
A Cognitive Serverless Framework for the Cloud-Edge Continuum

# D3.3 COGNIT FaaS Model - Scientific Report - c

Version 1.0

11 November 2024

## Abstract

COGNIT is an AI-enabled Adaptive Serverless Framework for the Cognitive Cloud-Edge Continuum that enables the seamless, transparent, and trustworthy integration of data processing resources from public providers and on-premises data centers in the Cloud-Edge Continuum. The main goal of this project is the automatic and intelligent adaptation of those resources to optimise where and how data is processed according to application requirements, changes in application demands and behaviour, and the operation of the infrastructure in terms of the main environmental sustainability metrics. This document describes the research and development carried out in WP3 “Distributed FaaS Model for Edge Application Development” during the Third Research & Innovation Cycle (M16-M21), providing details on the status of a number of key components of the COGNIT Framework (i.e. Device Client, COGNIT Frontend, and Edge Cluster) as well as reporting the work related to supporting the Secure and Trusted Execution of Computing Environments.




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:** | 30/09/2024 |
| **Actual Date of Delivery:** | 11/11/2024 |
| **Lead Author:** | Idoia de la Iglesia (Ikerlan) |
| **Authors:** | Monowar Bhuyan (UMU), Malik Bouhou (CETIC), Aritz Brosa (Ikerlan), Christophe Ponsart (CETIC), Jean Lazarou (CETIC), Fátima Fernández (Ikerlan), Torsten Hallmann (SUSE), Johan Kristiansson (RISE), Marco Mancini (OpenNebula), Alberto P. Martí (OpenNebula), Philippe Massonet (CETIC), Nikolaos Matskanis (CETIC), Daniel Olsson (RISE), Mikel Irazola (Ikerlan), Álvaro Puente (Ikerlan), Thomas Ohlson Timoudas (RISE), Paul Townend (UMU), Iván Valdés (Ikerlan), Constantino Vázquez (OpenNebula), Daniel Clavijo (OpenNebula), Jorge Lobo (OpenNebula), Michal Opala (OpenNebula). |
| **Status:** | Submitted |

## Document History

| Version | Issue Date | Status[1] | Content and changes |
|---|---|---|---|
| 0.1 | 24/10/2024 | Draft | Initial Draft |
| 0.2 | 05/11/2024 | Peer-Reviewed | Reviewed Draft |
| 1.0 | 11/11/2024 | Submitted | Final Version |

## Peer Review History

| Version | Peer Review Date | Reviewed By |
|---|---|---|
| 0.1 | 30/10/2024 | Yashwant Singh Patel (UMU) |
| 0.1 | 05/11/2024 | Antonio Álvarez (OpenNebula) |

## Summary of Changes from Previous Versions

| |
|---|
| First Version of Deliverable D3.3 |

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

# Executive Summary

This is the third “COGNIT FaaS Model - Scientific Report” that has been produced in WP3 “Distributed FaaS Model for Edge Application Development”. It describes in detail the progress of the software requirements stated in deliverable D2.4 that have been active during the Third Research & Innovation Cycle (M16-M21) in connection with these main components of the COGNIT Framework:

**Device Client**

- **SR1.1** Interface with COGNIT Frontend
  *Implementation of the communication of the Device Client with the COGNIT Frontend.*
- **SR1.2** Interface with Edge Cluster
  *Implementation of the communication of Device Client with the Edge Cluster.*
- **SR1.3** Programming languages
  *Support for different programming languages.*
- **SR1.5** Security
  *Device Runtime must be secured.*

**COGNIT Frontend**

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

**Edge Cluster**

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

**Secure and Trusted Execution of Computing Environments**

- **SR6.1** Advanced Access Control
  *Implement policy-based access control to support security policies on geographic zones that define a security level for specific areas.*
- **SR6.2** Confidential Computing
  *Enable privacy protection for the FaaS workloads at the hardware level using Confidential Computing (CC) techniques.*

This deliverable has been released at the end of the Third Research & Innovation Cycle (M21), and will be updated with incremental releases at the end of each research and innovation cycle in M27, and M33.

# Table of Contents

# Abbreviations and Acronyms

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **API** | Application Programming Interface |
| **CFE** | COGNIT Frontend Engine |
| **CORS** | Cross Origin Resource Sharing |
| **CPU** | Central Processing Unit |
| **DaaS** | Data as a Service |
| **DC** | Device Client |
| **ECFE** | Edge Cluster Frontend Engine |
| **FaaS** | Function as a Service |
| **GDPR** | General Data Protection Regulation |
| **HOTP** | HMAC based One Time Password |
| **HTTP** | Hypertext Transfer Protocol |
| **HMAC** | Hash based Message Authentication Code |
| **HW** | Hardware |
| **IAM** | Identity and Access Management system |
| **IP** | Internet Protocol |
| **JSON** | Javascript Object Notation |
| **JWT** | JSON Web Token |
| **LDAP** | Lightweight Directory Access Protocol |
| **OAuth2.0** | Open Authentication 2.0 |
| **PE** | Provisioning Engine |
| **REST** | Representational State Transfer |
| **SAML** | Security Assertion Markup Language |
| **SDK** | Software Development Kit |
| **SPI** | Service Provider Interfaces |
| **SR** | Serverless Runtime (with no number) |
| **SRx** | Software Requirement (with a number associated, e.g.: SR1.1) |
| **SSL** | Secure Sockets Layer |
| **TEE** | Trusted Execution Environments |
| **TLS** | Transport Layer Security |
| **TOTP** | Time based One Time Password |
| **VM** | Virtual Machine |
| **YAML** | Yaml Ain’t a markup language |

# 1. Device Client

The Device Client uses the COGNIT library through which you can extract the DeviceRuntime class. This class will provide two methods with which the client device can communicate with the COGNIT Cloud-Edge Continuum. The first method, init(), will initiate communication with COGNIT by means of an exchange of requirements and authentication. Therefore, it will be necessary to provide to this method a valid authentication file with valid credentials.

Once the communication is initialised, it is possible to use the call() method. The call() method will allow the client device to upload functions to the COGNIT cloud and execute them from there. To do so, it is necessary to provide the function as well as the parameters with which the function is to be executed. An example demonstrating how this library is used is provided at the end of Section 1.4. The following table summarises how these two functions work:

| Description | Method | Parameters | Return Type |
|---|---|---|---|
| Authenticates the device with the CFE to get a valid biscuit token. Uploads the initial application requirements to the DaaS gateway. | init() | A dict python object containing the application requirements. | Nothing |
| Executes different actions (upload/ update/ delete app requirements, get ECFE endpoint…) depending on the internal state of the Device Runtime. | call() | Function to be offloaded (Callable type)<br><br>App requirements (dictionary type , optional) | Result code, Any |

**Table 1.1.** API definition of the DeviceRuntime class

**New architecture lifecycle management**

In order to support the management of the lifecycle of the new components introduced in this version of the architecture, another component, the Device Runtime State Machine, has been implemented.

Device Runtime State Machine adds an additional layer of abstraction to the Device Client. Instead of manually invoking the set of functionalities from the Cognit and Edge Cluster clients, the communication is done automatically based on the state of the communication. In other words, the Device Runtime State Machine now governs when specific actions, such as task offloading or requirement updates, are executed. Following this approach, the Device Client now only has to execute a few sets of functions that trigger transitions between different states, simplifying the overall communication flow.

Specifically the state machine operates around two core functions: offload_function() and update_requirements(), ignoring the complexity of the communication process.

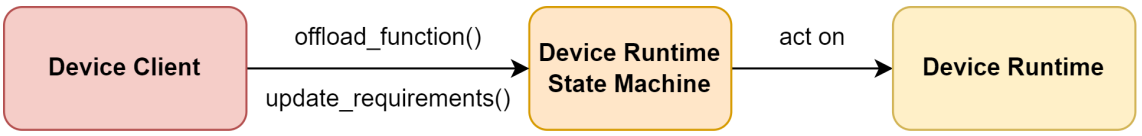


**Figure 1.1.** Device Client - Device Runtime State Machine workflow

The usage of these two functions serves as the catalyst for generating changes in the state of the state machine. Each state represents a distinct stage in the communication process between the two Frontends. For example, invoking offload_function() may trigger a transition from a waiting state to an offloading state, indicating that the device is actively transferring tasks to the Edge Cluster. Similarly, update_requirements() can update the necessary computational resources or other prerequisites, which may move the state machine to a negotiation or validation state. As a result, the Device Client is simplified, as it no longer needs to manage the entire communication process manually. Instead, the state machine handles transitions, event-driven changes, and the flow of information between the Frontends, enabling more efficient task execution and reducing potential errors from manual state management.

**Architecture & components of the Device Runtime State Machine**

The Device Runtime State Machine is composed of a set of four states. Each of these states represents a particular situation in the communication process between the Device Client and the COGNIT framework. The different states work as follows:

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

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

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

### Data Model of the Device Runtime State Machine

Two different data structures are used in the communication with the Device Runtime State Machine. On the one hand, the ExecResponse model is used to retrieve the result from the function offloading using offload_function(). On the other hand, the Scheduling model is the following structure to send the requirements in the update_requirements() function.

| Attribute | Description | Fields | Type |
|---|---|---|---|
| ExecResponse | Response of a generic execution, with its return code, result and error if applicable. | ret_code: ExecReturnCode<br>res: str (Optional)<br>err: str (Optional) | Inherits from pydantic BaseModel |
| Scheduling | String describing the policy applied to scheduling. Eg: “energy, latency” will optimise the placement according to those two criteria. | POLICY: str<br>REQUIREMENTS: str | Inherited from pydantic’s BaseModel |

**Table 1.2.** Data Model defining the Device Client’s interaction with the Device Runtime state machine.

### API & interfaces of the Device Runtime State Machine

As it was previously mentioned, there are two different ways to communicate with the Device Runtime State Machine: offload_function() and update_requirements().

The offload_function() functions allows the user to upload a function and process it taking into account the requirements that were previously uploaded. The user does not have to care about how the result is obtained. For now, there is no option to upload a function asynchronously, the execution of this function will block the thread until the result is given.

The other way to interact with the state machine and, therefore, with the COGNIT Frontend, is using the update_requirements() function. This function allows the user to upload the requirements that its application needs for processing its data. It can be called whenever the requirements are needed to change. This function will upload the requirements so later, when the user wants to offload a new function, a new Edge Cluster Frontend will attend those petitions. As well as it occurs with the offload_function(), it abstracts the complexity of how the requirements are uploaded.

| Description | Method | Parameters | Return Type |
|---|---|---|---|
| Perform the offload of a function to the COGNIT platform and wait for the result | offload_function | func: Callable<br>args: Any [Bundled as positional arguments] | ExecResponse. |
| Uploads or updates the current requirements for the Device Client | update_requirements | requirements: Scheduling | None. |

**Table 1.3.** API that defines the Device Runtimestate machine functions to perform actions in the COGNIT framework

## 1.1 [SR1.1] Interface with COGNIT Frontend

### Description

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

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

The COGNIT Frontend Client employs an internal flag (*'has_connection'*) to indicate the current connection status, allowing other modules within the Device Runtime to check this flag. Additionally, it maintains an internal variable that maps uploaded functions to the IDs assigned by COGNIT, thereby minimising the processing overhead associated with re-uploading existing functions.

### Architecture & Components

This deliverable introduces the first implementation of the Python class *Edge Cluster Client*. While this initial version is developed in Python, future iterations will also provide the client in the C programming language. This client does not make use of any other component to achieve the communication with the Edge Cluster.

### Data model

The data model of the interaction with the COGNIT Frontend Engine defines all the fields expected by the COGNIT Frontend Engine for requests and responses.
The *'UploadFunctionDaaS'* data model follows the structure defined in *'ExecSyncParams'* of the previous version.

| Attribute | Description | Fields | Type |
|---|---|---|---|
| Scheduling | Object containing the information of the application requirements. | FLAVOUR: str<br>MAX_LATENCY: int (optional)<br>MAX_FUNCTION_EXECUTION_TIME: float (optional)<br>MIN_ENERGY_RENEWABLE_USAGE: int (optional)<br>GEOLOCATION: str (required if MAX_LATENCY is defined, optional otherwise) | Inherited from pydantic's BaseModel |
| FunctionLanguage | String defining the language of the offloaded function. | PY = "PY"<br>C = "C" | Enum |
| UploadFunctionDaaS | Object containing the information (language,function and hash) about the function to be uploaded to the DaaS gateway. | LANG: FunctionLanguage<br>FC: str<br>FC_HASH: sr | Inherited from pydantic's BaseModel |
| EdgeClusterFrontendResponse | Object containing the information about the optimal Edge Cluster obtained from the COGNIT Frontend Engine. | ID: int<br>NAME: str<br>HOSTS: list[int]<br>DATASTORES: list[int]<br>VNETS: list[int]<br>TEMPLATE: dict | Inherited from pydantic's BaseModel |

**Table 1.4.** Data model of the COGNIT Frontend Client component**.**

### API & Interfaces

The COGNIT Frontend Client is composed of several private methods, as depicted in Table 1.5, which are abstracted from the user and are used to interact with the COGNIT Frontend Engine.

| Description | Method | Parameters | Return Type |
|---|---|---|---|
| Used to delete the application requirements using the id stored as a class variable. | _app_req_delete | None | Bool indicating if the app requirement has been deleted. |

| | | | |
|---|---|---|---|
| Get the application requirements using the id stored as a class variable. | _app_req_read | None | Scheduling type object containing the information about the application requirements. |
| Update the application requirements using the id stored as a class variable. | _app_req_update | new_reqs: Scheduling | Bool indicating if the app requirement has been updated. |
| Authenticates the device with the CFE to get a valid biscuit token. Uploads the initial application requirements to the DaaS gateway. | _authenticate | None | Biscuit_token: str |
| Check if GEOLOCATION field has a value for the cases that MAX_LATENCY is defined | _check_geolocation_valid | reqs: Scheduling | is_valid: bool |
| Get the endpoint of the optimal Edge Cluster. | _get_edge_cluster_address | None | String containing the endpoint. |
| Serialize and upload the function to the DaaS gateway. Add uploaded function id to class variable map. | _serialize_and_upload_fc_to_daas_gw | func: Callable | None |
| Perform the upload of the serialised function. | _upload_fc | fc: UploadFunctionDaas | func_id: int |
| Getter for the “has_connection” flag. | get_has_connection | None | Bool indicating if the COGNIT FEClient has connection with the COGNIT Frontend. |
| Perform the authentication and upload the initial app requirements | init | initial_reqs: Scheduling | Bool indicating the status of the initialization. |

| | | | |
|---|---|---|---|
| Setter for “has_connection” fla. | set_has_connection | new_value: bool | None |

**Table 1.5.** API definition of the Cognit Frontend Client component.

## 1.2 [SR1.2] Interface with Edge Cluster

### Description

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

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

In essence, the Edge Cluster Client is an HTTP client that interacts with the Edge Cluster using its endpoints. The usage of this client is hidden from the end user, as it is managed by the Device Runtime State Machine. The end user simply calls the *call()* function to retrieve the result of an offloaded function. The State Machine handles this method to ensure that the user receives the result of a function execution. Among other tasks, when calling the *call()* method, the State Machine will eventually contact the Edge Cluster to obtain the result.

### Architecture & components

This deliverable introduces the first implementation of the Python class *Edge Cluster Client*. While this initial version is developed in Python, future iterations will also provide the client in the C programming language. This client does not make use of any other component to achieve the communication with the Edge Cluster. However, it is initialised with information that has to be obtained from the COGNIT Frontend.

To instantiate an Edge Cluster Client, you need the token obtained during the initial communication with the COGNIT Frontend. Additionally, the communication with the COGNIT Frontend provides the address of the specific Edge Cluster that will handle the device client’s requests. Both the token and the address are essential for initialising the Edge Cluster Client.

### Data Model

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

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

**Table 1.6.** Data Model followed by the Edge Cluster Client

### API & interfaces

In this release, three key methods have been implemented for this client: *execute_function()*, *get_has_connection()*, and *set_has_connection()*, along with auxiliary functions that support their functionality.

1. The *execute_function()* method is responsible for executing a function on the Edge Cluster via a REST API. It sends two query parameters: *mode* (which defines the execution mode, such as synchronous or asynchronous) and *req_id* (that identifies with which requirements the function is going to be executed). The function will create a POST request to the endpoint *“<address>/v1/functions/<func_id>/execute”* that will trigger the execution of the defined function.

   The body of the request contains the serialised parameters required for the function's execution. Currently, only synchronous execution mode is supported, meaning that the client waits for the result of the function execution in real-time. If the client does not receive a response or receives an unexpected response, the connection is considered lost, and appropriate error handling should be implemented. Future support for asynchronous execution is planned but not yet available

2. *get_has_connection()* and *set_has_connection()*: These methods are responsible for retrieving and updating the client's current connection status. A disconnection is detected if the client receives an HTTP status code equal or higher than 400

(meaning that either a client error or a server error has happened during a request). The connection status is also considered during the creation of the client depending on the parameters received in the constructor.

Table 1.7 summarises what the explained methods do. Future versions of the client will expand functionality to include communication of the Device Client's current location and latency, aligning with the overall development objectives.

| Description | Method | Parameters | Return Type |
|---|---|---|---|
| Communicates with the Edge Cluster to start the execution of a function. | execute_function() | func_id: string<br>app_req_id: string<br>exec_mode: ExecutionMode<br>params: List[str] | result: ExecResponse |
| Get the current connection status for the client. | get_has_connection() | - | has_connection: Bool |
| Sets the connection status of the client | set_has_connection() | connection_status: Bool | - |

**Table 1.7.** API defining the Edge Cluster Client interaction with the Edge Cluster

## 1.3 [SR1.3] Programming languages

### Description

Being the first version of the v2 of the architecture, all the developments have been focused on the Python version of the Device Client. Its C version remains to be designed and developed in the following development cycles to comply with the needs of the v2 architecture. Several preliminary discussions have taken place about how to structure the v2 of the C version of the client.

## 1.4 [SR1.5] Security

### Description

In this development cycle the integration of the Biscuit token has been integrated for enhancing the authorization procedures of COGNIT. This integration would also allow an even more advanced authorization scheme with the inclusion of a Keycloak server that makes use of the aforementioned token.

### Architecture & components

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

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

### API & interfaces

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

**Table 1.8.** Authentication API with single method

### Device Client usage example:

```
from cognit import device_runtime

# Functions used to be uploaded
def sum(a: int, b: int):
    print("This is a test")
    return a + b

def multiply(a: int, b: int):
    print("This is a test")
    return a * b

# Execution requirements, dependencies and policies
```

```
TEST_REQS_INIT = {
      "FLAVOUR": "Energy",
      "MAX_FUNCTION_EXECUTION_TIME": 2.0,
      "MAX_LATENCY": 25,
      "MIN_ENERGY_RENEWABLE_USAGE": 85,
      "GEOLOCATION": "IKERLAN ARRASATE/MONDRAGON 20500"
}

# Other requirements used
TEST_REQS_NEW = {
      "FLAVOUR": "SmartCity",
      "MAX_FUNCTION_EXECUTION_TIME": 3.0,
      "MAX_LATENCY": 45,
      "MIN_ENERGY_RENEWABLE_USAGE": 75,
      "GEOLOCATION": "IKERLAN ARRASATE/MONDRAGON 20500"
}

try:
    # Instantiate a device Device Runtime
    my_device_runtime =
device_runtime.DeviceRuntime("./examples/cognit-template.yml")
    # Send requirements to
    my_device_runtime.init(TEST_REQS_INIT)
    # Offload and execute sum function
    return_code, result = my_device_runtime.call(sum, 1, 3)
    print("Status code: " + str(return_code))
    print("Sum result: " + str(result))
    # It is also possible to update the requirements
    # when offloading a function or calling again the init function
    # Equivalent: my_device_runtime.call(multiply, 2, 3,
new_reqs=TEST_REQS_NEW)
    my_device_runtime.init(TEST_REQS_NEW)
    return_code, result = my_device_runtime.call(multiply, 2, 3)
    print("Status code: " + str(return_code))
    print("Multiply result: " + str(result))
    # Lets offload a function with wrong parameters
    return_code, result = my_device_runtime.call(multiply,
"wrong_parameter", 3)
    print("Status code: " + str(return_code))
    print("Multiply result: " + str(result))

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

# 2. COGNIT Frontend

## 2.1 [SR2.1] COGNIT Frontend

### Description

The COGNIT Frontend  is a software component that acts as the single point of contact for any Application Device Runtime that requests access to the COGNIT Framework to offload computation through the FaaS paradigm. As stated in the D2.4 Framework architecture document, requirement SR2.1 is satisfied by this component.

This is a new component needed in the revised v2.0 of the architecture used in this development cycle.

### Data Model

This component needs to handle different abstractions to serve as the interface between the Device Runtime and the other components of the COGNIT stack. Therefore the data model needs to capture the different components that it needs to interact with.

- The ***Application Requirements*** **object** represents all the requirements by the application running in the Device Client: flavour of the Serverless Runtime, latency requirements, energy consumption, capacity,etc. A full schema of this object can be seen in Figure 2.1. This information is persisted in the Cloud-Edge Manager document pool.
- The ***Function*** **object** represents the function to be offloaded to the Serverless Runtime. It contains the language where the function is written as well as the function itself, encoded and hashed. A full schema can be seen in Figure 2.2. This information is persisted in the Cloud-Edge Manager document pool.
- The ***Assigned Edge Cluster*** **Frontend object** represents a Cloud-Edge Manager cluster. It contains information regarding the infrastructure that belongs to the cluster, like networking, hypervisor hosts and datastores, as well as the Endpoint where the Edge Cluster Frontend is running. A full schema describing the cluster is borrowed from OpenNebula.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "FLAVOUR": {
      "type": "string",
      "default": "Nature",
      "description": "String describing the flavour of the Runtime. There
is one identifier per DaaS and FaaS corresponding to the different use
cases"
    },
    "MAX_LATENCY": {
      "type": ["integer", "null"],
```

```
      "default": 10,
      "description": "Maximum latency in milliseconds"
    },
    "MAX_FUNCTION_EXECUTION_TIME": {
      "type": ["number", "null"],
      "default": 1.0,
      "description": "Max execution time allowed for the function to
execute"
    },
    "MIN_ENERGY_RENEWABLE_USAGE": {
      "type": ["integer", "null"],
      "default": 80,
      "description": "Minimum energy renewable percentage"
    },
    "GEOLOCATION": {
      "type": ["string", "null"],
      "default": null,
      "description": "Scheduling policy that applies to the requirement"
    }
  },
  "required": ["FLAVOUR"],
  "additionalProperties": false,
  "if": {
    "properties": {
      "MAX_LATENCY": {
        "type": "integer",
        "not": {"const": 10}
      }
    },
    "required": ["MAX_LATENCY"]
  },
  "then": {
    "required": ["GEOLOCATION"]
  }
}
```

**Figure 2.1.** *Application Requirements* Object Description

➔ Function object:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "LANG": {
      "type": "string",
      "enum": ["PY", "C"],
      "description": "Programming Language of the function"
    },
```

```
    "FC": {
      "type": "string",
      "description": "Function bytes serialized and encoded in base64"
    },
    "FC_HASH": {
      "type": "string",
      "description": "Function contents hash. Acts as a function ID."
    }
  },
  "required": ["LANG", "FC", "FC_HASH"],
  "additionalProperties": false
}
```

**Figure 2.2.** *Function* Object Description

The COGNIT Frontend runs as a service, exposing a REST interface. This service, as well as other aspects of the behaviour of the whole component, can be configured using a YAML file (*/etc/cognit-frontend.conf*) described in Table 2.1.

| Attribute | Value |
|---|---|
| host | IP to which the COGNIT Frontend will bind to listen for incoming requests. |
| port | Port to which the COGNIT Frontend will bind to listen for incoming requests. Defaults to 1338. |
| one_xmlrpc | OpenNebula daemon contact information. |
| ai_orchestrator_endpoint | AI-Enabled Orchestrator endpoint. |
| log_level | *uvicorn* logging level. |

**Table 2.1.** COGNIT Frontend Server Configuration File

### Architecture

The COGNIT Frontend features five different submodules. The high level view of the architecture, laying out these submodules, is depicted on Figure 2.3.

- REST API module. This module is in charge of the secure communication between the Device Clients and the COGNIT Front-end, which is the entry point for all the functionality offered by the COGNIT stack.
- Auth Manager. This module handles the Biscuit Token based authentication used in COGNIT. It is responsible for issuing and verifying the tokens, as well as exposing the public key for other COGNIT components for them to verify tokens.

- AI-Enabled Orchestrator Client. Interface for communicating with the AI-Enabled orchestrator API.
- Application Requirements Handler. Takes care of the different management of Application Requirements CRUD operations as OpenNebula documents.
- Function Handler. Takes care of the different management of Function CRUD operations as OpenNebula documents.
- Cloud-Edge Manager Client. Logic built on top of the python OpenNebula client, abstracting some of authentication, document management logic and general API Call handling.

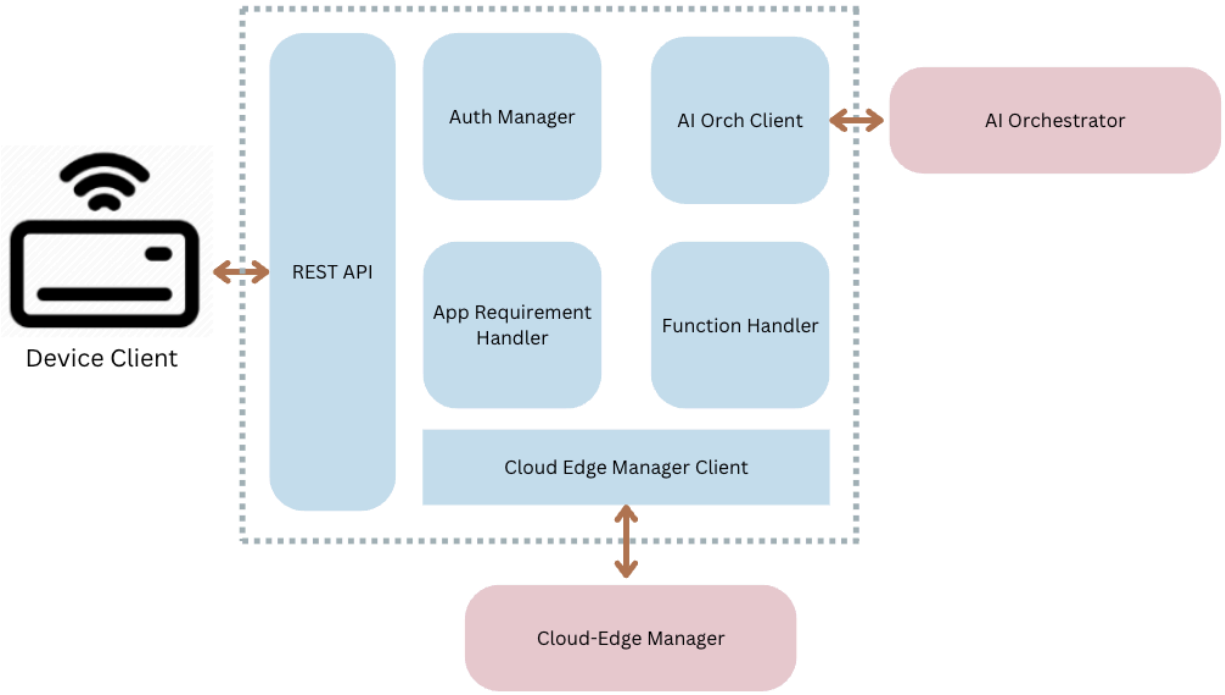


**Figure 2.3.** COGNIT Frontend High Level Architecture

### API & Interfaces

The API exposed by the COGNIT Frontend is a REST API that defines the interface from a given Device Client to consume the functionality exposed by the COGNIT stack. This API is specified in Table 2.2. The authorization is based on Biscuit public cryptography.

| Action | Verb | Endpoint | Request Body | Response |
|---|---|---|---|---|
| Authenticate to COGNIT Frontend | POST | /v1/authenticate | Sends credentials via HTTP basic auth | Status code 201 (Created) with the created Biscuit token |
| Get token public key | GET | /v1/public_key | JSON representation of the public key | Status code 200 (OK) with the public key string |

| | | | | |
|---|---|---|---|---|
| Upload application requirements | POST | /v1/app_requirements | JSON representation of requirements' object (with Biscuit token in the header) | Status code 200 (OK) with the created app requirements ID |
| Update application requirements | PUT | /v1/app_requirements/{id} | JSON representation of requirements' object (with Biscuit token in the header) | Status code 200 |
| Get application requirements | GET | /v1/app_requirements/{id} | - (with Biscuit token in the header) | Status code 200 (OK) with the application requirements document |
| Delete application requirements | DELETE | /v1/app_requirements/{id} | - (with Biscuit token in the header) | Status code 204 |
| Get assigned Edge Cluster Object | GET | /v1/app_requirements/{id}/ec_fe | - (with Biscuit token in the header) | Status code 200 (OK) with the Edge Cluster OpenNebula object |
| Upload functions and global user data | POST | /v1/daas/upload | JSON representation of function and/or global user data to upload (with Biscuit token in the header) | Status code 200 (OK) with the file ID object. |
| Get the file ID was already created | GET | /v1/daas/exist_file/{file_id} | - | Status code 200 (OK) with the file ID object. |
| Delete existing file ID | DELETE | /v1/daas/del_file/{file_id} | - | Status code 204 (No Content) if successful. |

**Table 2.2.** COGNIT Frontend API Specification

### Authorization

The COGNIT Frontend is a stateless component. All authorization from the Device Client is delegated to the Cloud-Edge Manager, which validates using biscuit as an authorization backend. This implies that Device Clients must have access to a user credential that is valid in the Cloud-Edge Manager in order to interact with the COGNIT Frontend.

After the Device Client authenticates with an existing user credential, a token is issued to it as a response to the authentication call. Further requests on other COGNIT components serving the device client are authorised with this token. The legitimacy of the token can be verified with the public key that the COGNIT Frontend shares. This key has a counterpart

private key that is used to generate the tokens, which is only known by the COGNIT Frontend and is regenerated every time the COGNIT Frontend starts.

**Documentation**

Two guides are available to install, configure and operate a COGNIT Frontend:

- The Administrator Guide covers the installation of the component, including instructions to install dependencies. It will also cover the configuration of the service, mostly by means of the provisioning-engine.conf configuration file, to configure the server and also the connection with the Cloud-Edge Manager. Hints and best practices for the management of the service will also be available for administrators in the guide.
- The User Guide covers the use of the COGNIT Frontend by the Device Client. It will state all the needed information that the Device Client must know, like the COGNIT Frontend endpoint and the Cloud-Edge Manager credentials.

# 3. Edge Cluster

## 3.1 [SR3.1] Edge Cluster Frontend

### Description

The Edge Cluster Frontend is the entrypoint for offloading functions from the Device Client in each Edge Cluster. It acts as a load balancer, to handle the redirection to the correct Serverless Runtime running within the particular Edge Cluster. As stated in the D2.4 Framework architecture document, requirement SR3.1 is satisfied by this component.

### Architecture

The Edge Cluster Frontend key functionality is to proxy the connection from the Device Clients to a particular Serverless Runtime running in the Cloud-Edge Manager Edge Cluster to which the Edge Cluster Frontend is binded to. Its main goal will be coherently load balancing the requests that the Frontend of this particular Edge Cluster is receiving. This load balancing is based on CPU usage. Functions are offloaded to the Serverless Runtimes with the lowest CPU usage, provided they have the appropriate flavour.

The Edge Cluster Frontend architecture is depicted in Figure 3.1, featuring the following modules:

- REST API: REST API module. This module is in charge of the secure communication between the Device Clients and the COGNIT Edge Cluster Front-end, which is the entry point for offloading function executions.
- Auth Manager. This module is responsible for acquiring the public key for token verification purposes from the COGNIT Frontend, and verifying the tokens issued by the device client in the requests.
- Execution Handler. Responsible for the execution of a function according to the specifications of the device client. Offloads the execution to a given Serverless Runtime.
- Load Balancer. Selects the ideal Serverless Runtime out of the Serverless Runtime VMs running on the Edge Cluster for which the COGNIT Edge Cluster Frontend is responsible for.

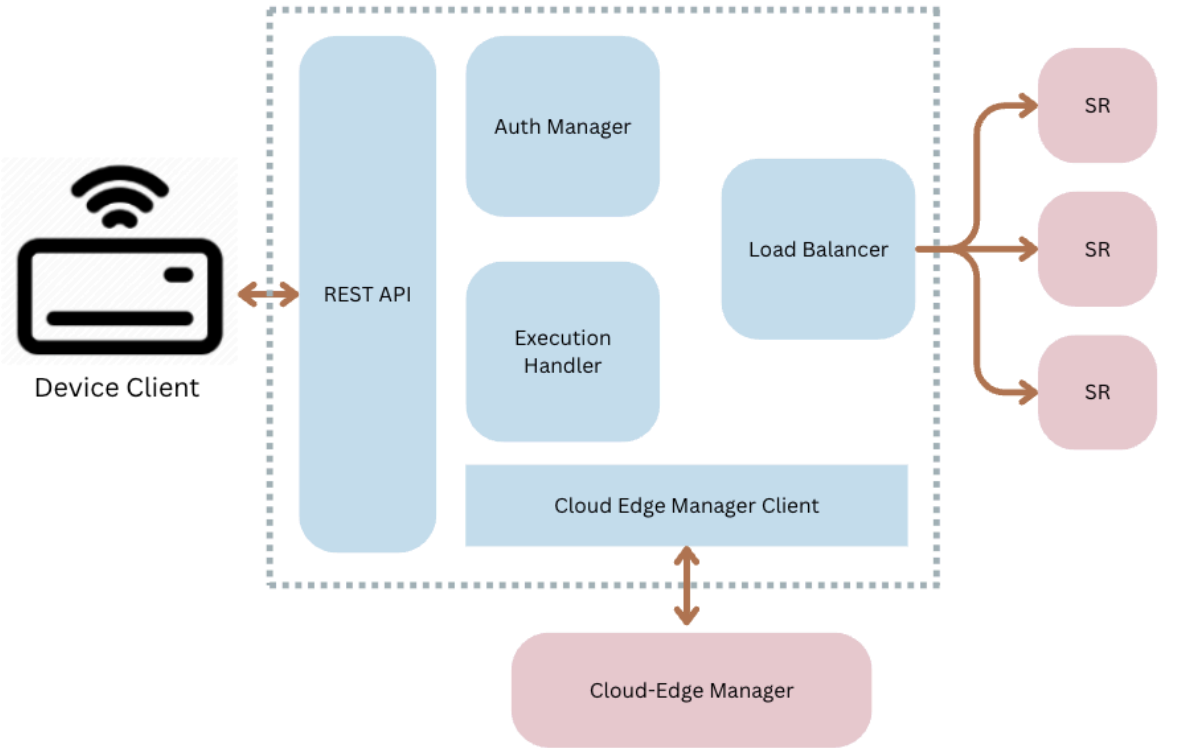


**Figure 3.1.** Edge Cluster Frontend High Level Architecture

### Data Model

This component handles the **Function Parameters** as input in the request body, as a **list of strings** containing the serialised parameters by each device runtime. Other parameters admitted are expected as path parameters:

- **Function ID** (an **Integer**): Document ID of a Function previously uploaded to the Cloud-Edge Manager document pool using the COGNIT Frontend.
- **Execution Mode** (**Enum**, can be **async** or **sync**): On sync mode, the execution result will be returned to the device client as soon as it finishes, as a response to the execution request. On async mode, an execution id is returned on the response, and the execution is performed on the Serverless Runtime. The status of the execution can then be queried afterwards.
- **App Requirement ID** (an **Integer**): Document ID of an Application Requirement previously uploaded to the Cloud-Edge Manager document pool using the COGNIT Frontend.

The Edge Cluster Frontend runs as a service, exposing a REST interface. This service, as well as other aspects of the behaviour of the whole component, can be configured using a YAML file (*/etc/cognit-edge_cluster_frontend.conf*) described in Table 3.1.

| Attribute | Value |
|---|---|
| host | IP to which the COGNIT Frontend will bind to listen for incoming requests. |
| port | Port to which the COGNIT Frontend will bind to listen for incoming requests. Defaults to 1338. |
| one_xmlrpc | OpenNebula daemon contact information. |
| oneflow | OpenNebula multi VM daemon contact information. |
| ai_orchestrator_endpoint | AI-Enabled Orchestrator endpoint. |
| log_level | uvicorn logging level. |

**Table 3.1.** Edge Cluster Frontend Server Configuration File

### API & Interfaces

This component exhibits an API enabling the management of remote functions to be executed in the Serverless Runtime load balanced by the Edge Cluster Frontend. Two endpoints are exposed, to execute and add Device Client related metrics. A description of the endpoint is provided in Tables 3.2 and 3.3, as well as an example of the *execute* endpoint.

| Type | Description |
|---|---|
| HTTP method | POST |
| Query Parameters | **`app_reqs_id`**, mode={sync \| async } |
| HTTP Header | **`biscuit_token`**: Authorization biscuit token for performing the action. |
| HTTP Request body | **`params`**: A list containing the params encoded in base64 |
| HTTP Response Code | **200:** Success |
| | **400:** Bad request |
| | **405:** Not allowed |
| HTTP Response body | **result:** Serverless Runtime response model |

**Table 3.2.** */v1/functions/{id}/execute* Edge Cluster Frontend Endpoint

***Request:***

*POST /v1/functions/32/execute?app_req_id=28&mode=sync*

```
 {
      “params”: ["gAVLAi4=", "gAVLAy4=", "gAVLBC4="]
 }
```

***Response:***

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

| Type | Description | Comments |
|---|---|---|
| HTTP **method** | POST | Upload device client connection metrics |
| HTTP Request body | JSON with metrics | No model enforced yet, planned for the next development cycle. |
| HTTP Response Code | **200:** Success | |
| | **400:** Bad request | |
| | **404:** Not found | |

**Table 3.3.** */v1/device_metrics* Edge Cluster Frontend Endpoint

The load balancer that is done within the Edge Cluster Frontend is configurable only by the administrator of the concerned Edge Cluster. The user will only interact with this component through a particular Device Client.

### Documentation

This component will be administered by the Edge Cluster system admin. Admin documentation is available with contents to deploy, configure and maintain the component.

### Tests

This component features tests that cover the following functionality:

- the biscuit public key can be gathered from the COGNIT Frontend.
- a function can be offloaded to a Serverless Runtime.

## 3.2 [SR3.2] Secure and Trusted Serverless Runtimes

### Description

In the current development cycle minor changes have been performed to this component, however from architecture standpoint its placement has changed quite considerably. With the inclusion of Edge Clusters concept, the Serverless Runtimes, or FaaS execution units are being deployed in each Edge Cluster, and are managed by the Edge Cluster, which will act as a proxy to any Device Client trying to access any resource in that particular Edge Cluster.

The amount of Serverless Runtimes deployed in the Edge Cluster needs to be coherent with the demand of resources on that location of the infrastructure, which will need to be managed at the Edge Cluster level, but also at COGNIT's global management level (COGNIT Frontend scope).

In order to be able to make decisions, AI-Enabled Orchestrator at COGNIT global level, and Edge Cluster at Edge Cluster level need to be aware of function execution information. For enabling this fact, the Prometheus exporter integrated in the Serverless Runtime's code has been enhanced to expose requirement ID (reqs_id) that belongs to the execution of the particular function.

The metric labels provide additional information about it through the mentioned Prometheus exporter, being:

- ➔ **end_time**=*"Mon Mar 18 11:23:13 2024"*, a timestamp that defines when the function finished executing.
- ➔ **func_hash**=*"dcf7b3aafca4b048d63c5b296f76e3988e44f9592d0b732fb8e7b0ae5f2c26cb"*, the hash of the function that defines the bytecode of the function that was offloaded.
- ➔ **reqs_id**= Integer value that describes to which requirement ID does correspond to the execution of the current function.
- ➔ **func_type**= Either *"sync"* or *"async"* depending on the type of function.
- ➔ **param_l_0**=*"xx"*, The list of parameters' size in bytes.
- ➔ **param_l_n**=*"yy"*, The nth parameter's size in bytes.
- ➔ **start_time**=*"Mon Mar 18 11:23:06 2024"*, a timestamp that states when the function execution started.
- ➔ **vm_id**=*"1548"*, the VM_ID identifies in which Serverless Runtime within the COGNIT infrastructure the function was executed.

This information will be available at Edge Cluster level, and will be aggregated using Thanos or other metric aggregator solution to have all the information of different Edge Clusters and global or COGNIT Frontend level.

### Architecture & Components

The Serverless Runtime provides a public FastAPI [2]REST Server that listens to FaaS requests. As illustrated in Figure 3.2, multiple components are involved in the execution of the task offloading function, this is the same as in the v1 of the architecture:

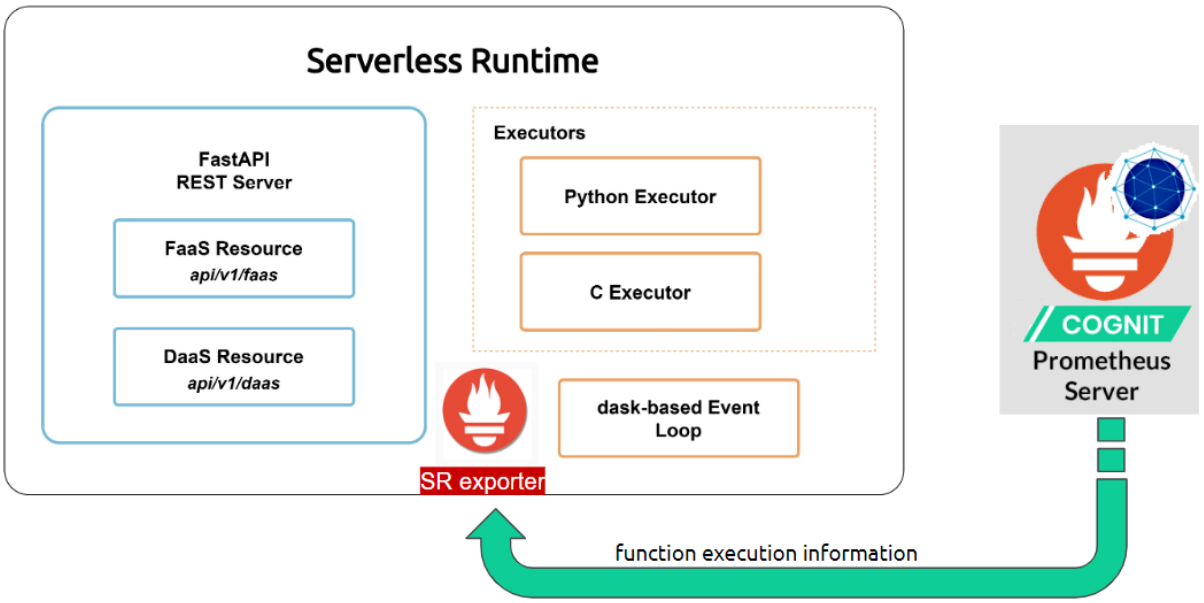


**Figure 3.2.** Block Diagram of Serverless runtime modules.

However in the next development cycle, *C Executor* will be dropped, allowing executions only through the *Python Executor*, and the C version of Device Client will need to be designed to be able to use this Executor in a standardised way from any other language in which Device Client could be developed (although in the scope of this project only C version will be developed).

As it was in the v1 architecture, the FAST API REST Server is accessible to the user and makes use of the functionalities given by the private API components, which are abstracted from the user for convenience.

### Data Model

The only modification in this Data Model with respect to the earlier version, was the addition of reqs_id field to comply with the needs of the v2 architecture of COGNIT (highlighted in bold letters):

| Attribute | Description | Value |
|---|---|---|
| lang | String describing the programming language of the code to be offloaded | Base64 string. |

[2] https://fastapi.tiangolo.com

| | | |
|---|---|---|
| fc | String describing the function to be offloaded coded in base64. | Base64 string. |
| fc_hash | Defines the hash of the function, which could be used as its identifier. | String, with HEX characters. |
| reqs_id | Specifies which requirement ID corresponds to the execution of this function. | Int |
| params | Array of strings describing the in/out parameters of the function coded in base64. | Array of base64 strings. |
| result | String describing the result of the function to be offloaded with the parameters given. | String. |
| faas_uuid | String describing the UUID of the task to process asynchronously. | String. |
| state | String describing the execution of the function. | WORKING,<br>READY,<br>FAILED. |

**Table 3.4.** Data model showing the data structures of the Serverless Runtime.

### API & Interfaces

The API of the Serverless Runtime has not changed in this development cycle.

# 4. Secure and Trusted Execution of Computing Environments

### Threat Model

In order to support the risk analysis, we performed a threat assessment on the framework architecture by creating its threat model.Our work is the follow-up of the one reported in the document named “D3.1 COGNIT FaaS Model - Scientific Report - a”, section 4 (4. Secure and Trusted Execution of Computing Environments). In this document we take into account the new architecture of the framework. We also adopt a more concrete and precise perspective by relying on a new threat analysis tool.

To create a threat model we need to look at the framework and answer to the following 4 questions [Threat Modeling Manifesto]:

1. What is the framework?
2. What can go wrong?
3. What are we going to do about it?
4. Did we do a good job?

We address the questions by

1. Describing the architecture framework so that we have a good picture to produce the threat model, see the The COGNIT Framework Architecture section.
2. Based on the knowledge of the framework, we can then try to find what can go wrong with all the components and the flows between the components that we identified, see the Identified Threats section.
3. The next step is to decide what we can do to either eliminate threats or to mitigate the weaknesses they represent.
4. Lastly, review that we did not forget anything in collaboration with the team in charge of the framework architecture.

All the information is collated in documents making up the threat model.

We decided to use a tool named Microsoft Threat Modeling Tool for several reasons:

- Drawing diagrams of an IT system is very easy.
- Each element that we place on the diagram has several properties which are used by the tool to make an analysis and produce a list of known threats that are relevant.
- Managing the threats, like marking them as mitigated, not applicable, etc. is another key feature.
- It supports updating the system definition during a project lifecycle, so that it can update the list of threats.
- It serves as a follow-up and threats management tool.
- It produces comprehensive reports which enables to improve communication with all stakeholders.

**Threat Modelling Definition**

Threat modelling first focuses on describing the system we build. As we said, we identify all the components and the flows but we also identify trust boundaries. When a flow comes from a component from its own "trust network" to another network, it crosses a change of security zone, it is said to cross a trust boundary.

Once we know the system, we analyse all the elements. We try to determine the weaknesses that can lead to threats that malicious people can exploit. Note that the consequences related to threats can be due to errors and not attacks per se.

The process of identifying the threats can use a classification proposed by Microsoft named STRIDE. The tool we used does analyse the system using this classification.

The table below describes what the STRIDE acronym stands for:

| | **Threat** | **Violates** | **Description** |
|---|---|---|---|
| S | spoofing | authentication | pretending to be someone or something else. |
| T | tampering | integrity | making changes to data. |
| R | repudiation | non-repudiation | pretending not to have done something when no evidence can challenge it. |
| I | information disclosure | confidentiality | release of secure or private/confidential information. |
| D | denial of service | availability | attack making a service temporarily or permanently unavailable. |
| E | elevation of privilege | authorisation | gain elevated access to protected resources. |

**Table 4.1:** STRIDE Acronym Meaning

The analysis process using the STRIDE classification consists in looking at each element of the system and checking if it is affected by one of the STRIDE threat categories.

The next step is to assess the risks of the threats being exploited and evaluate the extent of the damage so that we can:

- change the system to remove the threat.
- change the system in a way it is mitigated.
- accept the risk, leave the system unchanged.

Each time the system is changed we need to update the analysis to reflect the new system.

**The COGNIT Framework Threat Model**

The following diagram presents the global view of the COGNIT Framework (please refer to the Deliverable D2.4 for the details about the COGNIT Framework Architecture).

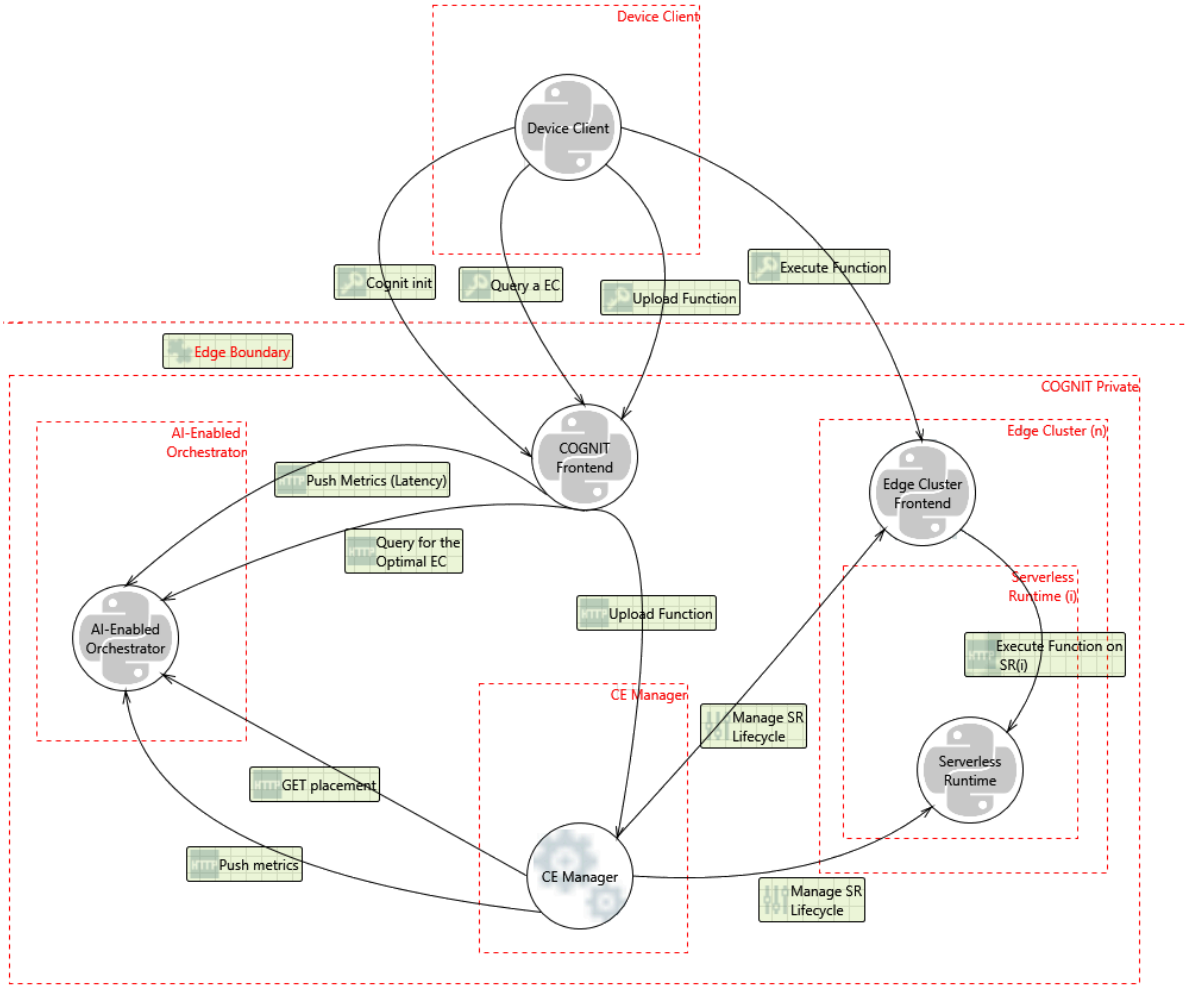


**Figure 4.1:** COGNIT Framework Threat Model Diagram

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[3], if the credentials are valid.

[3]Biscuit JWT token

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).

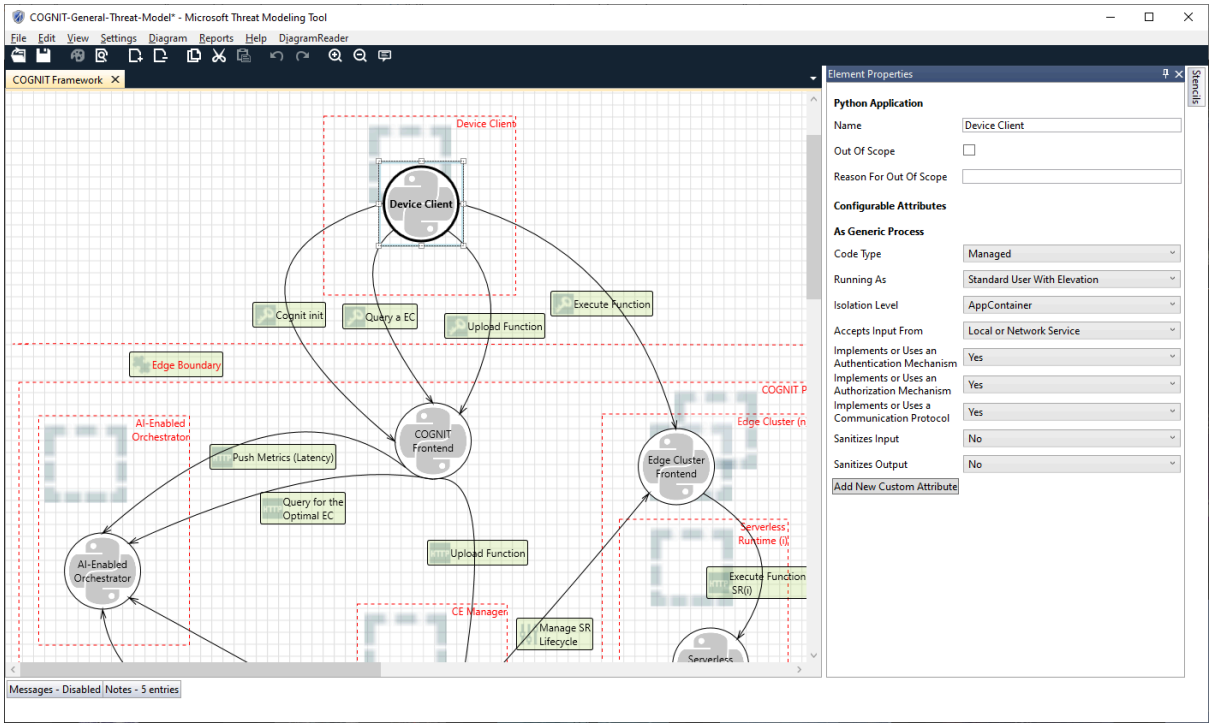


**Figure 4.2:** Microsoft Threat Modeling Tool

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.

- 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
- 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. |
| | COGNIT Component crashes, halts, stops or runs slowly. |
| E | COGNIT Component 1 may be able to impersonate the context of COGNIT Component 2 in order to gain additional privilege. |
| | 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. |

**Table 4.2:** 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).

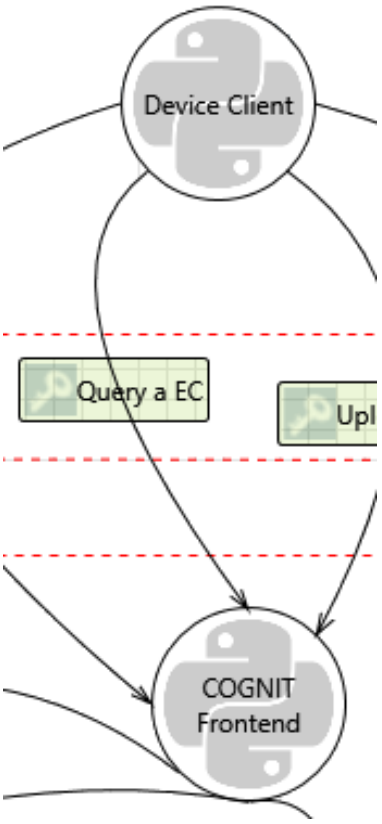


**Figure 4.3:** The *Query a EC* Flow

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

Properties for the Device Client component.

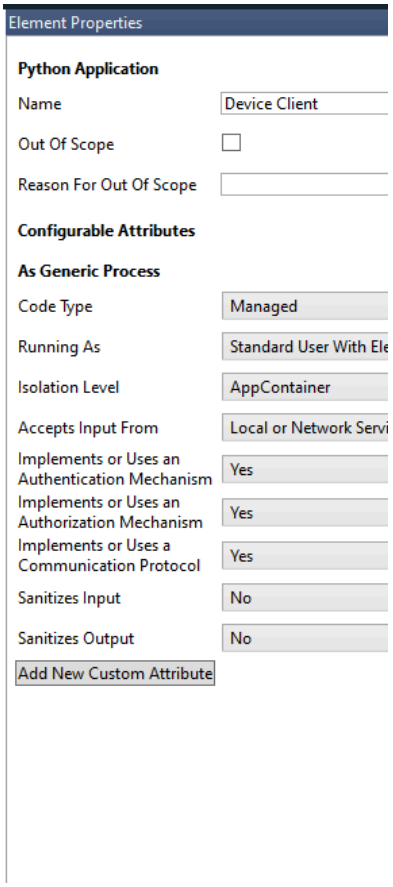


**Figure 4.4:** Device Client Properties

Properties for the COGNIT Frontend component.

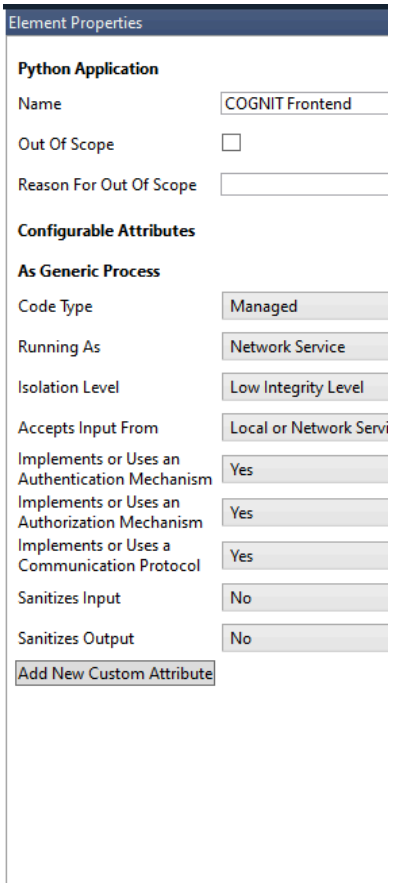


**Figure 4.5**: COGNIT Frontend Properties

Properties for the *Query a EC* flow.

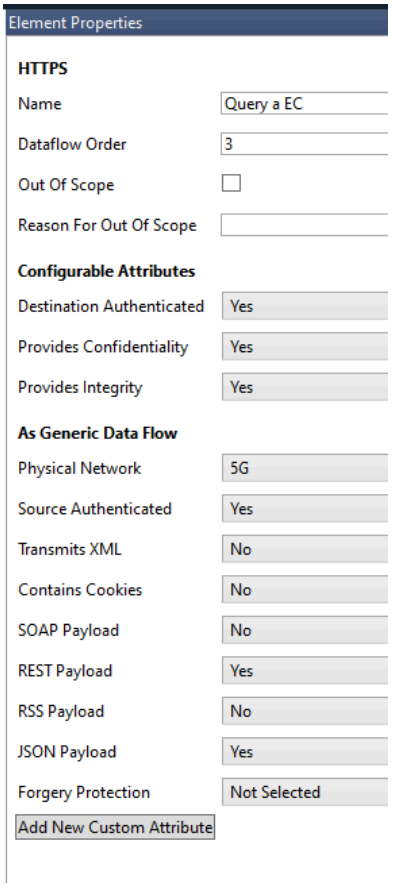


**Figure 4.6:** Query a EC Flow Properties

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 | Not Started |
| 2 | Tampering | JavaScript Object Notation Processing | Not Started |
| 3 | Tampering | Replay Attacks | Not Started |
| 4 | Repudiation | Potential Data Repudiation by COGNIT Frontend | Not Started |
| 5 | Information Disclosure | Weak Authentication Scheme | Not Started |

| # | Category | Threat | Status |
|---|---|---|---|
| 6 | Denial Of Service | Data Flow Query a EC Is Potentially Interrupted | Not Started |
| 7 | Denial Of Service | Potential Process Crash or Stop for COGNITt Frontend | Not Started |
| 8 | Elevation Of Privilege | COGNIT Frontend May be Subject to Elevation of Privilege Using Remote Code Execution | Not Started |
| 9 | Elevation Of Privilege | Elevation by Changing the Execution Flow in COGNIT Frontend | Not Started |
| 10 | Elevation Of Privilege | Elevation Using Impersonation | Not Started |

**Table 4.3:** Query a EC  Flow Threats

The status in the table is what we can do about the threats, we did not yet work on that aspect (see Next Steps and Observations). The status values are the following:

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

**Work done**

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 we collected about the new architecture of the COGNIT Framework, we created a threat model necessary to point to possible threats. We did not address the threats:

- Are they confirmed?
- How to mitigate or remove them?

We reduced the number of different threats to 13, because the same vulnerabilities are present for several components of the framework.

### Next Steps and Observations

Our next steps will include:

- consider the threats for all the flows, set the correct status, with a justification.
- schedule one or several meetings with the stakeholders so that they review, validate and give more input about the framework to consolidate the system description.
- schedule meetings with the people in charge of the components to review the identified threats.
- consider using other tools, such as threagile, to improve threat analysis.
- Illustrate how the threats are exploited through attack flows targeting specific assets of the cybersecurity use case such as the availability of the anomaly detection.

Meetings with everyone involved in developing the framework are important because everyone can provide input that is fundamental to identify vulnerabilities and threats early on, and decide how to address them.

The process is not a one time job. Each time new components are added, components are removed, or responsibility changes, it can impact the threats to the framework. Therefore, the threat model must be maintained throughout the development process. As an example, a recent change to the architecture, that the architecture team did communicate, reduced the number of possible threats from 165 to 134.

Our current work is the starting point of the process, so we have not worked more deeply in the analysis on the threats produced by the tool we use.

## 4.1 [SR6.1] Advanced Access Control

### Description

In terms of authorization, as commented in the SR1.5 section on this document, in the current development cycle the Biscuit token has been introduced, which enhances the authorization scheme of the previous versions. This allows doing  a more advanced access control within the COGNIT framework.

## 4.2 [SR6.2] Confidential Computing

### Description

In this development cycle we evaluated the available technologies for confidential computing to secure the processing of data in Serverless Runtimes within the COGNIT framework. Based on the application agonistic behaviour of Confidential Virtual Machines (CVM) and Open Source code availability, we decided to proceed with AMD Secure Encrypted Virtualization (AMD-SEV).