

# D3.2 COGNIT FaaS Model - Scientific Report - b

Version 1.0

30 April 2024

## Abstract

COGNIT is an AI-enabled Adaptive Serverless Framework for the Cognitive Cloud-Edge Continuum that enables the seamless, transparent, and trustworthy integration of data processing resources from 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 Second Research & Innovation Cycle (M10-M15), providing details on the status of a number of key components of the COGNIT Framework (i.e. Device Client, Serverless Runtime, and Provisioning Engine) as well as reporting the work related to supporting the Secure and Trusted Execution of Computing Environments.



Copyright © 2023 SovereignEdge.Cognit. All rights reserved.



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



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

## Deliverable Metadata

<b>Project Title:</b>	<a href="#">A Cognitive Serverless Framework for the Cloud-Edge Continuum</a>
<b>Project Acronym:</b>	SovereignEdge.Cognit
<b>Call:</b>	HORIZON-CL4-2022-DATA-01-02
<b>Grant Agreement:</b>	101092711
<b>WP number and Title:</b>	WP3. Distributed FaaS Model for Edge Application Development
<b>Nature:</b>	R: Report
<b>Dissemination Level:</b>	PU: Public
<b>Version:</b>	1.0
<b>Contractual Date of Delivery:</b>	31/03/2024
<b>Actual Date of Delivery:</b>	30/04/2024
<b>Lead Author:</b>	Idoia de la Iglesia (Ikerlan)
<b>Authors:</b>	Monowar Bhuyan (UMU), Malik Bouhou (CETIC), Aritz Brosa (Ikerlan), Sébastien Dupont (CETIC), Aitor Garciandia (Ikerlan), Torsten Hallmann (SUSE), Johan Kristiansson (RISE), Martxel Lasa (Ikerlan), Marco Mancini (OpenNebula), Alberto P. Martí (OpenNebula), Philippe Massonet (CETIC), Nikolaos Matskanis (CETIC), Daniel Olsson (RISE), , Goiuri Peralta (Ikerlan), Samuel Pérez (Ikerlan), Thomas Ohlson Timoudas (RISE), Paul Townend (UMU), Iván Valdés (Ikerlan), Constantino Vázquez (OpenNebula), Daniel Clavijo (OpenNebula), Jorge Lobo (OpenNebula), Michal Opala (OpenNebula).
<b>Status:</b>	Submitted

## Document History

Version	Issue Date	Status <sup>1</sup>	Content and changes
0.1	22/04/2024	Draft	Initial Draft
0.2	29/04/2024	Peer-Reviewed	Reviewed Draft
1.0	30/04/2024	Submitted	Final Version

## Peer Review History

Version	Peer Review Date	Reviewed By
0.1	29/04/2024	Per-Olov Östberg (UMU)
0.1	29/04/2024	Antonio Álvarez (OpenNebula)

## Summary of Changes from Previous Versions

First Version of Deliverable D3.2
-----------------------------------

<sup>1</sup> A deliverable can be in one of these stages: Draft, Peer-Reviewed, Submitted, and Approved.

## Executive Summary

This is the second “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 that have been active during the Second Research & Innovation Cycle (M10-M15) in connection with these main components of the COGNIT Framework:

### Device Client

- **SR1.1** Interface with Provisioning Engine:  
*Implementation of the communication with the Provisioning Engine.*
- **SR1.2** Interface with Serverless Runtime:  
*Implementation of the communication of with the Serverless Runtime*
- **SR1.3** Programming languages:  
*Support for different programming languages.*

### Serverless Runtime

- **SR2.1** Secure and Trusted FaaS Runtimes:  
*Automated building of secure and trusted images (vulnerability scans, security assessment) related to different flavours of FaaS Runtimes.*

### Provisioning Engine

- **SR3.1** Provisioning Interface for the Device to manage Serverless Runtimes:  
*Provide an interface to the Device asking for a Serverless Runtime to offload functions and data transfer on any resource of the cloud-edge continuum.*

### 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 Second Research & Innovation Cycle (M15), and will be updated with incremental releases at the end of each research and innovation cycle in M21, M27, and M33.

## Table of Contents

Abbreviations and Acronyms	5
1. Device Client	6
[SR1.1] Interface with Provisioning Engine	6
[SR1.2] Interface with Serverless Runtime	10
[SR1.3] Programming languages	11
2. Serverless Runtime	22
[SR2.1] Secure and Trusted FaaS Runtimes	22
3. Provisioning Engine	26
[SR3.1] Provisioning Interface for the Device to manage Serverless Runtimes	26
4. Secure and Trusted Execution of Computing Environments	33
[SR6.1] Advanced Access Control	33
[SR6.2] Confidential Computing	36

## Abbreviations and Acronyms

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

# 1.Device Client

## [SR1.1] Interface with Provisioning Engine

### Description

The Device Runtime is the component that enables devices to communicate with the COGNIT Platform to perform offloading of tasks. This component communicates with the Provisioning Engine to create/retrieve/update/delete Serverless Runtimes. It communicates with the provided FaaS Runtime to perform offloading of functions and uploading of content to the DaaS Runtime, if configured.

The Device Runtime will be delivered as a library with implementations in C and Python that abstracts uploading tasks from the internal application protocol by offering a user-friendly API.

The interface with the Provisioning Engine establishes communication with the COGNIT Framework, allowing the user's device to access its permitted resources.

### Architecture & Components

- ***Python version of Device Runtime***

The update details of the Python version can be seen in [CHANGELOG](#) file of the GitHub repository.

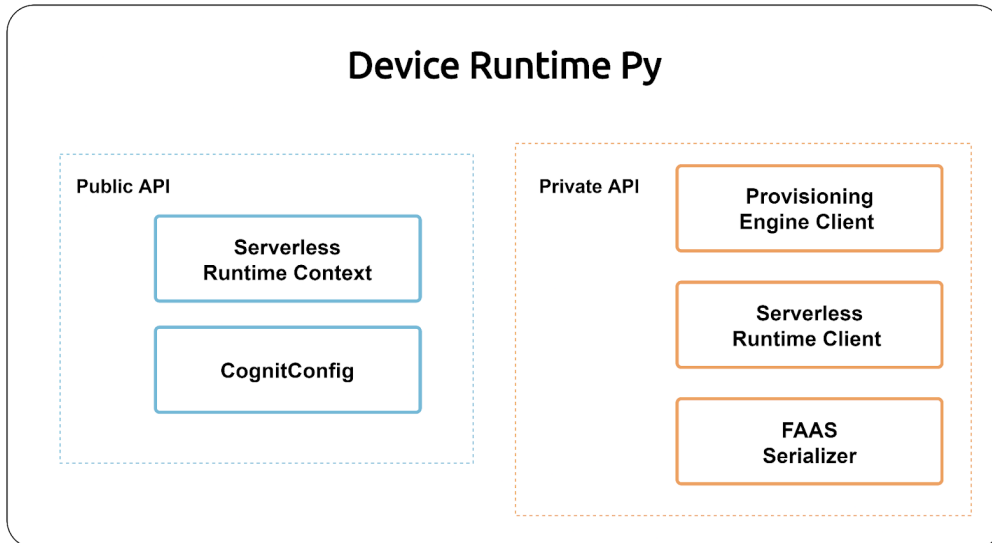
Mainly the Update method has been added, in order to allow the Device Client perform changes in the requirements of the User application, so the VM that serves the Serverless Runtime can be migrated within the COGNIT Testbed.

Another feature added was the default Geolocation of the device, based on tracing of its IP address. Also this Python SDK allows the user to specify any other Geolocation the user may want to add (from an external geolocation source, such as a GPS antenna) as shown in one of the articles of [Wiki documentation](#) of its GitHub repository. This information fills the DeviceInfo.GEOGRAPHIC\_LOCATION of **Table 1.1**.

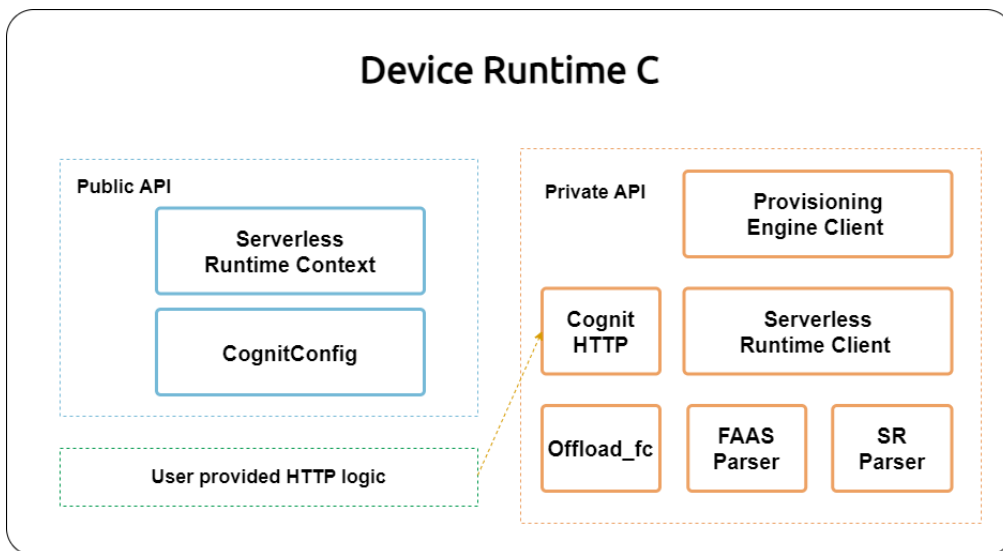
Moreover, the measured latency in the communication between the Device Client and the Provisioning Engine is being sent in the M15 release version, so the associated Serverless Runtime is aware of this metric. This information fills the DeviceInfo.LATENCY\_TO\_PE of **Table 1.1**.

Last but not least, the hash of the function to be offloaded is computed before sending the offloading request, so the function itself gets registered with a unique ID within the COGNIT infrastructure.

The first step to establish connection with the COGNIT Framework is to be able to communicate with the Provisioning Engine, which will specify the Serverless Runtime to be used by the Device Client, provided that the credentials of the device are validated as able to interact with the Framework.



**Figure 1.1.** Block diagram of Python version of Device Runtime's modules



**Figure 1.2.** Block diagram of C version of Device Runtime

- ***C version of Device Runtime***

Regarding the C version of the Device Runtime, as shown in **Figure 1.2**, it is very similar to the Python version, although there are some differences. Apart from the way of defining the function to be offloaded as can be seen in one of the examples provided in the GitHub [repository](#), where function parameters need to be defined as "IN" or "OUT", the main difference with the Python version is that the HTTP client logic is handed over to the user application, as its logic is directly linked to the Platform where the application needs to be running, which is usually a very constrained device, hence the client would need to be customised (generally speaking) within this device.

The other components of the private API, such as FaaS Parser implements all the needed logic for the process of serialisation of a given function that will be offloaded to the Serverless Runtime and to format specific JSONs to internal library structures for the correct execution of the function in C. Similar to SR Parser (shown in **Figure 1.2**) that serialises the Serverless Runtime configuration.

The Offload\_fc component generates the function to offload and the params in the string format, all in compliance with the format that the C executor is expecting in the Serverless Runtime.

## Data Model

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

The last attribute called Serverless Runtime is the type of object that encompasses the rest of the attributes of the following table:

Attribute	Description	Fields	Type
FaaSState	String describing the state of the Serverless Runtime.	PENDING = "PENDING" RUNNING = "RUNNING"	Enum
FaaSConfig	Object containing information about the requirements of the Serverless Runtime (CPU, MEM, ...)	CPU: int (optional) MEMORY: int (optional) DISK_SIZE: str (optional) FLAVOUR: str ENDPOINT: str (optional) STATE: FaaSState VM_ID: str (optional)	Inherited from pydantic's BaseModel
Scheduling	String describing the policy applied to scheduling. Eg: "energy, latency" will optimise the placement according to those two criteria.	POLICY: str REQUIREMENTS: str	Inherited from pydantic's BaseModel
DeviceInfo	Information related to the device where the Serverless Runtime is being hosted.	LATENCY_TO_PE: float GEOGRAPHIC_LOCATION: str	Inherited from pydantic's BaseModel
ServerlessRuntime	Definition of the Serverless Runtime to communicate to the PE.	NAME: str ID: int FAAS: FaaSConfig DAAS: DaaSConfig (optional) SCHEDULING: Scheduling (optional) DEVICE_INFO: DeviceInfo (optional)	Inherited from pydantic's BaseModel

**Table 1.1.** Data Model defining basic Serverless Runtime



## API & Interfaces

The novelty of this development cycle is the **Update** method, which as mentioned has been added to the Device Client's API (marked in bold).

Description	Method	Parameters	Return Type
Enables the developer to establish a Serverless Runtime context to be used in the application being run.	create	Valid CognitConfig object.	StatusCode.
Get the current Serverless Runtime status (Property)	status	-	FaaSState
<b>Updates the scheduling and device info fields of the associated Serverless Runtime</b>	<b>update</b>	<b>Update Scheduling and Device_info to associated Serverless Runtime.</b>	<b>StatusCode</b>
Delete the current Serverless Runtime context	delete	Delete the current Serverless Runtime context	Nothing

**Table 1.2.** API defining the Device Client's interaction with the Provisioning Engine.

## [SR1.2] Interface with Serverless Runtime

### Description

The novelty of this development cycle is the addition of the function hash.

### Data Model

The data structures defining the possible inputs and responses from/towards a given SR, from the Device Client's standpoint. The function hash addition is remarked in bold letters.

Attribute	Description	Fields	Type
status_exec	Execution status	OK WORKING NOT_OK	Enum
Param	Parameter definition.	type: str var_name: str value: str # Coded b64 mode: str	Inherits from pydantic BaseModel
ExecSyncParams	Synchronously executed function's details, comprising language of function, function itself, its parameters and function's hash.	lang: str fc: str <b>fc_hash: str</b> params: list[str]	Inherits from pydantic BaseModel
ExecAsyncParams	Asynchronously executed function's details, comprising language of function, function itself, its parameters and <b>function's hash</b> .	lang: str fc: str <b>fc_hash: str</b> params: list[str]	Inherits from pydantic BaseModel
FaaSUuidStatus	State and result (if any) of a given SR.	state: str result: str (Optional)	Inherits from pydantic BaseModel
... (other elements of the whole Data Model were skipped)			

**Table 1.3.** Data Model defining the Device Client's interaction with the Serverless Runtime.

### API & Interfaces

The API of this interface didn't have any modification from the previous development cycle.

## [SR1.3] Programming languages

### Description

In this version an extended Python version of the Device Client has been implemented (representing interpreted languages), and first version of the C version for more easily integrating COGNIT with constrained devices.

### Architecture & Components

Architecture and components are similar in both (Python and C) versions, although there are certain modifications due to language (C) intrinsic constraints.

#### *Python Specification*

Class	Description
CognitConfig	The global configuration to access the COGNIT Platform (Provisioning Engine IP and port, and needed credentials) will be stored in an instance of this class.
ServerlessRuntimeContext	Represents the Serverless Runtime context and provides runtime operations. This is a session with an assigned Serverless Runtime for offloading functions.
ServerlessRuntimeConfig	Represents the requirements for the Serverless Runtime.
ServerlessRuntimeStatus	Represents the status of the Serverless Runtime. Possible values: FAILED, READY, REQUESTED.
StatusCode	Represents the status code for an operation. Possible values: ERROR, SUCCESS.

**Table 1.4.** Classes associated with the SDK

Method	Description	Arguments	Return Type
<b>configure</b>	Enables the developer to configure the endpoint and credentials to connect to the COGNIT Platform instance. By default it will be obtained from env vars	Endpoint: The COGNIT Platform endpoint that will be used	None

**Table 1.5.** Methods linked to CognitConfig

The ServerlessRuntime Context provides the following functions to interact with the serverless runtime:

Method	Description	Arguments	Return Type
<b>create</b>	Enables the developer to establish a Serverless Runtime context to be used in the application being run.	ServerlessRuntimeConfig: specifying the initial configuration in terms of HW requirements, flavour, scheduling and device information.	StatusCode
<b>call_async</b>	Perform the offload of a function to the COGNIT Platform without blocking	func: Callable args: Union[List[Any], Tuple[Any, ...], Dict[str, Any]]	AsyncExecResponse
<b>call_sync</b>	Perform the offload of a function to the COGNIT Platform and wait for the result	func: Callable args: Union[List[Any], Tuple[Any, ...], Dict[str, Any]]	ExecResponse
<b>wait</b>	Wait for an asynchronous execution to finish (becoming the workflow of the function in synchronous). If the timeout is reached it will continue having asynchronous execution.	Id :AsyncExecId, timeout: seconds to wait for a response	AsyncExecResponse
<b>update</b>	Updates the scheduling and device info fields of the associated Serverless Runtime	ServerlessRuntimeConfig: Specifying the new configuration in terms of scheduling and device information.	StatusCode
<b>delete</b>	Delete the current ServerlessRuntime context	-	-
<b>status</b>	Get the current Serverless Runtime status (Property)	-	ServerlessRuntime

**Table 1.6.** List of methods of SDK.

### C Specification

Class	Description
cognit_config_t	A struct which holds the global configuration of the library which includes the config to access to the COGNIT Platform instance.
serverless_runtime_context_t	Represents the Serverless Runtime context and needs to be provided to the Serverless Runtime Context module to execute runtime operations.

<code>serverless_runtime_conf_t</code>	Represents the configuration for the Serverless Runtime (flavours, requirements...).
<code>serverless_runtime_cli_t</code>	Stores the endpoints of a possible task.
<code>e_faas_state_t</code>	Represents the status of the Serverless Runtime. Possible values: ERROR, PENDING, RUNNING, NO_STATE.
<code>e_status_code_t</code>	Represents the status code for an operation. Possible values: ERROR, SUCCESS, PENDING.
<code>exec_faas_params_t</code>	Represents the function to be offloaded with the params.

**Table 1.7.** Structures associated with the SDK

The ServerlessRuntime Context provides the following functions to interact with the serverless runtime:

Method	Description	Arguments	Return Type
<code>serverless_runtime_ctx_init</code>	Enables the developer to configure the endpoint and credentials to connect to the COGNIT Platform instance.	<code>Cognit_config_t</code>	<code>e_status_code_t</code>
<code>serverless_runtime_ctx_create</code>	Enables the developer to communicate to Prov Engine and create a runtime based on a Serverless Runtime specifications	<code>Serverless_runtime_ctx_t</code> , <code>Serverless_runtime_conf_t</code>	<code>e_status_code_t</code>
<code>serverless_runtime_ctx_status</code>	Enables the developer to get the current Serverless Runtime status	<code>Serverless_runtime_ctx_t</code>	<code>e_faas_state_t</code>

<b>serverless_runtime_ctx_call_sync</b>	Perform the offload of a function to the COGNIT Platform and wait for the result	Serverless_runtime_ctx_t Exec_faas_params_t Exec_response_t	e_status_code_t
<b>serverless_runtime_ctx_call_async</b>	Perform the offload of a function to the COGNIT Platform without blocking (with the integration of Phoenix systems)	Serverless_runtime_ctx_t Exec_faas_params_t Async_exec_response_t	e_status_code_t
<b>serverless_runtime_ctx_wait_for_task</b>	Wait for a task id to be executed	Serverless_runtime_ctx_t Async_task_id timeout_ms Async_exec_response_t	int
<b>serverless_runtime_delete</b>	Delete the current ServerlessRuntime context	Serverless_runtime_ctx_t	e_status_code_t

**Table 1.8.** List of methods of SDK.

## Data Model

The data models for the Python and the C version of the API are semantically equivalent..

## API & Interfaces

For consistency it needs to implement the same API endpoints with equally formatted bodies.

### **Python SDK usage example**

As specified in the [GitHub README](#) for the Device Client, there are several steps to be followed in order to build the Python module (named as “*cognit*”). Once done with the “Setting up COGNIT module” section, the user should be able to use it freely.

In this version (M15), the **Update** method is introduced in the Python version of the client, which allows the user to create a Serverless Runtime and define the initial placement of the associated VM based on the requirements provided by the user app’s request. With the **Update** method the user is able to change the requirements of the associated Serverless Runtime (VM), and it may trigger a rescheduling of the SR causing a migration of it, depending on the constraints of the underlying infrastructure.

Showcasing the way to use the Python module (which implements all the methods above mentioned in the SDK specification) is the `create_offl_update_offl` example under the `examples` subfolder in the repository, which creates a request for a Serverless Runtime to the corresponding Provisioning Engine with initial placement, and once it is ready it updates the requirements of it, which may trigger a rescheduling of the associated SR.

```
Python
# Configure the Serverless Runtime requirements
sr_conf = ServerlessRuntimeConfig()
sr_conf.name = "Example Serverless Runtime"
sr_conf.scheduling_policies = [EnergySchedulingPolicy(30)]
# This is where the user can define the FLAVOUR to be used within COGNIT to
# deploy the FaaS node.
sr_conf.faas_flavour = "<Flavour_to_be_used>"

# Request the creation of the Serverless Runtime to the COGNIT Provisioning
# Engine
try:
    # Set the COGNIT Serverless Runtime instance based on 'cognit.yml'
    # config file
    # (Provisioning Engine address and port...)
    my_cognit_runtime =
ServerlessRuntimeContext(config_path="./examples/cognit.yml")
    # Perform the request of generating and assigning a Serverless Runtime
    # to this Serverless Runtime context.
    ret = my_cognit_runtime.create(sr_conf)
except Exception as e:
    print("Error in config file content: {}".format(e))
    exit(1)

# Wait until the runtime is ready

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

print("COGNIT Serverless Runtime ready!")

# Update the Device info and requirements of the SR.
sr_conf = ServerlessRuntimeConfig()
sr_conf.name = "Updated Serverless Runtime"
sr_conf.scheduling_policies = [EnergySchedulingPolicy(80)]

result = my_cognit_runtime.call_sync(sum, 2, 2)
print("Pre-Update offloaded function result", result)

## Use this if you want to update any SR specifying the ID.
# This will update the SR of the context.
my_cognit_runtime.update(sr_conf)
```

```
# First, check status is != RUNNING, as there is some lag when reporting in
UPDATING state.
while my_cognit_runtime.status != FaaSState.UPDATING:
    print(f"VM state: {my_cognit_runtime.status}")
    time.sleep(1)
while my_cognit_runtime.status != FaaSState.RUNNING:
    print(f"VM state: {my_cognit_runtime.status}")
    time.sleep(3)

print("COGNIT Serverless Runtime ready after Updated!")

# Example offloading a function call to the Serverless Runtime

# call_sync sends to execute sync.ly to the already assigned Serverless
Runtime.
# First argument is the function, followed by the parameters to execute it.
result = my_cognit_runtime.call_sync(mult, 4, 5)

print("Post-Update offloaded function result", result)

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

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

### **C SDK usage example**

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

FUNC_TO_STR(
    mult_fc,
    void mult(int a, int b, float* c) {
        *c = a * b;
    })
```



```
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 (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
    {
        COGNIT_LOG_ERROR("[http_send_req_cb] curl_easy_setopt()
failed");
        return -1;
    }

    // Find '[' or ']' in the URL to determine the IP version
    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_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_ERROR("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();
free(headers);
```

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

int main(int argc, char const* argv[])
{
    cognit_config_t t_my_cognit_config;
    serverless_runtime_context_t t_my_serverless_runtime_context;
    serverless_runtime_conf_t t_my_serverless_runtime_conf;
    exec_response_t t_exec_response;
    exec_faas_params_t exec_params = { 0 };

    // Initialize the config for the serverless runtime context instance
    t_my_cognit_config.prov_engine_endpoint = "";
    t_my_cognit_config.prov_engine_pe_usr = "";
    t_my_cognit_config.prov_engine_pe_pwd = "";
    t_my_cognit_config.prov_engine_port = 0;
    t_my_cognit_config.ui32_serv_runtime_port = 0;

    serverless_runtime_ctx_init(&t_my_serverless_runtime_context,
&t_my_cognit_config);

    // Configure the initial serverless runtime requirements
    t_my_serverless_runtime_conf.name
= "my_serverless_runtime";
    t_my_serverless_runtime_conf.faas_flavour
= "DC_C_version_tests";

    t_my_serverless_runtime_conf.m_t_energy_scheduling_policies.ui32_energy_perc
centage = 50;

    if (serverless_runtime_ctx_create(&t_my_serverless_runtime_context,
&t_my_serverless_runtime_conf) != E_ST_CODE_SUCCESS)
    {
        printf("Error configuring serverless runtime\n");
        return -1;
    }

    // Check the serverless runtime status

    while (true)
    {
        if (serverless_runtime_ctx_status(&t_my_serverless_runtime_context)
== E_FAAS_STATE_RUNNING)
        {
            printf("Serverless runtime is ready\n");
            break;
        }

        printf("Serverless runtime is not ready\n");

        sleep(1);
    }
}
```

```
    }

    // Offload the function execution to the serverless runtime
    // This will use the callback function my_http_send_req to send the
    request

    const char* includes = INCLUDE_HEADERS(#include<stdio.h> \n);
    offload_fc_c_create(&exec_params, includes, mult_fc_str);
    // Param 1
    offload_fc_c_add_param(&exec_params, "a", "IN");
    offload_fc_c_set_param(&exec_params, "int", "3");
    // Param 2
    offload_fc_c_add_param(&exec_params, "b", "IN");
    offload_fc_c_set_param(&exec_params, "int", "4");
    // Param 3
    offload_fc_c_add_param(&exec_params, "c", "OUT");
    offload_fc_c_set_param(&exec_params, "float", NULL);

    serverless_runtime_ctx_call_sync(&t_my_serverless_runtime_context,
    &exec_params, &t_exec_response);

    COGNIT_LOG_INFO("Result: %s", t_exec_response.res_payload);

    // Free the resources
    faasparser_destroy_exec_response(&t_exec_response);
    offload_fc_c_destroy(&exec_params);

    COGNIT_LOG_INFO("Deleting serverless runtime");

    while
    (prov_engine_delete_runtime(&t_my_serverless_runtime_context.m_t_prov_engine
    _cli, t_my_serverless_runtime_context.m_t_serverless_runtime.ui32_id,
    &t_my_serverless_runtime_context.m_t_serverless_runtime) != 0)
    {
        COGNIT_LOG_ERROR("Error deleting serverless runtime");
        sleep(1);
    }

    COGNIT_LOG_INFO("Serverless runtime deleted");

    return 0;
}
```

## 2. Serverless Runtime

### [SR2.1] Secure and Trusted FaaS Runtimes

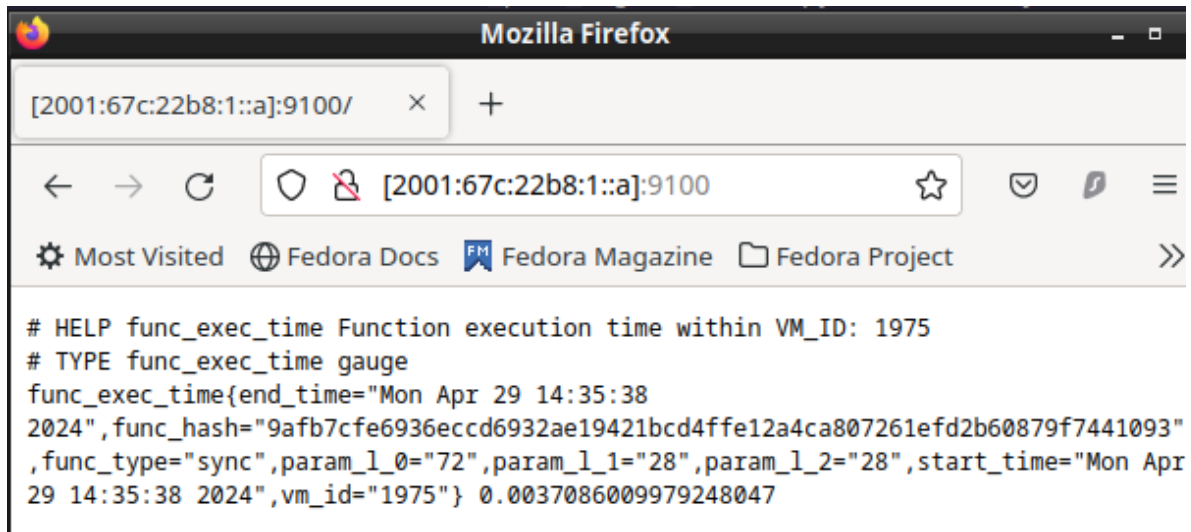
#### Description

The FaaS component of the Serverless Runtime is the environment in which functions to be offloaded are executed within the COGNIT Framework by the machine that provides the Provisioning Engine, as already described in the previous cycle's Scientific Report.

Various runtime configurations are available for deployment to meet the specific requirements of each function. These runtimes communicate with the Device Client, offloading the designated function through a RESTful API. The system supports the execution of functions written in both Python and C languages.

However, the runtime's image must contain all the software requirements for the execution of the function. Regarding the management of the images needed by the different use cases, an effort has been put in place which so far has not been linked to any automated solution. In order to address that, there were identified many weak points on the current version of COGNIT that need to be improved throughout the next development cycles, although some actions were taken already in the end of the current one. Next steps related to these issues are mentioned in the D2.3 (section 4. Priorities for Third Research & Innovation Cycle (M16-M21), related to [Open Build Service \(OBS\)](#)) of the current COGNIT release.

Moreover, the SR needs to expose some information about function execution, allowing the AI orchestrator to get key information (Prometheus metric) about them, so the orchestration can be more data driven. The example shown in **Figure 2.1** is representative of a function execution offloaded by an user app through the Device Client in a given Serverless Runtime that was created in the Testbed infrastructure used in the project. The IPv6 address (that ends in '::14') shown in the browser's address bar is the IP of the Serverless Runtime that was assigned to that particular Device Client, and the 9100 value followed by a colon is the port in which the SR exporter is exposing those metrics. This figure depicts the Prometheus metrics structure and the meaning of each field exposed by the SR exporter, as shown in **Figure 2.2**.



```

# HELP func_exec_time Function execution time within VM_ID: 1975
# TYPE func_exec_time gauge
func_exec_time{end_time="Mon Apr 29 14:35:38
2024",func_hash="9afb7cfe6936eccd6932ae19421bcd4ffe12a4ca807261efd2b60879f7441093"
,func_type="sync",param_l_0="72",param_l_1="28",param_l_2="28",start_time="Mon Apr
29 14:35:38 2024",vm_id="1975"} 0.0037086009979248047

```

**Figure 2.1.** Prometheus metric example.

Describing the content of **Figure 2.1**, the 'func\_exec\_time' is the metric name, and the value is a Prometheus Gauge data type, showing a float-like value that represents the function execution time in seconds. The metric labels provide additional information about it, 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.
- **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 SR within the COGNIT infrastructure the function was executed.

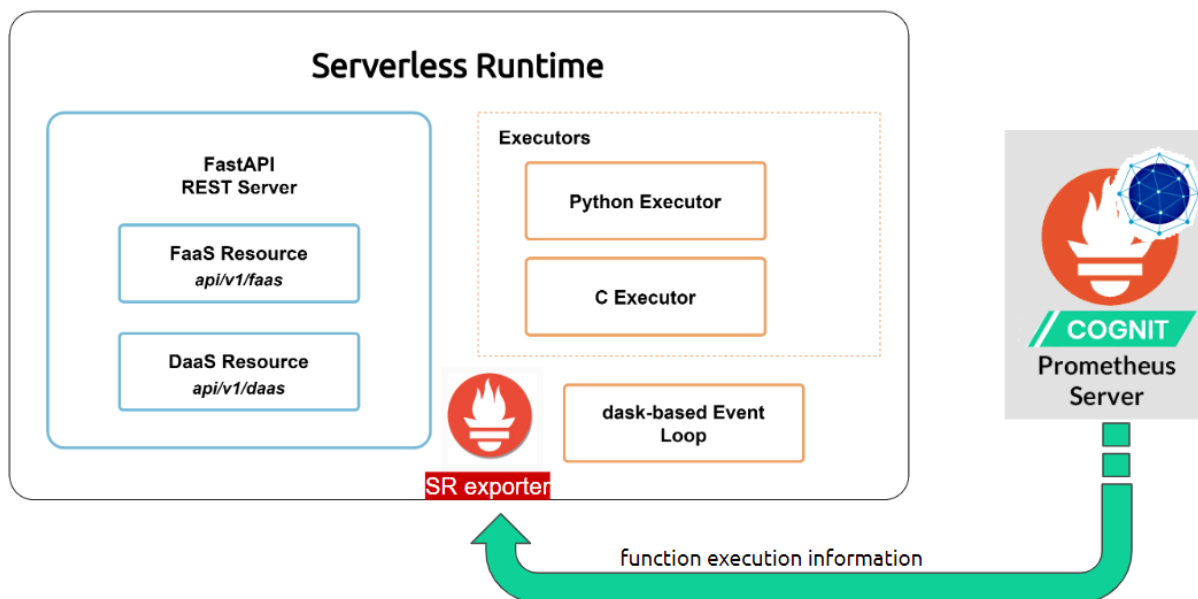
## Architecture & Components

The Serverless Runtime provides a public FastAPI <sup>2</sup>REST Server that listens to FaaS requests. As illustrated in Figure 2.2, 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.

<sup>2</sup> <https://fastapi.tiangolo.com>

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. FaaS Manager: Responsible for adding an asynchronous task offloading and managing its execution status.
5. Dask-based Event Loop: Provides parallel task execution and scalability of data processing applications.
6. C Executor: Groups all the logic needed to execute C language with <sup>3</sup>Cling, and interactive C interpreter.
7. Py Executor: Groups the logic needed to execute Python language.



**Figure 2.2.** Block Diagram of Serverless runtime modules.

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 *fc\_hash* field to enable the reception of the hash of the function that had been offloaded (highlighted in bold):

```
Unset
{
  "lang": "string",
```

<sup>3</sup> <https://github.com/root-project/cling>  
<https://root.cern/cling/>



```

"fc": "string",
"fc_hash": "string",
"params": ["string", "string", "string"]
}

```

**Table 2.1** shows the addition from the D3.1 document.

Attribute	Description	Value
lang	String describing the programming language of the code to be offloaded	Base64 string.
fc	String describing the function to be offloaded coded in base64.	Base64 string.
fc_hash	<b>Defines the hash of the function, which could be used as its identifier.</b>	<b>String, with HEX characters.</b>
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, READY, FAILED.

**Table 2.1.** 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.

### 3. Provisioning Engine

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

##### Description

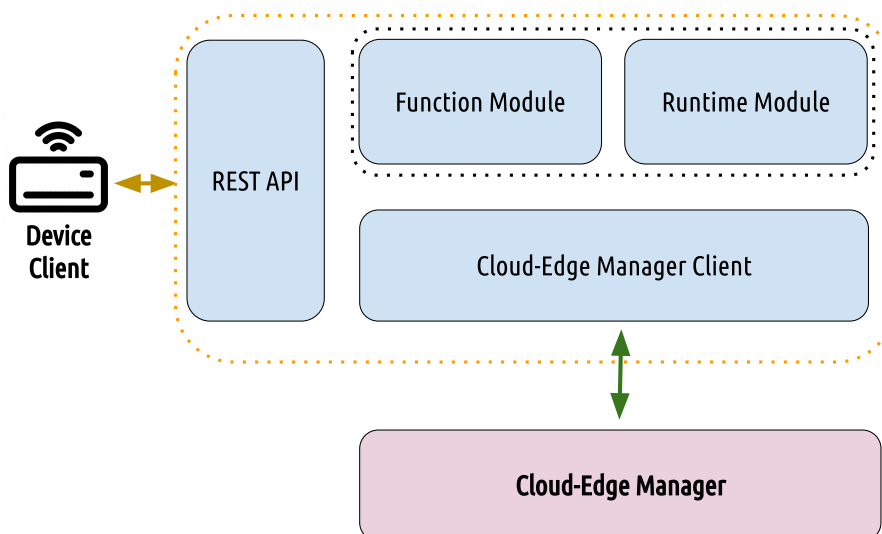
The Provisioning Engine is a software component that acts as a single point of contact for Application Device Runtimes that request access to Serverless Runtimes, are comprised of a FaaS Runtime to offload computation through the FaaS paradigm, and/or a DaaS Runtime to offload data into the cloud. Once this component receives a request for a Serverless Runtime it communicates with the Cloud-Edge Manager, waits for the Serverless Runtime to be available and returns the endpoints for the Device Runtime to communicate with.

In this second development cycle the following changes were made to this component:

- Slight change in the architecture, the Translation Module was split into two modules.
- A new ERROR attribute was added to the Serverless Runtime JSON description document.
- The Update operation over a Serverless Runtime has been implemented.
- The communication with the device client has been securitized.

##### Architecture & Components

The Provisioning Engine architecture introduced in D3.1 has been slightly modified to decompose the Translator module into two different modules as seen in Figure 3.1, Function Module and Runtime Module (old Translator module represented by a black dashed rectangle, whereas the whole Provisioning Engine is contained in the orange dashed rectangle), as this better captures two different functions of the Translator Module and helps to keep the implementation clean.



**Figure 3.1.** Provisioning Engine High Level Architecture

- **Function Module:** Translates Serverless Runtime FLAVOUR requirements into Cloud-Edge Manager Compute instances operations.
- **Runtime Module:** Translates Serverless Runtime high level requirements into Cloud-Edge Manager Multi-Compute instances operations. Handles the Serverless Runtime entry persistence in the Cloud-Edge Manager.

As mentioned in D3.2 the Provisioning Engine 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 (provisioning-engine.conf). After testing and bug fixing in this second development cycle, new options have been added to allow timeout fine tuning and better debug info for troubleshooting. This can be checked in Table 3.1.

Attribute	Value
host	IP to which the Provisioning Engine will bind to listen for incoming requests.
port	Port to which the Provisioning Engine will bind to listen for incoming requests. Defaults to 2719.
onflow_server	OpenNebula OneFlow contact information
timeout	Seconds to wait for backend (OpenNebula and OneFlow) responses
capacity	Default capacity values (disk, cpu, memory) for the SR Virtual Machines to be used if not specified in the SR creation call
log	Sets the log debug level

**Table 3.1.** Provisioning Engine Server Configuration File

## Data Model

The only addition to the existing JSON description of the Serverless Runtime, as handled by the Provision Engine, is the addition of an Error attribute (see Table 3.2) to better describe the state of the Serverless Runtime in case of an error.

```
Unset
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "SERVERLESS_RUNTIME": {
      "type": "object",
      "properties": {
        "NAME": {
          "type": "string"
        },
        [...]
      },
    }
  }
}
```

```

    "FAAS": {
      "type": "object",
      "properties": {
[...]
```

```

        "ERROR": {
          "type": "string"
        }
      }
    },
[...]
```

Attribute	Value
ERROR	The error message of the VM containing the Serverless Runtime. This aggregates the error coming from different components of the Cloud Edge Manager, namely OpenNebula's OneFlow and core component (oned)

**Table 3.2.** New SR attribute in the second development cycle

## API & Interfaces

The Update operation shown in Table 3.3 has been implemented in this cycle, through which an existing Serverless Runtime can be modified. This translates into a change of the characteristics of the Virtual Machines containing the Serverless Runtime service in the Cloud-Edge Manager.

Action	Verb	Endpoint	Request Body	Response
Update Serverless Runtime	PUT	/serverless-runtimes/{id}	JSON representation of the updated serverless-runtime object	Status code 200 (OK) with the updated serverless-runtimeobject

**Table 3.3.** Operation implemented in the second development cycle

The Serverless Runtime properties that can be updated are described in Table 3.4, along with the impact on the underlying Cloud-Edge manager resource (Virtual Machine or OneFlow Service) that represents the Serverless Runtime.

Attribute	Impact
DAAS/FAAS CPU	Change the physical CPU allocated to this Serverless Runtime DaaS or FaaS component
DAAS/FAAS Memory	Change the memory allocated to this Serverless Runtime DaaS or FaaS component
DAAS/FAAS Disk Size	Change the main disk size allocated to this Serverless Runtime DaaS or FaaS component
NAME	Change the name of the Serverless Runtime
SCHEDULING	Change the scheduling requirements and policy associated to the Serverless Runtime
DEVICE_INFO	Change the device details associated with the Serverless Runtime: Geolocation and Latency to Provisioning Engine.

**Table 3.4.** Serverless Runtime attributes that can be updated

The new REST API endpoint update operation receives a schema compatible Serverless Runtime definition representing the desired state of the Serverless Runtime. The current Serverless Runtime state definition is requested from the Cloud-Edge Manager (since the Provisioning Engine is a pure stateless component). Both definitions, the current and the desired one in the update operation, are then compared to scan differences in the supported attributes, defined in Table 3.4, and changes are issued accordingly. These changes are of different nature, depending on the attribute that is actually modified:

- If the name is different, the document that backs the Serverless Runtime will be renamed.
- If the CPU is different, a live CPU hotplug operation will be issued to the Virtual Machine containing the Serverless Runtime.
- If the SCHEDULING and DEVICE\_INFO are different, the Serverless Runtime document body will be updated and the VMs definitions backing each of the Serverless Runtime services will have their instance templates updated as well. This way, the scheduler acts upon those properties.



RUNNING	Virtual Machine backing the Serverless Runtime is currently running
UPDATING	Virtual Machine backing the Serverless Runtime is currently being updated, for example, a hotplug due to the update operation
ERROR	Maps to any possible failure state on the Virtual Machine backing the Serverless Runtime.

**Table 3.5.** Serverless Runtime state meaning

The update operation also contains a fail safe recovery functionality. If an update operation is issued to a Serverless Runtime in error state (at least one of the Serverless Runtime Virtual Machines in an erroneous state), an attempt to recover the problematic Serverless Runtime will be made instead of updating the associated Virtual Machine capacity. Ideally this would result in an error recovery, putting the Serverless Runtime Virtual Machine on RUNNING. Then the update operation would run as normal and the Virtual Machine would go from RUNNING to UPDATING to then RUNNING again. More details about the update call in the [API documentation](#).

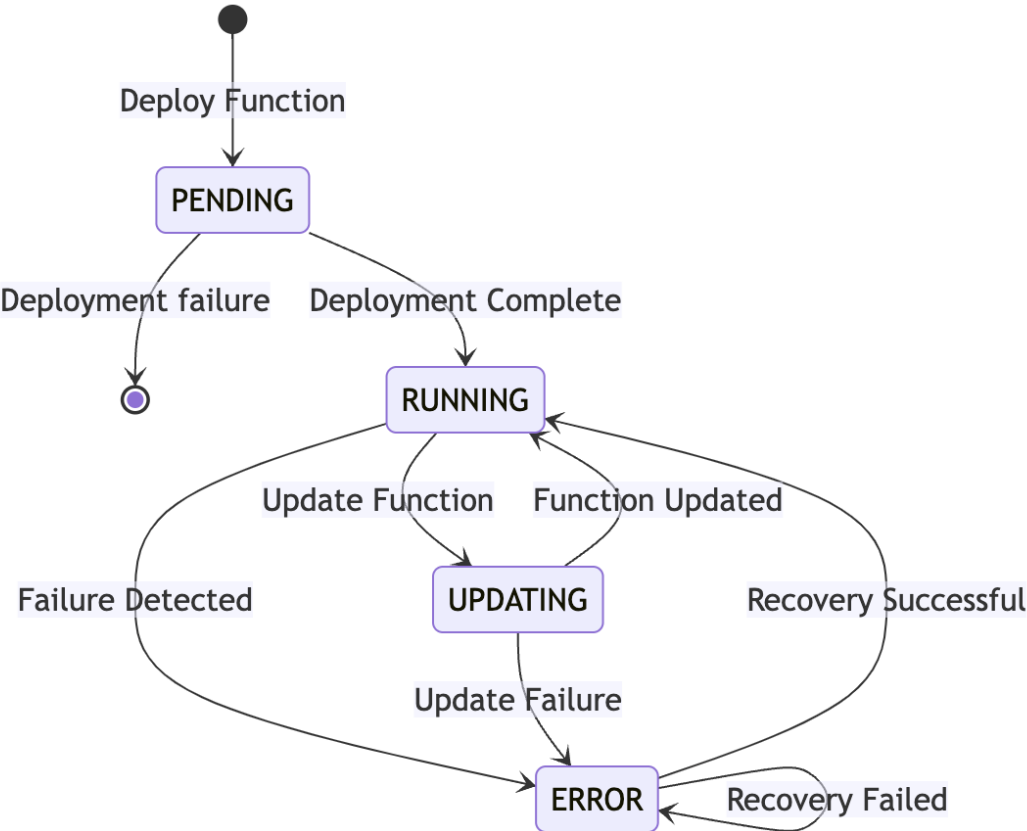


Figure 3.3. State Machine for Serverless Runtimes



## 4. Secure and Trusted Execution of Computing Environments

### [SR6.1] Advanced Access Control

During this cycle, we identified the requirements of an Identity and Access Management (IAM) mechanism tailored to the needs of the COGNIT Framework and the specific challenges of Edge Computing. 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. Based on these requirements, an analysis of features and technologies was conducted to determine those best suited to the COGNIT Framework environment and the constraints of Edge Computing.

We summarised 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 also identified certain features and technologies that meet the requirements of an IAM mechanism adapted to the COGNIT Framework:

1. **Single-Sign On and Single-Sign Out:** Allows users to sign in once to access multiple applications, thus facilitating identity management in a distributed environment like COGNIT. Additionally, single-sign-out functionality ensures that users are automatically logged out from all connected applications when they sign out from one.

2. **OpenID Connect:** Provides secure and standardised authentication, promoting interoperability and compatibility with other systems and services that COGNIT may communicate with.
3. **OAuth 2.0:** Enables granular and secure authorization of applications, users, and devices to control access to COGNIT resources, an environment where security and privacy are paramount.
4. **SAML:** Facilitates integration with existing authentication systems, simplifying device identity management in a heterogeneous environment like COGNIT.
5. **User Federation - Sync users from LDAP and Active Directory servers:** Enables synchronisation of users from LDAP and Active Directory servers, offering centralised identity management and simplifying data synchronisation.
6. **Admin Console for central management of devices, users, roles, role mappings, clients, and configuration:** Provides a centralised interface for managing devices, users, roles, clients, and configuration, thus facilitating administration and supervision.
7. **Two-factor Authentication - Support for TOTP/HOTP:** Enhances security by offering two-factor authentication, crucial for protecting sensitive data and resources in COGNIT.
8. **Session management - Admins can view and manage user sessions:** Allows administrators to view and manage user sessions, providing increased control and visibility of devices.
9. **Token mappers - Map devices and user attributes, roles, etc.:** Allows mapping of user attributes and roles into authentication tokens.
10. **Not-before revocation policies per realm, application, and user:** Offers precise revocation policies to control access to resources based on time, thus improving security and access management in COGNIT.
11. **CORS support - Client adapters have built-in support for CORS:** Supports resource sharing between different origins, essential for enabling integration with certain components of COGNIT.
12. **Service Provider Interfaces (SPI) - A number of SPIs to enable customising various aspects of the server:** Provides extensible interfaces to customise different aspects of the server, allowing adaptation of the IAM solution to the specific needs of COGNIT.
13. **Client adapters for Python and C:** Offers client adapters for Python and C, enabling integration with COGNIT.

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

1. **Device Client to Provisioning Engine:**

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. **Provisioning Engine to Cloud-Edge Manager:**

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.

By analysing the available features and technologies, we were able to establish a list of potential solutions that meet these specific requirements. However, to select the optimal solution for integration into the COGNIT Framework, our next step will be to deepen our research through an in-depth state-of-the-art analysis. This approach will allow us to examine in detail the various IAM solutions available, evaluate their relevance and suitability to our usage context, and finally define the optimal solution for integration into the COGNIT Framework. This process will ensure the implementation of a robust and efficient IAM mechanism, thereby contributing to the success and security of the COGNIT project as a whole.

## [SR6.2] Confidential Computing

In the last cycle, our team explored the possibilities offered by Confidential Computing technologies in the Edge Computing environment. We identified several promising avenues to enhance data security and confidentiality within the COGNIT Framework, particularly concerning OpenNebula as the baseline technology for the Cloud-Edge Manager component.

We examined the benefits of Trusted Execution Environments (TEE) for creating secure execution environments, as well as the use of homomorphic encryption for securely processing sensitive data. Additionally, we investigated the compatibility and integration of these technologies with an Identity and Access Management (IAM) mechanism. We found it necessary to ensure that Confidential Computing features could be seamlessly integrated with the IAM solution to enhance security and confidentiality throughout the data lifecycle.

We are analysing how Confidential Computing could be used to meet privacy requirements related to GDPR compliance. Confidential Computing can be used to meet privacy requirements and adhere to GDPR regulations. Confidential Computing can be used to process private data at the data controller's premises but also in external edge and cloud Platforms while providing high privacy guarantees. It guarantees privacy throughout processing by keeping data encrypted at rest (when stored) and also in use (while being processed). This is achieved through Trusted Execution Environments (TEEs) that are hardware-based secure enclaves in the processor. Data is only decrypted inside the hardware enclave to be processed. This prevents unauthorised access or modification of the data, even if the underlying system is compromised. Confidential Computing can provide cryptographic proof verifying the integrity of the processing environment. Authorised parties can access the results outside the TEE with the decryption key if needed.

Confidential Computing can be used to show private data processing is compliant with GDPR. GDPR emphasises data minimization by collecting and processing only the minimal data necessary for a specific purpose. Confidential Computing can work with minimal anonymized or privacy-preserving data sets. Data collected under GDPR must be used only for specific purposes (purpose limitation). Confidential Computing limits opportunities for unauthorised usage beyond its intended purpose. GDPR requires that appropriate measures are taken to protect personal data. Confidential Computing provides extra security by processing the decrypted data only in the hardware TEE enclave.

For the upcoming development cycles, we plan to conduct an in-depth state-of-the-art analysis of Confidential Computing technologies to integrate aspects of these solutions into OpenNebula and the COGNIT Framework. This approach will allow us to fully leverage the potential of confidentiality within the infrastructure, thereby ensuring an optimal level of security for applications deployed using the COGNIT Framework.