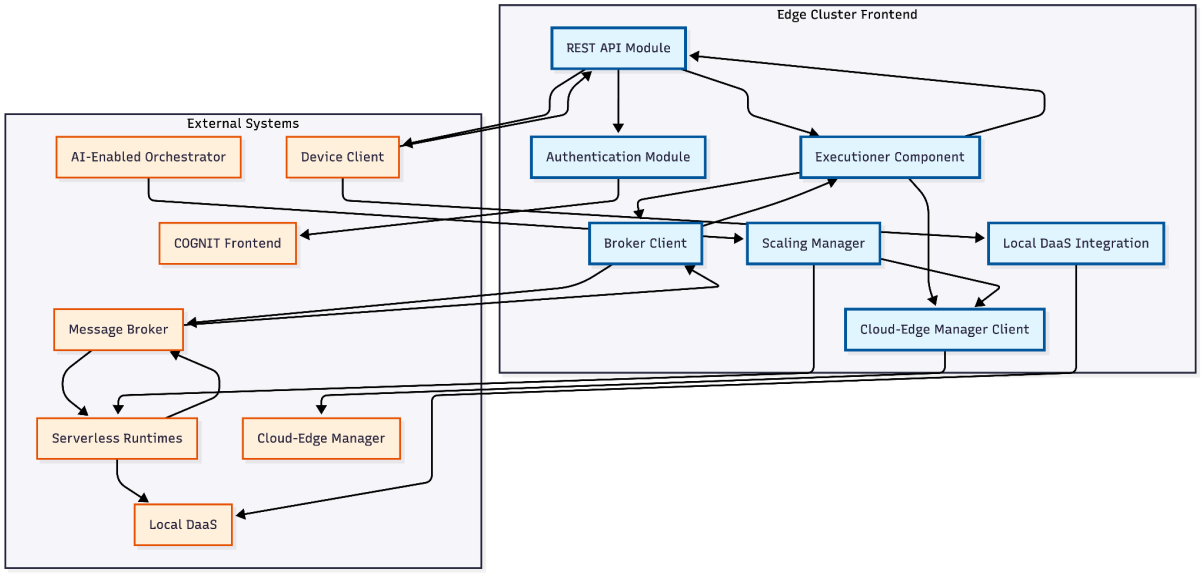

Figure 4.1. Edge Cluster Frontend Architecture.

The Edge Cluster Frontend is composed of the following core modules, as shown in Figure 4.1.

The **REST API Module**, built on FastAPI, exposes the HTTP endpoints that Device Clients use to submit function execution requests. The API handles request validation, token verification, parameter deserialization, and response serialization. For synchronous execution requests, the API blocks until the result is available from the message broker, maintaining the appearance of a traditional request-response cycle while leveraging the benefits of queue-based processing internally.

The **Authentication Module** verifies Biscuit tokens provided in request headers. On startup, the Edge Cluster Frontend retrieves the public key from the COGNIT Frontend and caches it locally. All incoming requests are validated against this public key to ensure they originate from authenticated devices. The token contains embedded user credentials that are extracted and used to authorize access to function and application requirement documents stored in the Cloud-Edge Manager.

The **Broker Client** manages all interactions with the RabbitMQ message broker. It establishes and maintains a persistent connection to the broker, handles connection failures with automatic reconnection, and provides high-level abstractions for message publishing and consumption. The client ensures that required exchanges and queues exist, creating them on demand if necessary. For each execution request, the Broker Client publishes a message to a flavour-specific queue and then subscribes to a results exchange using the execution request ID as the routing key to receive the result when processing completes.

The **Executioner Component** orchestrates the end-to-end execution workflow. When a device submits a function execution request, the Executioner retrieves the function document and application requirements from the Cloud-Edge Manager, generates a unique execution ID, constructs an execution request message containing all necessary information (function code, parameters, requirements, mode), publishes this message to the appropriate queue, and waits for the result. For synchronous executions, the Executioner blocks the REST API response until the result arrives on the results exchange.

This **Scaling Manager** implements the scaling logic for adjusting Serverless Runtime cardinality. When a scaling request arrives from the AI-Enabled Orchestrator, the Scaling Manager interacts with OneGate APIs to retrieve the current service state, determine the target cardinality, and execute the appropriate scaling operation. For scale-up operations, the manager simply instructs OneFlow to increase the cardinality, and new Serverless Runtime are provisioned automatically. For scale-down operations, the manager implements a multi-phase algorithm that classifies Serverless Runtimes as idle or busy, terminates idle Serverless Runtimes immediately, unbinds busy Serverless Runtimes from the message queue to prevent new work assignment, polls until in-flight executions complete, and finally terminates the drained Serverless Runtimes. This graceful shutdown approach ensures no executing functions are interrupted during scale-down.

The **OpenNebula Client** abstracts interactions with the Cloud-Edge Manager. It provides methods for retrieving function and application requirement documents, querying service state via OneGate, and executing OneFlow operations for service scaling. The client handles XML-RPC communication with OpenNebula's API, error handling, and response parsing.

The sequence diagram related to the interaction among the different components is shown in Figure 4.2.

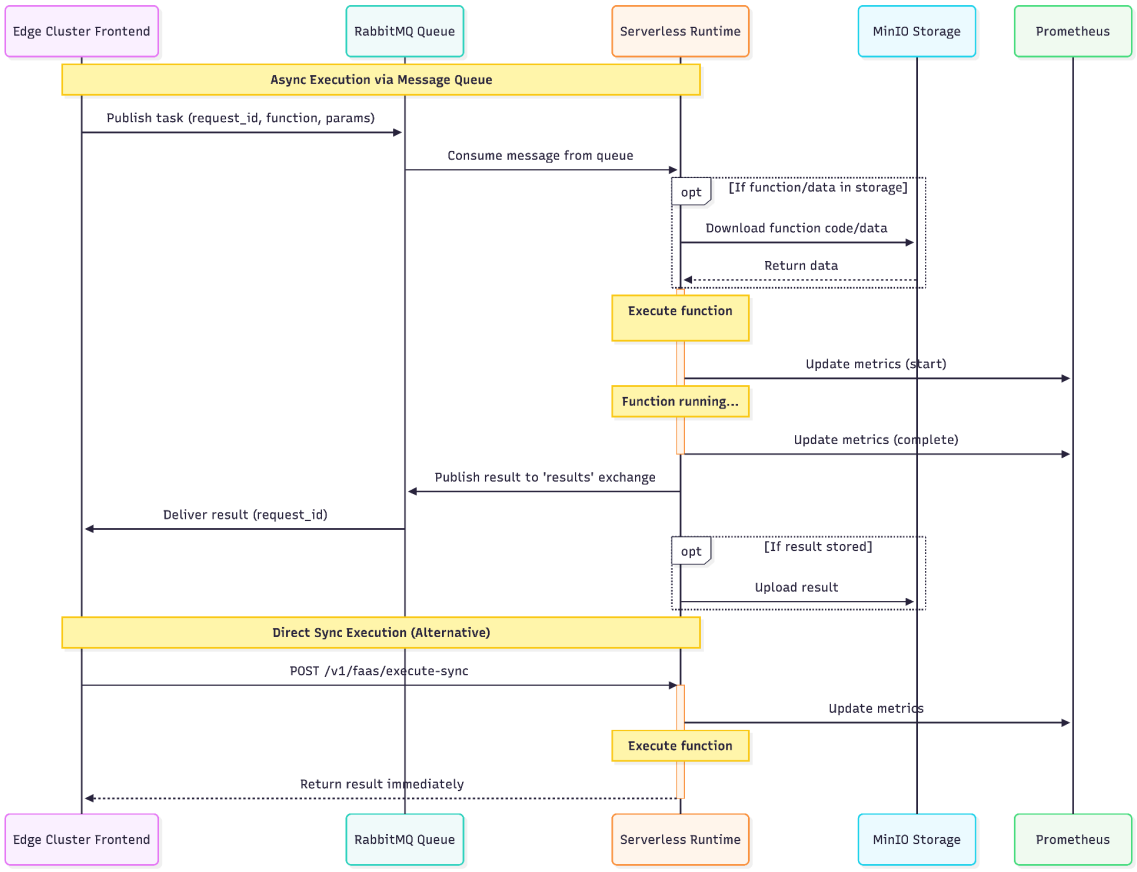

Figure 4.2. Edge Cluster Frontend Execution Flow.

4.1.3 Queue-based Execution Model

The Edge Cluster Frontend's queue-based architecture is crucial for achieving scalability, resilience, and efficient resource utilization.

RabbitMQ is configured with a separate queue for each Serverless Runtime flavour deployed in the Edge Cluster. Each queue operates as a first-in-first-out (FIFO) buffer for execution requests. When a Device Client submits a function execution request, the Edge Cluster Frontend determines the target flavour from the application requirements and publishes the execution message to the corresponding queue. Multiple Serverless Runtime VMs configured for that flavour act as competing consumers on the queue. RabbitMQ's round-robin dispatching ensures that each consumer (Serverless Runtime) receives an equal number of messages, achieving natural load distribution without requiring the Edge Cluster Frontend to track Runtime states or implement complex scheduling algorithms.

Results are returned via a direct exchange named `results`. When a Serverless Runtime completes a function execution, it publishes the result to this exchange using the execution ID as the routing key. The Edge Cluster Frontend, which has been blocking on

the original execution request, is subscribed to a temporary exclusive queue bound to the `results` exchange with a binding key matching the execution ID. When the result arrives, the exclusive queue receives the message, the Edge Cluster Frontend deserializes it, and returns it to the Device Client as the HTTP response body.

This design provides several advantages:

- **Isolation**: Each execution request has its own result queue, preventing cross-contamination or delivery to the wrong client.
- **Automatic Cleanup**: Temporary exclusive queues are automatically deleted when the Edge Cluster Frontend connection closes, preventing resource leaks.
- **Timeout Handling:** If no result arrives within a configurable timeout, the Edge Cluster Frontend can return a timeout error to the Device Client without disrupting the Serverless Runtime.

Each Serverless Runtime runs a RabbitMQ consumer that listens on the flavour-specific queue. The consumer is configured with a prefetch count of 1, meaning it only receives one message at a time from the queue. This prevents a single Runtime from being overwhelmed with multiple execution requests while other Runtimes remain idle. After the Runtime completes an execution, it acknowledges the message, and RabbitMQ delivers the next pending message (if any) to that consumer.

This approach ensures that execution requests are distributed across all available Serverless Runtimes, naturally balancing load based on actual execution time rather than request arrival rate. If a Serverless Runtime crashes or becomes unresponsive, unacknowledged messages are automatically requeued by RabbitMQ and delivered to another consumer, providing built-in fault tolerance.

4.1.4 Scaling Logic

The Edge Cluster Frontend's scaling capabilities enable dynamic adaptation to workload fluctuations, a critical requirement for efficient resource utilization in edge computing environments. The scaling logic is designed to be invoked by the AI-Enabled Orchestrator, which monitors queue depth metrics and performance indicators to make scaling decisions.

Scaling up is straightforward. When the AI-Enabled Orchestrator determines that additional Serverless Runtimes are needed (e.g., due to increasing queue depth or elevated execution latency), it sends a POST request to `/v1/scale?target_cardinality=N` where N is greater than the current number of Runtimes. The Scaling Manager retrieves the current service state from OneGate, validates that the service is in a scalable state, and invokes the OneFlow API to adjust the cardinality of the FaaS role. OpenNebula provisions new VMs using the Serverless Runtime template, which includes pre-configured RabbitMQ consumer scripts that automatically start consuming from the flavour-specific queue upon VM boot. The Scaling Manager polls the service state until the new Runtimes are fully operational and returns a success response.

Scaling down requires more sophisticated logic to avoid interrupting in-flight function executions. The implemented algorithm proceeds in multiple phases:

1. **Identification**: Retrieve the current service state from OneGate and calculate how many VMs need to be removed (`vms_to_remove = current_cardinality - target_cardinality`).
2. **Classification**: For each Serverless Runtime VM in the FaaS role, fetch Prometheus metrics from the VM's Prometheus exporter (port 9100) and parse the `vm_is_executing` gauge. If `vm_is_executing == 0`, the VM is classified as idle (no active execution); if `vm_is_executing == 1`, it is classified as busy (currently executing a function).
3. **Terminate Idle VMs**: Select up to `vms_to_remove` VMs from the idle list and terminate them immediately using the OneFlow API. Decrement `vms_to_remove` by the number of VMs terminated. If `vms_to_remove == 0`, scaling is complete.
4. **Unbind Busy VMs**: If additional VMs still need to be removed, select the required number from the busy list. For each selected busy VM, send a POST request to its control endpoint (`http://{vm_ip}:8000/control/stop-consuming`), which instructs the RabbitMQ consumer to stop accepting new messages from the queue. The VM continues processing its current execution but will not receive additional work.
5. **Poll Until Idle**: Start a timeout timer (default: 10 minutes) and enter a polling loop. Every 5 seconds, fetch Prometheus metrics from each unbound busy VM and check if `vm_current_function` (count of executing functions) has reached zero. When a VM finishes its execution, move it to the `ready_to_terminate` list.
6. **Terminate Drained VMs**: For each VM in `ready_to_terminate`, invoke the OneFlow API to terminate the VM.
7. **Timeout Handling**: If the timeout expires and some VMs are still executing, log a warning and force-terminate them to prevent indefinite blocking of the scaling operation.
8. **Verification**: Poll the OneFlow service until the actual cardinality matches the target cardinality, then return a success response.

This algorithm ensures that function executions are not prematurely terminated during scale-down, providing a better user experience while still allowing the system to reduce resource usage when demand decreases.

The AI-Enabled Orchestrator relies on queue depth metrics to make informed scaling decisions. As illustrated in Figure 4.3, these metrics are collected through an integration between Prometheus exporters running on Serverless Runtime VMs, the RabbitMQ management API, OpenNebula's monitoring system, and QEMU Guest Agent.

OpenNebula's monitoring system executes a guest command via QEMU Guest Agent to fetch Prometheus metrics from each Serverless Runtime VM. The Prometheus exporter,

before returning metrics, queries the RabbitMQ management API to retrieve the current `messages_ready` count for the flavour-specific queue and exposes it as a Prometheus gauge (`rabbitmq_queue_messages_ready`). OpenNebula parses the returned metrics, extracts the queue depth value, and injects it into the VM's template as `vm[:queue_total]`. This metric is then accessible to the AI-Enabled Orchestrator via OpenNebula's XML-RPC API, enabling it to correlate queue depth with scaling needs and trigger scale-up or scale-down operations accordingly.

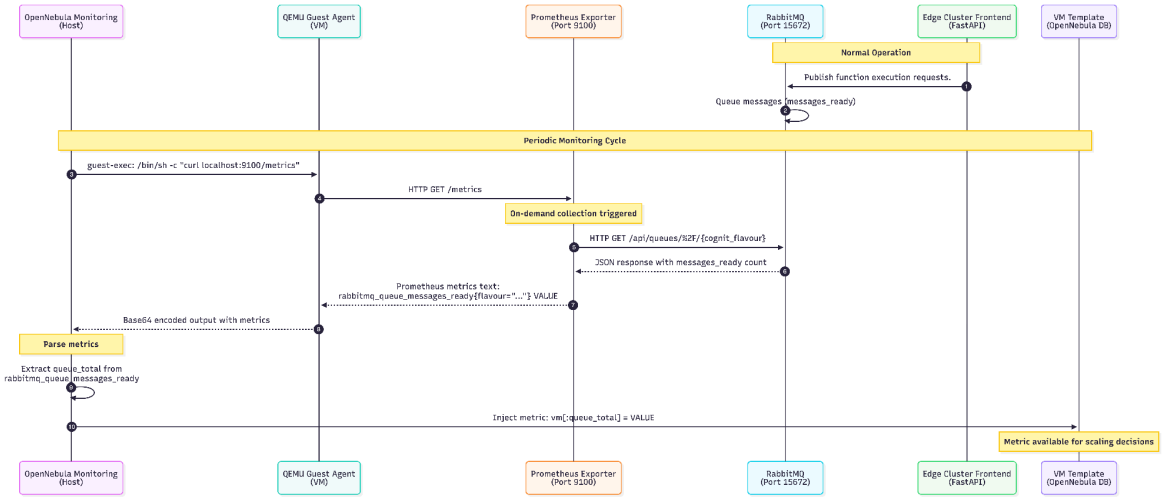

Figure 4.3. Edge Cluster Frontend monitoring integration showing how queue depth metrics are collected via Prometheus and injected into OpenNebula for scaling decisions

4.1.5 Data Model

The Edge Cluster Frontend data model defines the structure of information exchanged during function execution and scaling operations.

Function execution requests are submitted to the /v1/functions/{id}/execute endpoint with the following parameters:

Attribute	Description	Type
id	Document ID of the function previously uploaded to the Cloud-Edge Manager via the COGNIT Frontend. This ID is used to retrieve the function code from the document pool.	Path Parameter, Integer

parameters	Array of serialized and base64-encoded function parameters. Each element represents one parameter that will be passed to the offloaded function.	Request Body, List of Strings
app_req_id	Document ID of the Application Requirements document stored in the Cloud-Edge Manager. This specifies the execution constraints and preferences, including the Serverless Runtime flavour.	Query Parameter, Integer
token	Biscuit token obtained from the COGNIT Frontend used for authentication and authorization.	Header, String

When publishing to RabbitMQ, the Edge Cluster Frontend constructs an execution message containing:

```JSON
{
  "execution_id": "unique-uuid-v4",
  "function_id": 123,
  "app_req_id": 456,
  "function_code": "base64-encoded-function",
  "function_hash": "sha256-hash",
  "language": "PY",
  "parameters": ["param1-base64", "param2-base64"],
}
```

This message is routed to a queue specific to the flavour specified in the application requirements (e.g., `smartcity, energy`).

Serverless Runtimes publish execution results to the `results` exchange using the execution ID as the routing key:

```JSON
{
```

```
  "execution_id": "unique-uuid-v4",
  "ret_code": 0,
  "res": "base64-encoded-result",
  "err": null
}
```

- `ret_code` (Integer): Return code (0 for success, -1 for error).
- `res` (String or null): Base64-encoded execution result if successful.
- `err` (String or null): Error message if execution failed.

The scaling endpoint accepts a single parameter:

- `target_cardinality` (Query Parameter, Integer): Desired number of Serverless Runtime VMs for the FaaS role in the OneFlow service.

All scaling operations return a standardized Json response:

```JSON
{
  "status": "success",
  "data": {
    "service_id": 123
  },
  "message": "Scaling operation completed successfully"
}
```

or in case of failure:

```JSON
{
  "status": "fail",
  "message": "Scaling operation already in progress",
  "code": "SCALING_IN_PROGRESS"
}
```

4.1.3 API & Interfaces

The Edge Cluster Frontend exposes a RESTful API that defines the interface for function execution and service scaling. All function execution endpoints require a valid Biscuit token; scaling endpoints use OpenNebula's internal OneGate authentication.

Action	Verb	Endpoint	Request Parameters	Response
Execute function	POST	`/v1/functions/{id}/execute`	**Path:** id (Function document ID) **Query**: app_req_id (App requirements ID), mode (sync/async) **Header**: token (Biscuit token) **Body**: parameters (list of base64-encoded strings)	Status 200 (OK) with execution result JSON (ret_code, res, err) Status 400 (Bad Request) for malformed input Status 401 (Unauthorized) for invalid token. Status 504 (Gateway Timeout) if execution exceeds timeout.
Upload device metrics	POST	`/v1/device_metrics`	**Header:** token (Biscuit token) Body: JSON with metrics	Status 200 (OK) on successful upload Status 400 (Bad Request) for malformed input. Status 401 (Unauthorized) for invalid token.

Action	Verb	Endpoint	Request Parameters	Response
Scale service	POST	`/v1/scale`	**Query:** target_cardinality (Integer - desired number of FaaS VMs)	JSON success response with service_id JSON fail response with error details (e.g., SCALING_IN_PROGRESS, SERVICE_UNAVAILABLE)
API documentation	GET	`/docs`	None	OpenAPI interactive documentation (SwaggerUI).
Root redirect	GET	/	None	Redirects to /docs

Table 4.2. Edge Cluster Frontend API Specification

Example Function Execution Request:

```shell
POST /v1/functions/123/execute?app_req_id=456&mode=sync
Headers:
  token: <biscuit-token-string>
Body:
{
  "parameters": ["gAVLAi4=", "gAVLAy4="]
}
```

Example Function Execution Response:

```json
{
```

```
  "ret_code": 0,
  "res": "gAVLGC4=",
  "err": null
}
```

Example Scaling Request:

```
Shell
POST /v1/scale?target_cardinality=10
```

Example Scaling Success Response:

```
JSON
{
  "status": "success",
  "data": {
    "service_id": 789
  },
  "message": "Scaling operation completed successfully"
}
```

4.1.4 Configuration

The Edge Cluster Frontend is configured through a YAML configuration file typically located at `/etc/cognit-edge_cluster_frontend.conf`. The configuration parameters control service behavior, connectivity to external systems, and operational characteristics:

Attribute	Description	Default
`host`	IP address to bind for listening to incoming requests.	`0.0.0.0`
`port`	Port number on which the service listens.	1339
`one_xmlrpc`	URL of the OpenNebula XML-RPC endpoint.	Required

Attribute	Description	Default
`oneflow`	URL of the OpenNebula OneFlow API endpoint.	Required
`cognit_frontend`	URL of the COGNIT Frontend for retrieving the public key.	Required
`broker`	URL of the RabbitMQ broker.	Required
`log_level`	Logging verbosity level (`debug`, `info`, `warning`, `error`)	`info`
`workers`	Number of Uvicorn worker processes for handling concurrent requests.	4

Table 4.3 Edge Cluster Frontend Configuration Parameters

4.2. [SR3.2] Secure and Trusted Serverless Runtimes

4.2.1 Description

As mentioned earlier, a Serverless Runtime (SR) is a core component in the COGNIT Framework, being the FaaS execution unit deployed within the Edge Cluster. This component has been built from scratch during the project. The Serverless Runtime is the service deployed into the scheduled node that will be in charge to execute the offloaded tasks, and it exposes the Serverless Runtime API to allow the devices to upload the functions and the needed data to execute them in an isolated environment. For more detailed technical information, please refer to the GitHub wiki at the following link.

4.2.2 Architecture & Components

The Serverless Runtime provides a public Fast API REST Server that listens to FaaS requests, making use of the functionalities given by the private API components and abstracting them from the user for convenience. Multiple components are involved in the execution of the task offloading function:

1. FaaS Models: Provide the data structures needed for the requests and internal communication between function calls.
2. FaaS Parser: It is responsible for serialising the offloaded functions and for deserializing the results returned from the Serverless Runtime.
3. Logger: Provides its own log structure based on different levels of logging.

4. Executor: Groups the logic needed to execute Python and C languages.
5. PyExec: Object that inherits from the base Executor module, extending its functionalities.
6. Metrics exporter: A service that exports metrics to the COGNIT Framework.
7. Minio client: A component enabling CRUD operations with the MinIO server deployed in the COGNIT Framework.
8. RabbitMQ client. A component used to communicate with the RabbitMQ broker located in the Edge Cluster.

Metrics captured by Serverless Runtimes

Each Serverless Runtime (SR) deployed within the Edge Cluster of the COGNIT Framework systematically collects detailed measurements on every function it executes. These metrics are fundamental for providing operational transparency and enabling advanced orchestration decisions across the continuum. The collected information is exposed to the COGNIT Framework.

The first category of metrics concerns the duration of each function execution. For every invocation, the SR records how long the function takes to complete, clearly distinguishing between successful and failed executions. This level of detail enables the system to profile the computational cost of different functions, identify persistent latency issues, and detect anomalies that could signal resource contention or unexpected software behaviour.

A second important aspect captured by the SR relates to the size of the input data provided to each function. By measuring the input size for each invocation and again differentiating between successful and unsuccessful executions, the SR makes it possible to analyse how data volume influences execution behaviour. This is particularly relevant in edge computing scenarios, where resource constraints are common and large or abnormal input sizes may lead to failures or performance degradation.

In addition to these measurements, the SR maintains precise records of the execution lifecycle for each function. This includes real-time status reporting (whether a function is currently running or serverless runtime is in idle state) as well as detailed start and end timestamps for every invocation. Such information is critical for tracing the execution path of individual functions and supporting debugging and performance tuning activities.

Finally, each SR aggregates statistics on the overall workload processed, continuously tracking the number of executed functions, together with the breakdown of those that completed successfully or failed. These high-level figures provide an essential view of reliability and throughput for each Serverless Runtime, enabling the system to monitor for systemic issues, adapt to changing workloads, and feed back essential data for adaptive orchestration strategies.

MinIO client

The MinIO client implementation serves as a core component within the COGNIT Serverless Runtime architecture, providing standardized access to distributed object storage across the framework. This client acts as an abstraction layer that enables Device Runtimes to interact with MinIO storage instances through a unified interface, regardless of their physical location or deployment context within the network infrastructure.

The client's architecture is built upon the AWS S3-compatible API provided by MinIO, utilizing the boto3 library with customized configuration parameters. The implementation incorporates robust error handling, timeout management, and retry mechanisms to ensure reliable operation in potentially unstable network conditions. Through its initialization process, the client establishes authenticated connections to MinIO instances using configurable endpoint URLs and credential pairs, enabling flexible deployment across diverse configurations.

This MinIO client implementation directly enables distributed object storage capabilities within Device Runtimes by providing complete object storage operations including bucket management, object upload and download, metadata handling, and file synchronization. The client supports both in-memory data operations and direct filesystem interactions, allowing serverless functions to process data streams while maintaining the ability to persist intermediate results or cache frequently accessed datasets. These capabilities are essential for enabling data-intensive serverless applications that require reliable and scalable storage backends.

RabbitMQ client

RabbitMQ serves as the primary message broker supporting communication within the distributed serverless runtime architecture. The developed RabbitMQ client provides an interface for managing execution requests and delivering responses across the device runtime infrastructure, ensuring reliable message exchange between system components.

The client is implemented using a multi-threaded architecture that maintains persistent connections to the message broker while processing incoming execution requests. Messages received from designated queues include execution parameters such as operation mode, payload data, and unique request identifiers. These messages are forwarded to local REST API endpoints for processing. Execution requests are offloaded from the main serverless runtime thread to prevent blocking, while internal handling of these requests follows a sequential order within the client’s execution queue. This approach maintains the responsiveness of the runtime environment while preserving an orderly processing of requests.

Finally, response delivery is handled through RabbitMQ’s exchange and routing key mechanisms. Exchanges determine how messages are routed to queues based on predefined rules, while routing keys allow responses to be addressed to the correct recipient. Once a request has been processed, results are published to a dedicated ‘results’ exchange using the original request identifier as the routing key. This ensures that each response is delivered to the Device Runtime that initiated the request, supporting the

distributed and modular design of the runtime while maintaining message isolation and delivery guarantees.

4.2.3 Data Model

The data model of the interaction with the COGNIT Serverless Runtimes defines all the fields expected by it for requests and responses. The following table summarized the models used in the communication with the Serverless Runtime:

Attribute	Description	Fields	Type
`ExecSyncParams`	Object containing information about the offloaded function.	- `lang`: String. Language of the offloaded function. - `fc`: String. Function to be offloaded. - `fc_hash`: String. Hash of the function to be offloaded. - `params`: list[String]. List containing the serialized parameters by each device runtime transferred to the offloaded function. - `app_req_id`: int. Requirement ID that belongs to the current function.	Object inherited from pydantic's BaseModel.
`ExecutionMode`	Object containing the information about the execution mode of the function.	`SYNC = "sync"`	Enum

`FaasUuidStatus`	Object that shows the status of the offloaded function.	`- state`: String. Status of the offloaded function processing task. `- result`: String or Null. Result of the offloaded function.	Object inherited from pydantic's BaseModel.

`ExecReturnCode`	String describing if the execution of the function went successfully or not.	`SUCCESS = 0`
		
		`ERROR = -1`
			Enum

`ExecResponse`	Object containing information about the function execution response.	`ret_code`: `ExecReturnCode`. Reflects the offloaded function execution result. `res`: String or None. Result of the offloaded function. `err`: String or None. Offloaded function execution error description.	Object inherited from pydantic's BaseModel.

Table 4.4. Data model showing the data structures of the Serverless Runtime.

4.2.4 API & Interfaces

Table below shows the public API available for function execution:

Action	Verb	Endpoint	Request body	Response
Request a sync	`POST`	`/v1/faas/execute-sync`	JSON representation of the	Status code 200 (Success) if the execution was

execution of function			language of execution, function object and parameters.	successful. 400 (Bad request) if the Request body is not correctly formatted. 405 (not Allowed) if there is another error with the request.

Table 4.5. Public API defining the way to interact with a given Serverless Runtime.

Additionally, each SR also uses some private endpoints that are not publicly available for the device runtimes, which are depicted in the Table below.

Action	Verb	Endpoint	Request body	Response
Get main route information	`GET`	`/`	None	200 (Success): String describing available routes: "Main routes: POST -> /v1/faas/execute-sync "
Stop RabbitMQ consumer gracefully	`POST`	`/control/stop-consuming`	None	200 (Success): {"status": "success", "message": "RabbitMQ consumer stopped gracefully"} 404 (Not Found): {"status": "error", "message": "RabbitMQ client not initialized"} 500 (Internal Server Error): {"status": "error", "message": "Failed to stop consumer: {error}"}

Prometheus metrics scraping	`GET`	`:9100/metrics`	None	200 (Success): Prometheus-formatted metrics including function execution times, histograms, gauges, and counters

Table 4.6. Private (Operations) API for Serverless Runtime management and control.

5. Secure and Trusted Execution of Computing Environments

5.1. Threat Model

To support the overall risk analysis defined in the architecture deliverables, a comprehensive threat assessment of the COGNIT Framework was conducted through the development of a threat model. The purpose of this activity is to systematically identify, analyse, and document potential security threats affecting COGNIT assets, and to support the definition of appropriate mitigation measures.

Threat Modelling Approach
Threat modelling provides a security-oriented view of the framework, capturing how its components, data flows, and interactions with external entities may be exposed to cybersecurity risks. The threat model represents the system and its environment from a security perspective, enabling a structured understanding of threats and mitigations and improving communication among stakeholders.

The Microsoft Threat Modeling Tool (TMT) was used to produce detailed data flow diagrams and to support systematic threat identification, tracking, and reporting throughout the system lifecycle.

Scope of the Threat Model
The threat model is structured around the following core elements:

- **External Dependencies**: Components outside the direct control of the framework that may introduce security risks, such as external cloud storage services and external authentication providers (e.g. LDAP, OIDC).
- **Access Points**: Interfaces that allow users or systems to interact with the framework, including public APIs of the Device Client, Serverless Runtime, Cloud-Edge Manager, and AI-Enabled Orchestrator.
- **Assets**: Key software components and services that require protection, including execution environments, APIs, schedulers, monitoring services, data handling components, and AI-related modules.
- **Trust Levels**: Clearly defined access privilege levels for users, administrators, and software components, enabling the identification of trust boundaries and privilege separation across the system.
- **Data Flow Diagrams**: Visual representations of data flows between actors, processes, and data stores, highlighting trust boundaries where security controls must be enforced.

Threat Identification and Analysis

After defining the system architecture and trust boundaries, threats were identified using the **STRIDE classification**(Spoofing, Tampering, Repudiation, Information Disclosure,

Denial of Service, Elevation of Privilege). Each system component and data flow was analysed against these threat categories to identify potential weaknesses, whether arising from malicious attacks or from configuration or design errors.

Risk Evaluation and Mitigation

Identified threats are evaluated in terms of likelihood and potential impact. Based on this assessment, decisions are made to:

- eliminate threats through architectural or design changes,
- mitigate risks through appropriate security controls,
- or accept residual risks where justified.

The threat model is treated as a living artefact, updated whenever the system architecture or operational context evolves, ensuring continuous alignment between system design and its security posture.

5.2 The COGNIT Framework Threat Model

The following diagram presents the global view of the COGNIT Framework (please refer to the Deliverable D2.6 for the details about the COGNIT Framework Architecture).

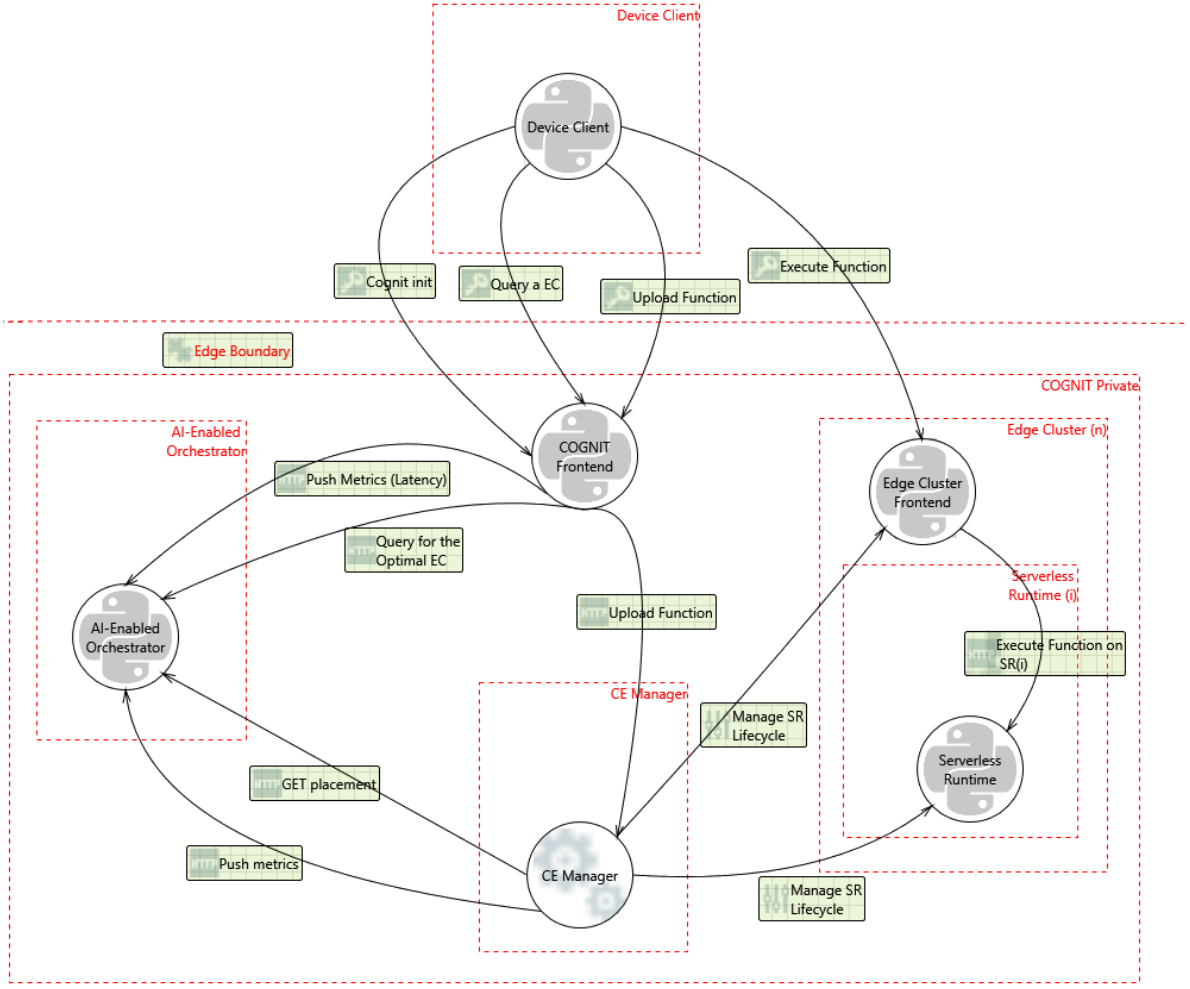

Figure 5.1: COGNIT Framework Threat Model Diagram

The above diagram presents the components and the flows (how the components communicate with each other). At the top, we see the public network and, below the Edge Boundary line, the internal network of the framework.

The arrows describe the data flows, the circles identify components and the dashed lines and boxes describe trust boundaries.

The flows we identified, from the Device Client perspective, are:

1. The COGNIT *init* flow, the client needs to initiate a connection, it uses a secured connection sending credentials as a JSON payload, the COGNIT Frontend sends back a Biscuit token that contains encrypted data[6], if the credentials are valid.

[6]Biscuit JWT token

- the token is generated by the framework and encrypted using a private key
- later on, all communications must contain the token
- the token validation is taking place only in the private network using the public key

2. The *Query a EC* flow, the client queries for an Edge Cluster, the COGNIT Frontend returns information about the selected edge cluster front end endpoint (ID).
3. The *Upload Function* flow, the Device Client sends the function it wants to offload to a Serverless Runtime.
4. The *Execute Function* flow, the device triggers the function execution.

The 4 flows we presented cross a trust boundary from the public network to the framework's internal network.

The internal flows are to be confirmed, therefore we do not provide any explanation about them. We **must** include private zones in the threat modelling because we must consider intrusion cases and their consequences (what attackers could exploit in such cases).

Identified Threats

To identify the threats of the framework, we introduced the architecture description in the threat modelling tool (see screenshot below).

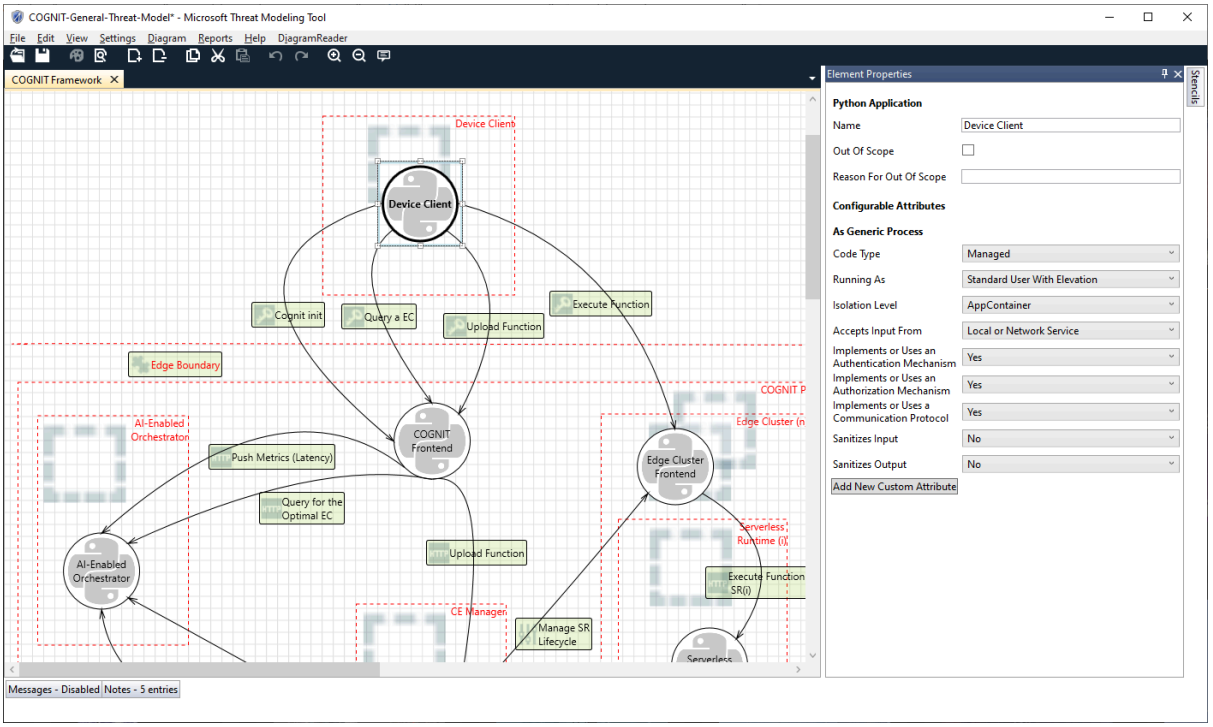

Figure 5.2. Microsoft Threat Modeling Tool

With this description, the tool generated a list of 134 threats. The same type of threats affect different components of the framework. We reduced the list to 13 different threats. This way we will see how to address them all in the same way, in the next steps.

- there is no need to share the public key with the outside world

In the following table we show the list grouped by STRIDE category, where the threats are about flows between two components of the framework named COGNIT *component1* and COGNIT *component 2,* for example COGNIT *component1* could be “Device Client” and COGNIT *component 2* could be “COGNIT Frontend”:

	Threat description
S	COGNIT Component 1 may be spoofed by an attacker and this may lead to information disclosure by COGNIT Component 2. Consider using a standard authentication mechanism to identify the destination process.
	COGNIT Component 1 may be spoofed by an attacker and this may lead to unauthorised access to COGNIT Component 2. Consider using a standard authentication mechanism to identify the source process.
T	Attackers who can send a series of packets or messages may be able to overlap data. For example, packet 1 may be 100 bytes starting at offset 0. Packet 2 may be 100 bytes starting at offset 25. Packet 2 will overwrite 75 bytes of packet 1. Ensure you reassemble data before filtering it, and ensure you explicitly handle these sorts of cases.
	Data flowing across may be tampered with by an attacker. This may lead to a denial of service attack against COGNIT Component or an elevation of privilege attack against COGNIT Component or an information disclosure by COGNIT Component. Failure to verify that input is as expected is a root cause of a very large number of exploitable issues. Consider all paths and the way they handle data. Verify that all input is verified for correctness using an approved list input validation approach.
	If a dataflow contains JSON, JSON processing and hijacking threats may be exploited.
	Packets or messages without sequence numbers or timestamps can be captured and replayed in a wide variety of ways. Implement or utilise an existing communication protocol that supports anti-replay techniques (investigate sequence numbers before timers) and strong integrity.

R	COGNIT Component claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.
I	Custom authentication schemes are susceptible to common weaknesses such as weak credential change management, credential equivalence, easily guessable credentials, null credentials, downgrade authentication or a weak credential change management system. Consider the impact and potential mitigations for your custom authentication scheme.
	Data flowing across may be sniffed by an attacker. Depending on what type of data an attacker can read, it may be used to attack other parts of the system or simply be a disclosure of information leading to compliance violations. Consider encrypting the data flow.
D	An external agent interrupts data flowing across a trust boundary in either direction. Several mitigation actions possible, including: - Integrity checks and message authentication (e.g. HMAC, digital signatures) to detect altered or incomplete data. - Encrypt data in transit using strong, industry-standard protocols (e.g. TLS 1.2/1.3) to prevent interception and tampering.
	COGNIT Component crashes, halts, stops or runs slowly. Apply rate limiting, upstream DDoS protection, resource isolation, and autoscaling to prevent resource exhaustion and maintain acceptable execution performance under attack.
E	COGNIT Component 1 may be able to impersonate the context of COGNIT Component 2 in order to gain additional privilege. Enforce strong service identity, least-privilege authorization, and validated context propagation to prevent component impersonation and privilege escalation.
	An attacker may pass data into COGNIT Component in order to change the flow of program execution within COGNIT Component to the attacker's choosing.

	Prevent attacker-controlled execution flow by enforcing strict input validation, safe execution patterns, and independent authorization checks before privileged operations.

Table 5.6. Grouped Possible Threats

In a threat modelling process we consider the threats of every flow. As an example here is the list of the potential threats of *Query a EC* (picture hereafter).

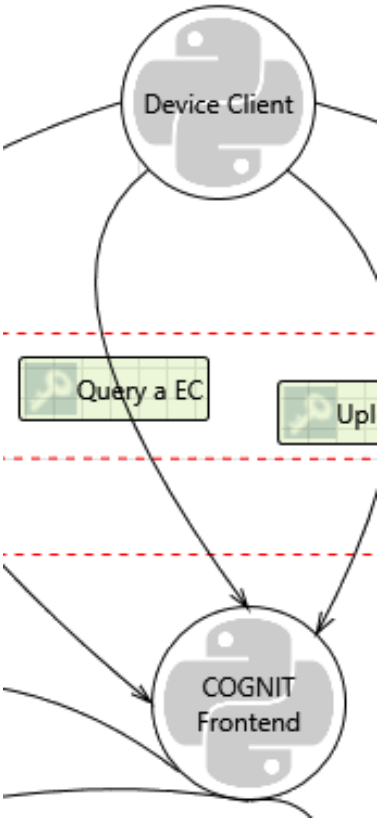

Figure 5.3: The *Query a EC* Flow

The *Query a EC* flow connects the Device Client and the COGNIT Framework components.

Properties for the Device Client component.

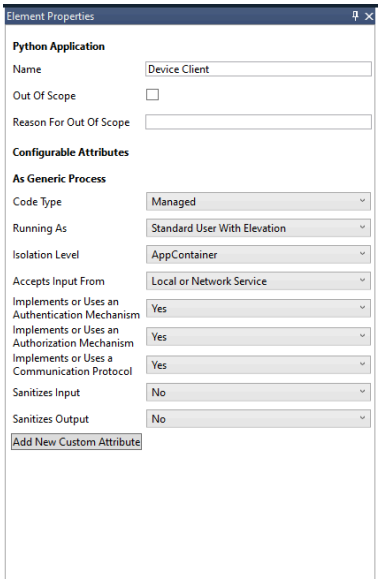

Figure 5.4: Device Client Properties

Properties for the COGNIT Frontend component.

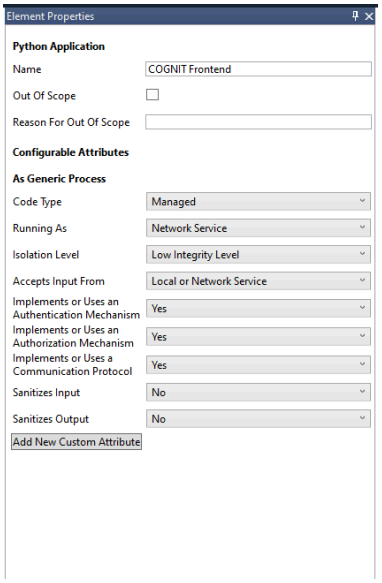

Figure 5.5: COGNIT Frontend Properties

Properties for the *Query a EC* flow.

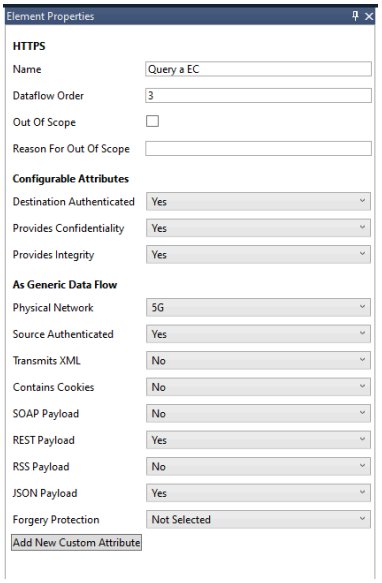

Figure 5.6: Query a EC Flow Properties

We defined the flow as being a HTTPS connection with a JSON payload (the token), then generated the threats. The tool identified 10 threats for the flow:

#	Category	Threat	Status
1	Tampering	Collision Attacks	Need Investigation
2	Tampering	JavaScript Object Notation Processing	Mitigated (CC)
3	Tampering	Replay Attacks	Mitigated
4	Repudiation	Potential Data Repudiation by COGNIT Frontend	Mitigated
5	Information Disclosure	Weak Authentication Scheme	Need Investigation
6	Denial Of Service	Data Flow Query a EC Is Potentially Interrupted	Need Investigation
7	Denial Of Service	Potential Process Crash or Stop for COGNITt Frontend	Need Investigation

#	Category	Threat	Status
8	Elevation Of Privilege	COGNIT Frontend May be Subject to Elevation of Privilege Using Remote Code Execution	Mitigated (CC)
9	Elevation Of Privilege	Elevation by Changing the Execution Flow in COGNIT Frontend	Mitigated (CC)
10	Elevation Of Privilege	Elevation Using Impersonation	Mitigated

Table 5.7. Query a EC Flow Threats

The status in the table is what we can do about the threats. The status values are the following:

- Not Started.
- Need Investigation.
- Not Applicable.
- Mitigated.

We decided to use a new tool, named Microsoft Threat Modeling Tool, to assist us in the modelling work and identified 134 possible threats.

We collaborated with the architecture team to improve and update the model of the framework, leading to more accurate threat assessment.

Based on the information gathered about the new architecture of the COGNIT Framework, we have developed a threat model to identify potential security threats. Certain threats, however, have not yet been addressed. For these, it remains to be determined whether they are confirmed and what measures can be implemented to mitigate or eliminate them.

We reduced the number of different threats to 10, because the same vulnerabilities are present for several components of the framework. These threats have 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.

To enhance the security of the COGNIT framework, we have leveraged Biscuit tokens to mitigate specific threats identified in our assessment. In particular, threats such as replay attacks, potential data repudiation by the COGNIT frontend, and elevation of privilege through impersonation have been addressed using Biscuit. By employing cryptographically verifiable tokens with fine-grained access control, Biscuit ensures that all actions performed through the frontend are authenticated, authorized, and traceable, reducing the risk of denial or misuse. This approach strengthens both accountability and integrity

within the system, providing a robust mechanism to prevent unauthorized access and protect against repudiation of actions.

When Confidential Computing (CC) is enabled in the COGNIT framework, it significantly strengthens the system's resilience against certain security threats. By executing sensitive operations within trusted execution environments (TEEs), CC ensures that both data and code remain protected from unauthorized access or tampering, even if the underlying host system is compromised. This capability helps mitigate threats such as remote code execution, alteration of execution flow in the frontend, and tampering with critical data processing, by guaranteeing code integrity and isolating sensitive computations. As a result, CC provides a robust layer of protection for the confidentiality and integrity of operations that cannot be achieved by traditional access control alone.

Closer look: Serverless Runtime Threat Analysis

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. 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 5.7 summarizes the COGNIT architecture and the interactions between the Device Client and the COGNIT Framework.

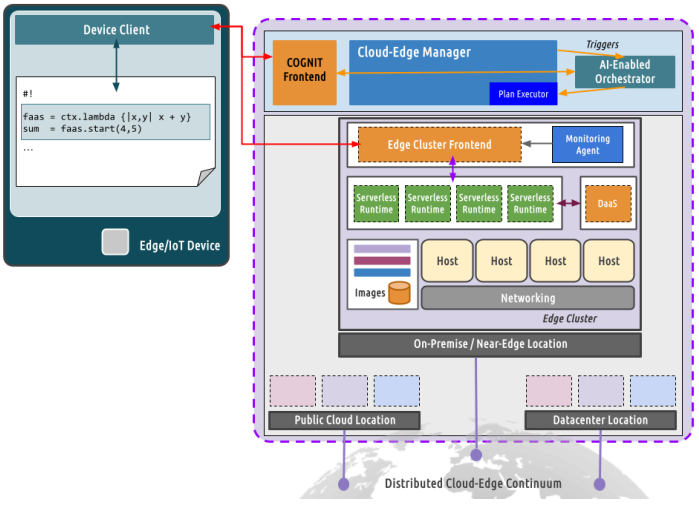

Figure 5.7. Device Client interaction with the COGNIT Platform.

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 5.8 presents the introduction of a MITM attack.

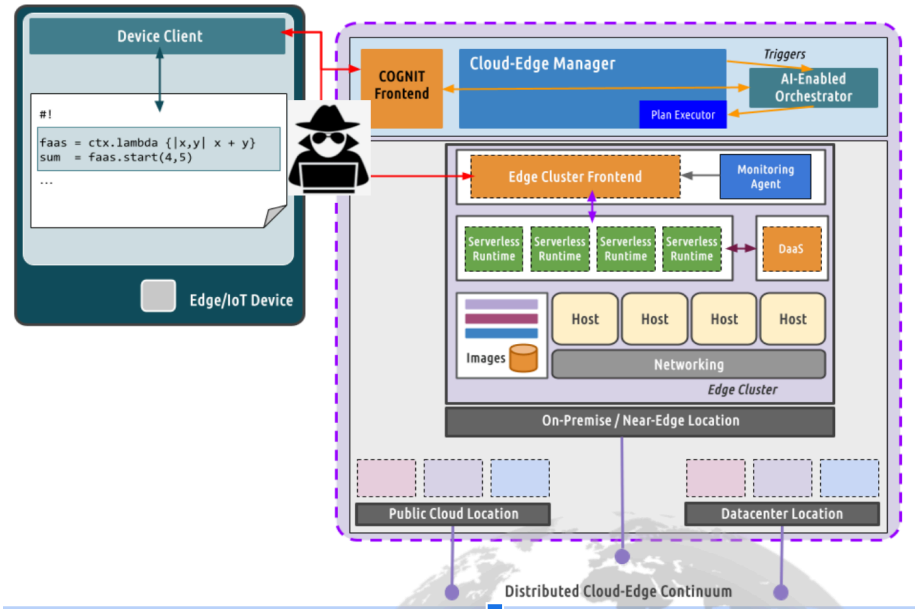

Figure 5.8. Man in the Middle attack

Our approach describes an attack flow, represented in Figure 5.8. 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 5.9 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.

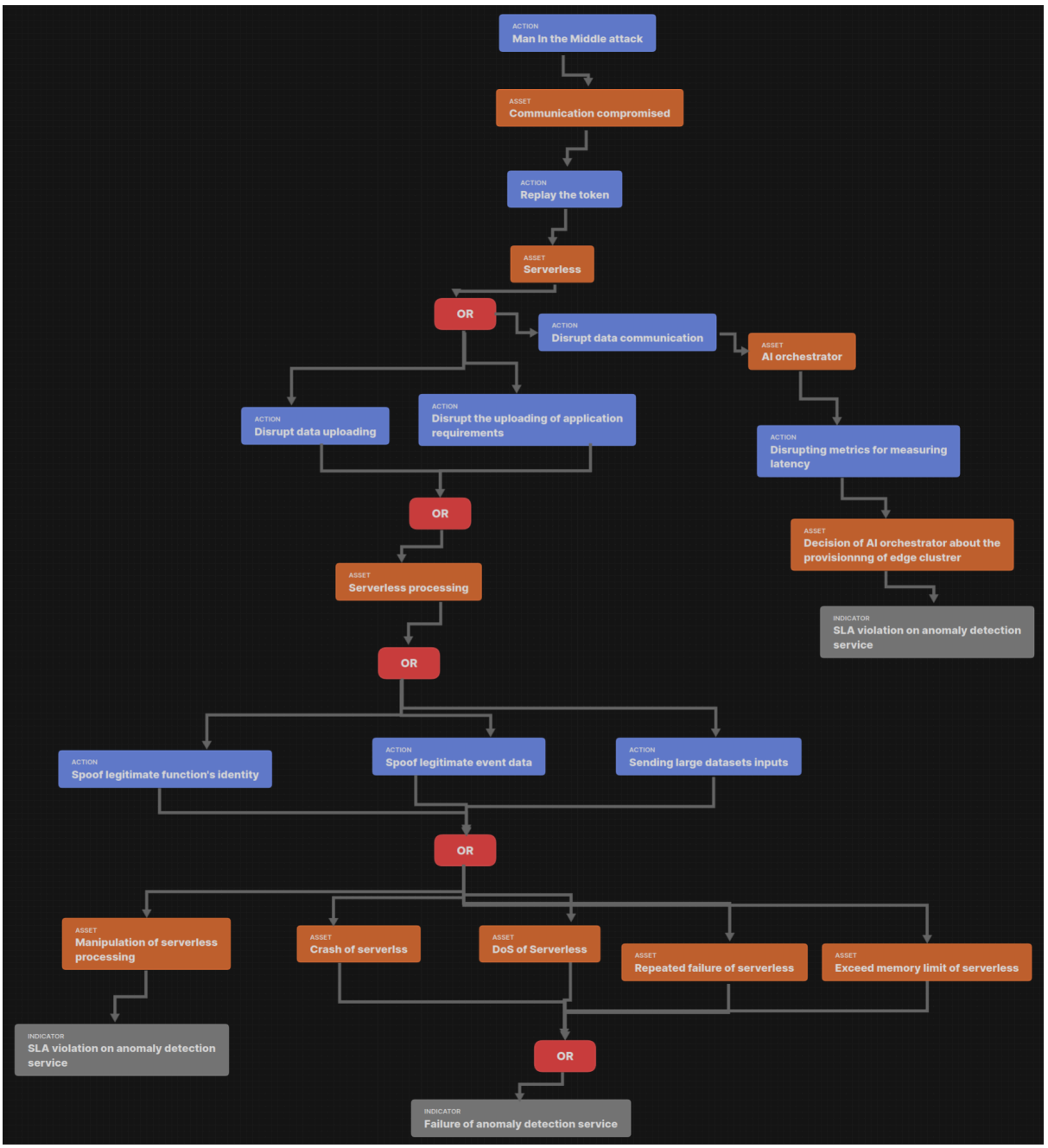

Figure 5.9. Attack flow diagram modeling the kill chain in the case of a Man In The Middle Attack.

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.

5.2. [SR6.1] Advanced Access Control

This section describes the requirements, architecture, component selection and deployment of an Identity and Access Management (IAM) mechanism to meet the needs of the COGNIT Framework and the specific challenges of Edge Computing and following the threat analysis described in the previous section and the risk analysis performed in WP2.

In a distributed and dynamic environment like COGNIT, where computing resources are spread across multiple sites or peripheral devices, securing communications and managing identities become critical challenges. In this regard, we focused our research on the requirements of distributed deployment, identity management, security and privacy, low latency, and scalability.

5.2.1 Requirements

The requirements for an IAM mechanism adapted to the COGNIT Framework:

- **Distributed Deployment:** The IAM mechanism must enable distributed deployment to meet the needs of an edge computing environment where resources are distributed across multiple sites or devices.
- **Identity Management:** The IAM mechanism must effectively manage the identities of devices, users, and components by assigning them specific accounts or using authentication flows tailored to these components to ensure appropriate authentication and authorization.
- **Security and Privacy:** It must provide robust security mechanisms, such as SSL/TLS encryption, to protect communications between edge computing devices and IAM servers, as well as key and certificate management features to ensure the confidentiality of identity and authentication data.
- **Low Latency:** The IAM mechanism should not introduce significant latency in authentication and authorization processes, minimising response times to meet the critical performance requirements of the edge computing environment.
- **Scalability:** It must be highly scalable to adapt to the scale of the edge computing infrastructure, supporting clustering to distribute the load and being capable of handling a large number of devices and concurrent users.

We have validated the three critical communication points within the COGNIT stack:

1. **Device Client to COGNIT Frontend:**

   Secure communication via JWT and SSL meets the security and privacy requirements for this critical interaction. JWT ensures secure verification of the device client's identity and authorization, while SSL encryption ensures the integrity and confidentiality of exchanged data.

2. **DeviceClient to Edge-Cluster Frontend:**

The use of SSL encryption to secure communication meets the security requirements for this connection. Additionally, authenticating cloud servers via OpenNebula, with a configuration similar to the serveradmin OpenNebula account, ensures the appropriate level of security and isolation needed in a multi-tenant environment.

3. **Device Client to Serverless Runtime:**

   The combination of JWT and SSL for communication between the Device Client and the Serverless Runtime meets the security and authentication requirements. JWT encapsulates the necessary authentication mechanisms, thereby improving identity and access management for the device client. SSL encryption ensures the confidentiality and integrity of exchanged data.

5.2.2 Authentication System Options and Choice for the COGNIT Framework

Given the distributed and performance-critical nature of the COGNIT Framework, designing a suitable Identity and Access Management (IAM) mechanism is essential to meet both security and operational requirements. The threat assessment conducted on the framework highlighted several potential vulnerabilities related to authentication, access control, and trust boundaries, which must be mitigated through a robust, low-latency authentication system.

Potential Authentication Mechanisms

Several mechanisms were considered to satisfy the framework's requirements:

1. Centralized OAuth 2.0 / OpenID Connect
   - Provides strong identity and authorization management for users and devices.
   - Well-supported in cloud-native environments.
   - Challenges: Centralized servers can become bottlenecks or single points of failure, potentially introducing latency in a highly distributed edge environment.
2. Mutual TLS (mTLS)
   - Ensures device and server authentication using certificates.
   - Strong cryptographic guarantees and confidentiality.
   - Challenges: Certificate distribution and management at scale across thousands of devices can be operationally complex and may impact edge latency.
3. JSON Web Tokens (JWT)
   - Self-contained tokens encode user/device identity and permissions.

- Easy to verify locally without contacting a central server.
- Challenges: Once issued, JWTs cannot be revoked easily, which can complicate dynamic access control in edge environments.

4. Biscuit Tokens
    - A modern token-based approach that provides decentralized, verifiable, and flexible authorization.
    - Tokens carry embedded capabilities and restrictions, allowing fine-grained access control.
    - Supports delegation, enabling edge nodes to issue temporary capabilities to other components without contacting a central authority for each operation.
    - Advantages for the COGNIT Framework:
        - Distributed Deployment: Each edge node can independently verify token validity, eliminating the need for a central authentication round-trip.
        - Low Latency: Local verification ensures minimal authentication delay, meeting real-time performance requirements.
        - Scalability: Tokens are self-contained, reducing load on central servers and supporting large numbers of devices and concurrent users.
        - Security and Privacy: Built-in cryptographic protection ensures the integrity and confidentiality of token contents, compatible with SSL/TLS communications and key management best practices.
        - Identity Management: Supports structured capabilities for devices, users, and components, providing fine-grained, auditable access control.

5.2.3 Biscuit Tokens

Considering the requirements of distributed deployment, low-latency authentication, and scalability, Biscuit tokens offer the most suitable mechanism for the COGNIT Framework. Unlike traditional centralized or static token systems, Biscuit allows for:

- Dynamic delegation of access rights across edge nodes.
- Local verification without repeated communication with a central authority, minimizing response times.
- Fine-grained control over device and user capabilities, directly addressing threat scenarios identified in the framework’s threat assessment.

By leveraging Biscuit tokens, the COGNIT Framework can provide a secure, scalable, and performance-optimized authentication and authorization system across the cloud–edge continuum.

The Biscuit token mechanism has been integrated with the COGNIT platform. This integration allows an even more advanced authorization scheme with the inclusion of a Keycloak server that makes use of the aforementioned token. In order to authenticate against the COGNIT Frontend, and hence to the whole COGNIT platform, included in the COGNIT frontend client's private API there is a specific method that allows the whole Device Client to be authenticated to the platform. Furthermore, in the state machine implemented on COGNIT, there are mechanisms to ensure at all times that the Device Client is authenticated, and otherwise make sure that it tries to authenticate before performing any other action towards the COGNIT platform.

5.3. [SR6.2] Confidential Computing Requirement

Description

COGNIT Framework can benefit significantly from Confidential Computing (CC), particularly in scenarios where sensitive data is processed outside traditional trusted environments.

First, Confidential Computing ensures that data remains protected not only at rest and in transit, but also during processing. By using secure hardware enclaves or trusted execution environments (TEEs), workloads executed at the edge—often in uncontrolled or semi-trusted locations—can run in an isolated and verifiable context. This greatly reduces the risk of data exposure, tampering, or unauthorized access.

Second, by using CC functions can be dynamically deployed across heterogeneous edge devices without compromising security. Confidential computing provides a consistent security baseline, allowing event-driven workloads to run close to the data source while maintaining strong confidentiality guarantees.

Finally, CC enhances trust, compliance, and scalability. Organizations can confidently offload sensitive processing to edge nodes, meet regulatory requirements, and scale serverless workloads without sacrificing privacy or integrity.

In order to test and demonstrate the Confidential Computing (CC) support and capabilities of the COGNIT Framework, we have enabled support for executing functions within a CC secured hardware enclave. This involved deploying Confidential Computing-enabled edge nodes using the framework with a CC enabled configuration. The CC testing scenario has been included in the Use Case 4 demonstration - as described in detail in Deliverable D5.6 - in order to study it, assess compatibility, performance, and potential benefits. For this scenario to be used, the sensitive function execution requirement must be enabled and passed to the COGNIT Framework. This requirement is a security strategy to enhance data protection and trust during sensitive processing tasks. According to the CC scenario of

UC4, memory-protection guarantees are provided when executing a function inside a Serverless Runtime (SR) on an AMD SEV–capable Edge Cluster that UC4 has deployed.