A Cognitive Serverless Framework for the Cloud-Edge Continuum

D3.1 COGNIT FaaS Model - Scientific Report - a

Version 1.0

31 October 2023

Abstract

COGNIT is an AI-enabled Adaptive Serverless Framework for the Cognitive Cloud-Edge Continuum that enables the seamless, transparent, and trustworthy integration of data processing resources from providers and on-premises data centers in the cloud-edge continuum, and their automatic and intelligent adaptation to optimise where and how data is processed according to application requirements, changes in application demands and behaviour, and the operation of the infrastructure in terms of the main environmental sustainability metrics. This document describes the research and development carried out in WP3 "Distributed FaaS Model for Edge Application Development" during the First Research & Innovation Cycle (M4-M9), 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.

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:	30/09/2023
Actual Date of Delivery:	31/10/2023
Lead Author:	Idoia de la Iglesia (Ikerlan)
Authors:	Monowar Bhuyan (UMU), Malik Bouhou (CETIC), Aritz Brosa (Ikerlan), Sébastien Dupont (CETIC), 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), Michał Opala (OpenNebula), Goiuri Peralta (Ikerlan), Samuel Pérez (Ikerlan), Thomas Ohlson Timoudas (RISE), Paul Townend (UMU), Iván Valdés (Ikerlan), Constantino Vázquez (OpenNebula).
Status:	Submitted

Document History

Version	Issue Date	Status[1]	Content and changes
0.1	20/10/2023	Draft	Initial Draft
0.2	27/10/2023	Peer-Reviewed	Reviewed Draft
1.0	31/10/2023	Submitted	Final Version

Peer Review History

Version	Peer Review Date	Reviewed By
0.1	27/10/2023	Marco Mancini (OpenNebula)
0.1	27/10/2023	Paul Townend (UMU)

Summary of Changes from Previous Versions

First Version of Deliverable D3.1

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

Executive Summary

This is the first version of Deliverable D3.1, the COGNIT FaaS Model Scientific Report, 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 First Research & Innovation Cycle (M4-M9) 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 First Research & Innovation Cycle (M9), and will be updated with incremental releases at the end of each research and innovation cycle (i.e. M15, M21, M27, M33).

Table of Contents

Abbreviations and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CC	Confidential Computing
CD	Continuous Delivery
DaaS	Data as a Service
DB	Database
FaaS	Function as a Service
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management system
IOPS	I/O Operations Per Second
IP	Internet Protocol
IoT	Internet of Things
JSON	Javascript Object Notation
LDAP	Lightweight Directory Access Protocol
ML	Machine Learning
NIS	Network and Information Security
OIDC	OpenID Connect
OS	Operating System
QoS	Quality of Service
REST	Representational State Transfer
RBAC	Role-Based Access Control
S3	Simple Storage Service
SDK	Software Development Kit
SEV	Secure Encrypted Virtualization
SGX	Software Guard eXtension
SLA	Service Level Agreement
SQL	Structured Query Language
TEE	Trusted Execution Environments
TLS	Transport Layer Security
VM	Virtual Machine
YAML	Yaml Ain’t a markup language

1. Device Client

[SR1.1] Interface with Provisioning Engine

Description

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

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

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

Architecture & Components

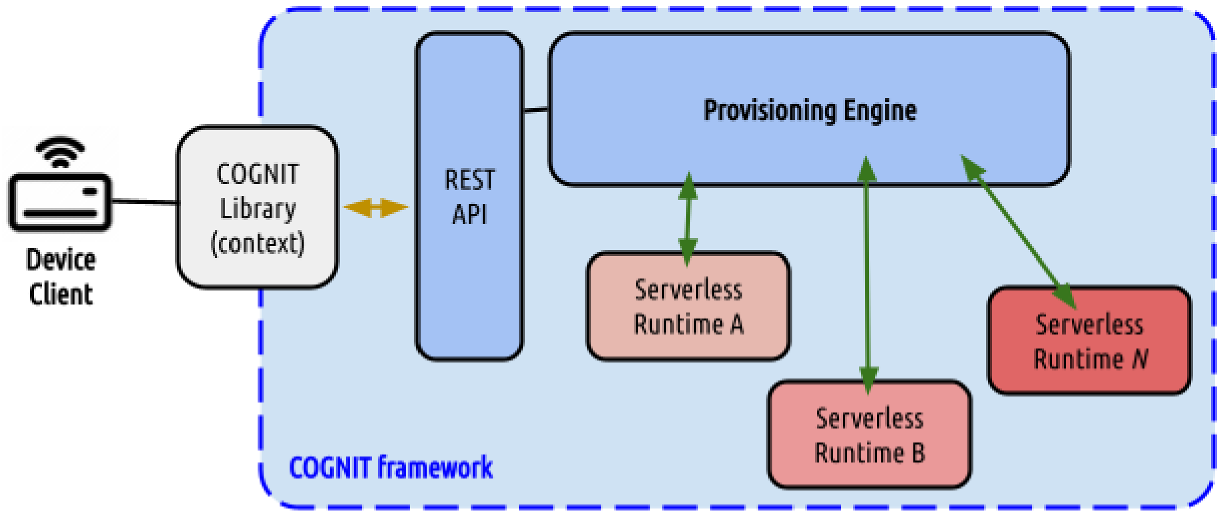

Figure 1.1. Schema of interaction of the Device Client with the Provisioning Engine.

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

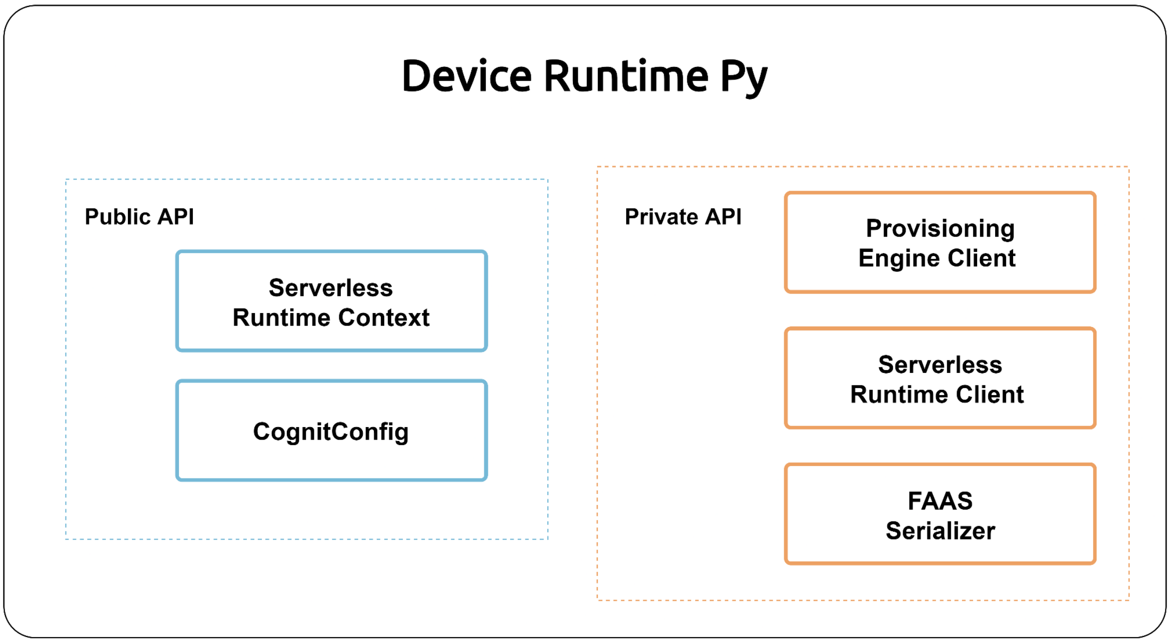

Figure 1.2. Block diagram of Device Runtime's modules

The COGNIT library is split into several parts. On one side there is the public API of the library, where the configuration for the communication with the COGNIT Framework (Provisioning Engine in the first step) can be defined, and the Serverless Runtime context (a valid session within COGNIT, having an assigned Serverless Runtime for function offloading), that exposes the actions concerned from the user's standpoint; such as: call_sync (send offload task request), call_async (send offload task asynchronously request), wait (wait until the async function execution in finished).

The private API defines three components: the Provisioning Engine client that implements the API client to request and manage Serverless Runtime instances.
The Serverless Runtime Client, implements the API client to manage offloaded tasks that are linked to a specific Serverless Runtime.

The FaaS serializer implements all the needed logic for the process of serialisation (format correctly) of a given function that will be offloaded by the Serverless Runtime Client.

The public API is available to the user, and makes use of functionalities given by the private API parts which are abstracted from the user for convenience, in order to provide all the functionalities needed by a device using the COGNIT Framework.

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: int 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 three methods, reported in the following table, allow the Device Client to request the creation of a COGNIT context (the session with an associated Serverless Runtime within the COGNIT infrastructure), to query the current status of a Serverless Runtime, or to delete an existing Serverless Runtime context.

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
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

Once the first communication with the Provisioning Engine is finished (and established connection to the COGNIT Framework), the interface with the Serverless Engine allows the user to interact with the Serverless Runtime to which it has been assigned. Through the defined API, the Device Client is able to manage offloaded tasks at the convenience of the application that is being run in the device.

Architecture & Components

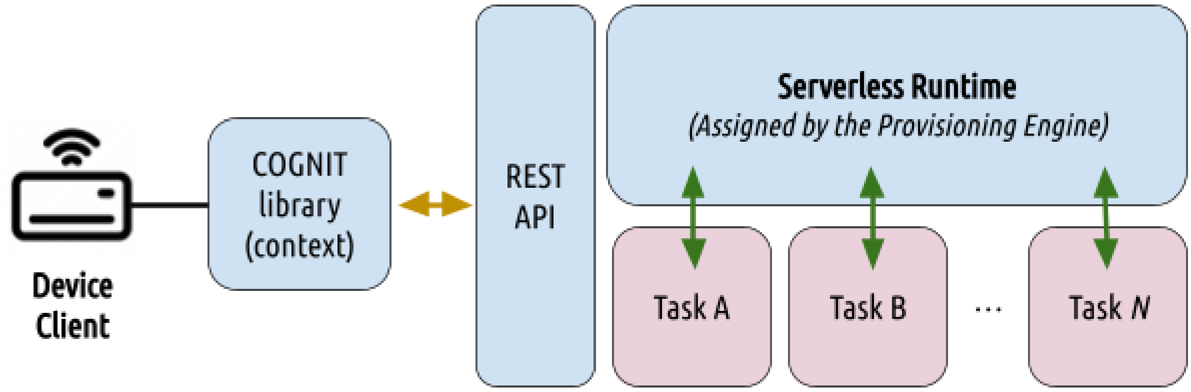

Figure 1.3. Block diagram of interaction between the Device Client and the Serverless Runtime.

The device is able to request execution of functions to be executed either synchronously or asynchronously.

Data Model

The data structures defining the possible inputs and responses from and towards a given SR, from the Device Client's standpoint.

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	lang: str fc: str params: list[str]	Inherits from pydantic BaseModel

	itself and its parameters.		
ExecAsyncParams	Asynchronously executed function's details, comprising language of function, function itself and its parameters.	lang: str fc: str 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
ExecReturnCode	Whether execution was successful or erroneous.	SUCCESS ERROR	Inherits from pydantic BaseModel
ExecResponse	Response of a generic execution, with it's return code, result and error if applicable.	ret_code: ExecReturnCode res: str (Optional) err: str (Optional)	Inherits from pydantic BaseModel
AsyncExecId	Id of the FaaS where the function is executed.	faas_task_uuid: str	Inherits from pydantic BaseModel
AsyncExecStatus	Whether a asynchronously executed function is still in process or already finished (either successfully or not)	WORKING READY	Enum
AsyncExecResponse	Defines Asynchronous execution status, response (if any) and the associated FaaS where is being executed.	status: AsyncExecStatus res: ExecResponse (Optional) exec_id: AsyncExecId	Inherits from pydantic BaseModel

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

API & Interfaces

There are two ways of offloading a function from the Device Client's standpoint:

Call_async allows the Device Client to execute it synchronously, passing the function object as first parameter and a set of positional arguments following it, which will act as the function's arguments. Its Response will be an ExecResponse, which shows the return code, result (if any), and error (if any), as shown in Table 1.3.

Call_async allows the Device Client executing a function asynchronously, with the same structure as call_sync, but instead this call's response will be an AsyncExecResponse, which includes an ExecResponse as execution response, status which defines AsyncExecStatus and exec_id which specifies the AsyncExecId, as shown in the table below:

Description	Method	Parameters	Return Type
Perform the offload of a function to the COGNIT platform and wait for the result	call_sync	func: Callable args: Any [Bundled as positional arguments]	ExecResponse.
Perform the offload of a function to the COGNIT platform without blocking	call_async	func: Callable args: Any [Bundled as positional arguments]	AsyncExecResponse.
Wait for an asynchronously executed function to finish to get its result (if applicable)	wait	Id: AsyncExecId, timeout: seconds to wait for a response	AsyncExecResponse.

Table 1.4. API that defines the Device Client functions to perform actions within an assigned Serverless Runtime.

Wait allows the Device Client waiting blocking the main program to finish (or timeout) a previously call_sync-ed function.

[SR1.3] Programming languages

Description

In this version only the Python version of the Device Client has been implemented (representing interpreted languages), which will be extended with a C version for more easily integrating COGNIT with constrained devices, in the M15 checkpoint of the project.

Architecture & Components

Architecture and components will be similar to the current (Python) version, although it may suffer from small modifications due to language (C) constraints.

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.
ServerlessRuntimeRequirements	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.

Method	Description	Arguments	Return Type
configure	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

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

Method	Description	Arguments	Return Type
call_async	Perform the offload of a function to the COGNIT platform without blocking	func: Callable	AsyncExecResponse

		args: Union[List[Any], Tuple[Any, ...], Dict[str, Any]]	
call_sync	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
wait	Wait for an	Id :AsyncExecId, timeout: seconds to wait for a response	AsyncExecResponse
copy	Copies src into dst	src: A string containing a local or remote path of a file to be uploaded to the Data Service, dst: Target path of the Serverless Runtime where the file will be copied	StatusCode
delete	Delete the current ServerlessRuntime context	-	-
status	Get the current Serverless Runtime status (Property)	-	ServerlessRuntime

Data Model

It will be similar to the current (Python) version, unless there are minimal tweaks required by the language to be used (C in this case).

API & Interfaces

For consistency it will need 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.

Showcasing the way to use the Python module (which implements all the methods above mentioned in the SDK specification) is the minimal_offload_sync example under the *examples* subfolder in the repository, which creates a request for a Serverless Runtime to the corresponding Provisioning Engine (specified in the COGNIT config file, named *"cognit.yml"*), checks its status, and once it is ready it requests the offload of a mock function (simple sum in the example) to be executed in the Serverless Runtime assigned to this Serverless Runtime context (meaning the associated session created by the Provisioning Engine):

```
import time

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

# Function to be offloaded in this example
def sum(a: int, b: int):
    return a + b

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

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

# Wait until the runtime is ready
# Checks the status of the request of creating the SR, and sleeps 1 sec. If
still not available.
while my_cognit_runtime.status != FaaSState.RUNNING:
    time.sleep(1)

print("COGNIT runtime ready!")

# Example offloading a function call to the Serverless Runtime
# Call_sync will send to execute sync.ly to the already assigned SR. First
argument is the function and the following argos are the parameters to execute
it.
result = my_cognit_runtime.call_sync(sum, 2, 2) [*]

print("Offloaded function result", result)
```

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

print("COGNIT runtime deleted!")
```

[*] Currently, there is also the option to request the offload of the function async.ly:

```
    # Send request to offload dummy_func async.ly
    status1 = test_ready_sr_ctx.call_async(dummy_func, 4,5,3)

    # Wait until status of the task changes from WORKING to READY
    while status1.status == AsyncExecStatus.WORKING:
        time.sleep(2)
    # Wait until the task is finished and the result is there (this blocks the
execution of the offloaded task, being the second argument (value 3) the timeout
of this blocking)
  status2 = test_ready_sr_ctx.wait(status1.exec_id, 3)
```

Serverless Runtime context set up code (example using the Python SDK)

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.

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.

The Serverless Runtime needs to implement an HTTPS server capable of handling the communication with the Device Client and executing the designated function.

Architecture & Components

The Serverless Runtime provides a public Fast API REST Server that listens to FaaS and Daas requests. Multiple components are involved in the execution of the task offloading function:

1. FaaS Models: Provide the data structures needed for the requests and internal communication between function calls.
2. FaaS Parser: It is responsible for serialising the offloaded functions and for deserializing the results returned from the Serverless Runtime.
3. Logger: Provides its own log structure based on different levels of logging.
4. 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 Cling, and interactive C interpreter.
7. Py Executor: Groups the logic needed to execute Python language.

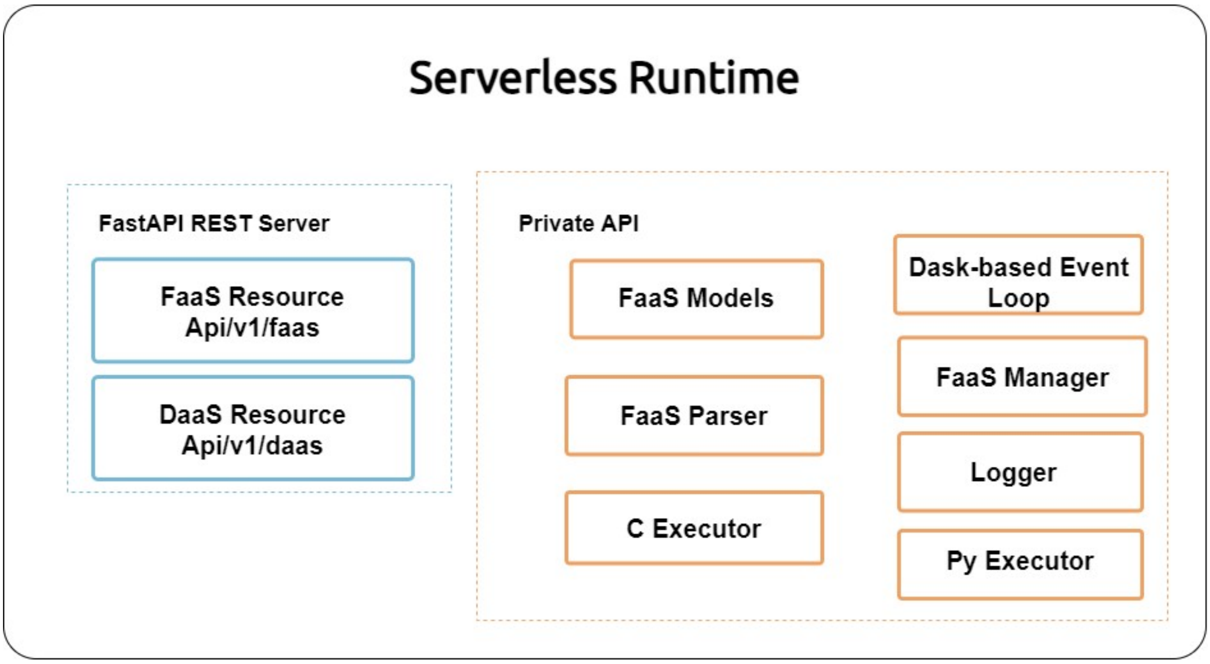

Figure 2.1. Block Diagram of Serverless runtime modules.

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 task offloading is performed by sending a JSON object to the Serverless Runtime. There are two main types of models, request and response, generated by the Serverless Runtime. All POST request share the same fields:

```
{
  "lang": “string”,
  "fc": "string",
  "params": [“string”, “string”, “string”]
}
```

Nevertheless, the body remains empty if the request is intended for checking the status of a certain FaaS UUID.

There are different response bodies, depending on the request:

- Synchronous task offloading response:

```
{
  "result": “string”
}
```

- Asynchronous task offloading response:

```
{
  "faas_uuid": “string”
}
```

- FaaS UUID status response when execution has finished:

```
{
  "state": “string”,
  "result": “string”
}
```

- FaaS UUID status response when execution is still executing:

```
{
  "state": “string”
}
```

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.
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

This component has two types of calls, synchronous and asynchronous, thus, the synchronous calls are evaluated, executed and then returned blocking the Serverless Runtime's thread.

On the other hand, we have asynchronous calls, in which the petition is evaluated and then the execution takes place in another thread without blocking the Serverless Runtime's thread. This execution has an associated state, so polling is needed from the Device Client to ensure that the execution has been successful, failed or working.

Action	Verb	Endpoint	Request Body	Response
Request a sync execution of function	POST	/v1/faas/execute-sync	JSON representation of the language of execution, function object and parameters.	Status code 200 (Success) if the execution was successful. 400 (Bad request) if the Request body is not correctly formatted. 405 (not Allowed) if there is another error with the request.
Request an async execution of function	POST	/v1/faas/execute-async	JSON representation of the language of execution, function object and parameters.	Status code 201 (Created) with the faas-uuid object. 400 (Bad request) if the Request body is not correctly formatted. 405 (not Allowed) if there is another error with the request.
Get given function execution status	GET	/v1/faas/{faas_uuid}/status	JSON representation of the state of the async. function, result if applicable and code of HTTP request.	Status code 200 (Success) if the execution was successful. 400 (Bad request) if the Request body is not correctly formatted. 404 (not Found) if the specified faas-uuid has not been found.

Table 2.2. API that defines the way to interact with a given Serverless Runtime.

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 the single point of contact for any Application Device Runtime that requests access to a Serverless Runtime, 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 FaaS 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.

Two guides will be produced to install, configure and operate a Provisioning Engine. The Administrator Guide will cover 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 will cover the use of the Provision Engine by the Device Client. It will state all the needed information that the Device Client must know, like the Provisioning Engine endpoint and the Cloud/Edge manager credentials.

Architecture & Components

The Provisioning Engine is composed of four main modules, depicted in the High Level Architecture contained in Figure 3.1:

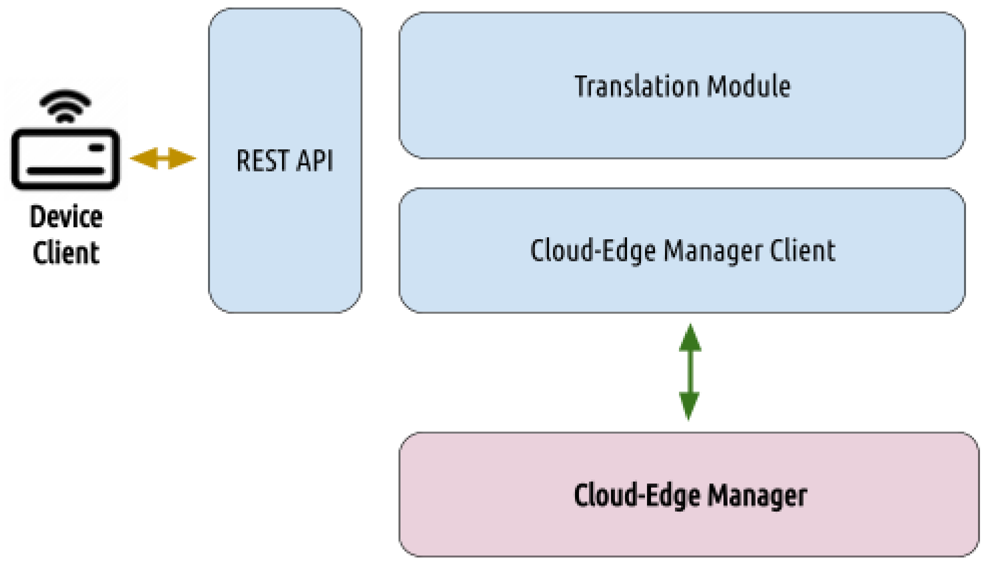

Figure 3.1. Provisioning Engine High Level Architecture

This component is stateless, it does not feature a database backend. Instead, it uses the Document Pool[2] of the Cloud/Edge Manager to save the different Serverless Runtime Description Object it received from the Device Client. A brief description of the different modules follow:

- **Rest API.** This module is in charge of the communication between the Device Client and the Provisioning Engine. It exposes a REST API specified in the API section.
- **Translation Module.** Converts the Serverless Runtime Description Object into VM Templates that can be submitted to the Cloud/Edge Manager.
- **Cloud/Edge Manager Client.** Handles all communication between the Provisioning Engine and the Cloud/Edge Manager to create and manage the lifecycle of the Serverless Runtimes.

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) described in the following table:

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.
one_xmlrpc	OpenNebula daemon contact information
flavour_mapping	Item list of tuples mapping the correspondence between Serverless Runtime flavours to Cloud/Edge Manager OneFlow VM Templates. - [nature-s3, 1] - [phoenix-mariadb, 2]

Table 3.1. Provisioning Engine Server Configuration File

All authorization from the Device Client is delegated to the Cloud/Edge Manager, which validates it against its internal DB or configured external authorization backends (such as LDAP/AD, etc). 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 Provisioning Engine.

Data Model

The first class citizen managed by the Provisioning Engine is the so-called Serverless Runtime, which is a service running in a VM in the Cloud Edge manager, in charge of processing the function offloading requests from the Device Client.

[2] https://docs.opennebula.io/6.6/integration_and_development/system_interfaces/api.html#documents

Serverless Runtimes are described using a JSON object, specified below (each attribute of the Serverless Runtime Description Object is explained in Table 3.2):

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "SERVERLESS_RUNTIME": {
      "type": "object",
      "properties": {
        "NAME": {
          "type": "string"
        },
        “ID”: {
          “type”: “integer”
        },
        “SERVICE_ID” : {
           “type”: “integer”
        },
        "FAAS": {
          "type": "object",
          "properties": {
            "CPU": {
              "type": "integer"
            },
            "MEMORY": {
              "type": "integer"
            },
            "DISK_SIZE": {
              "type": "integer"
            },
            "FLAVOUR": {
              "type": "string"
            },
            “ENDPOINT”: {
              “type”:  “string”
            },
            “STATE”: {
              “type”: “string”
            },
            “VM_ID”: {
              “type”: “string”
            }
          }
        },
        "DAAS": {
```

```
        "type": "object",
        "properties": {
          "CPU": {
            "type": "integer"
          },
          "MEMORY": {
            "type": "integer"
          },
          "DISK_SIZE": {
            "type": "integer"
          },
          "FLAVOUR": {
            "type": "string"
          },
          “ENDPOINT”: {
            “type”:  “string”
          },
          “STATE”: {
            “type”: “string”
          },
          “VM_ID”: {
            “type”: “string”
          }
        }
      },
      "SCHEDULING": {
        "type": "object",
        "properties": {
          "POLICY": {
            "type": "string"
          },
          "REQUIREMENTS": {
            "type": "string"
          }
        }
      },
      "DEVICE_INFO": {
        "type": "object",
        "properties": {
          "LATENCY_TO_PE": {
            "type": "integer"
          },
          "GEOGRAPHIC_LOCATION": {
            "type": "string"
          }
```

```
            }
          }
        }
      }
    }
  }
```

Attribute	Value
SERVERLESS_RUNTIME	JSON object describing the Serverless Runtime, comprised of a mandatory FaaS and an optional DaaS
NAME	Name of the Serverless Runtime. Optional in creation.
ID	Integer describing a unique identifier for the Serverless Runtime. Must be empty or nonexistent in creation.
SERVICE_ID	Integer describing an internal identifier for the Cloud Edge Manager. Must be empty or nonexistent in creation.
FAAS	JSON object describing the Function as a Service Runtime
DAAS	JSON object describing the Data as a Service Runtime
CPU	Integer describing the number of CPUs allocated to the VM serving the Runtime
MEMORY	Integer describing the RAM in MB of CPUs allocated to the VM serving the Runtime
DISK_SIZE	Integer describing the size in MB of the disk allocated to the VM serving the Runtime
FLAVOUR	String describing the flavour of the Runtime. There is one identifier per DaaS and FaaS corresponding to the different use cases.
ENDPOINT	String containing the HTTP URL of the Runtime. Must be empty or nonexistent in creation.
STATE	String containing the state of the VM containing the Runtime. It can be any state defined by the Cloud/Edge Manager[3], the relevant subset is “pending” and “running”. Must be empty or nonexistent in creation.
VM_ID	String containing the ID of the VM containing the Serverless Runtime, running in the Cloud/Edge Manager. Must be empty or nonexistent in creation.
SCHEDULING	JSON object describing the scheduling policies and requirements
POLICY	String describing the policy applied to scheduling. Eg: “energy, latency” will optimise the placement according to those two criteria

[3] https://docs.opennebula.io/6.6/management_and_operations/vm_management/vm_instances.html#virtual-machine-states

REQUIREMENTS	String describing the requirements of the placement. For instance, “energy_renewal” will only consider hypervisors powered by renewable energy.
DEVICE_INFO	JSON object containing information about the client device environment
LATENCY_TO_PE	Integer describing in ms the latency from the client device to the Provisioning Engine endpoint
GEOGRAPHIC_LOCATION	String describing the geographic location of the client device in WGS84[4].

Table 3.2. Explanation of Serverless Runtime Attributes

API & Interfaces

All the calls to this component are synchronous in the sense that they are evaluated in the component and returned immediately, without an external dependency that may block the call. Instead when a new resource is created, it has an associated state, a polling is needed from the Client to ensure the resource is properly created or if there is a failure on creation.

The API specified in the following table exposes methods to control the lifecycle of Serverless Runtimes, the only object managed by a Provisioning Engine:

Action	Verb	Endpoint	Request Body	Response
Create Serverless Runtime	POST	/serverless-runtimes	JSON representation of the serverless-runtime object	Status code 201 (Created) with the created dserverless-runtimeobject
Retrieve Serverless Runtime	GET	/serverless-runtimes/{id}	-	Status code 200 (OK) with the serverless-runtimeobject
Update Serverless Runtime	PUT	/serverless-runtimes/{id}	JSON representation of the updated serverless-runtime object	Status code 200 (OK) with the updated serverless-runtimeobject
Delete Serverless Runtime	DELETE	/serverless-runtimes/{id}	-	Status code 204 (No Content) if successful

Table 3.3. Provisioning Engine API Specification

[4] https://it.wikipedia.org/wiki/WGS84

Serverless Runtime provisioning is a two calls operation:

1. First, the Device Client requests the creation to the Provisioning Engine sending the Serverless Runtime Description Object.
2. Then, the Provisioning Engine, coupled with the AI-Enabled Orchestrator, creates the VM (or VMs, which do not impact the flow of the creation calls) and returns the Serverless Runtime Object filling the missing information (ID, STATE, ENDPOINT).

Afterwards, the Device Client polls regularly the Provisioning Engine until the STATE of the desired Server states a running state. This workflow is depicted in the following sequence diagram:

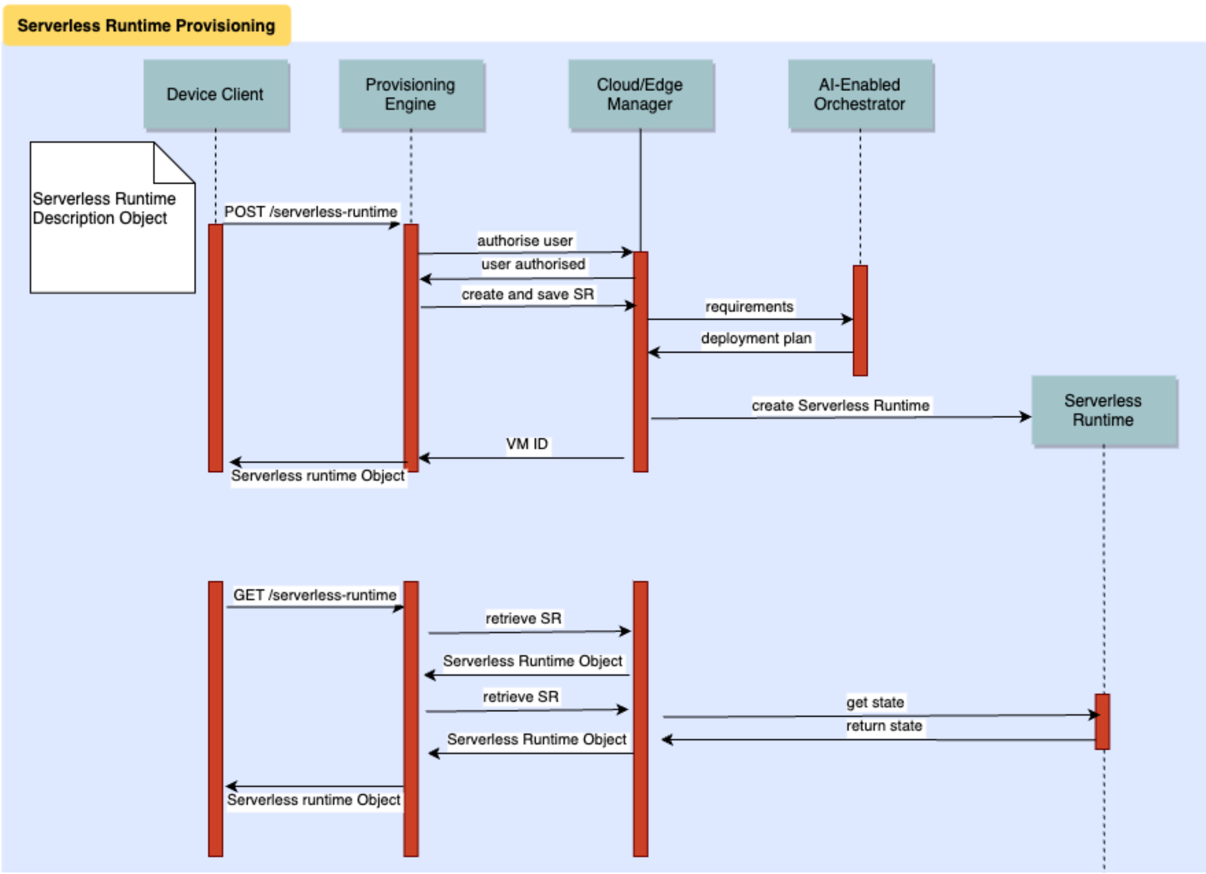

Figure 3.2. Serverless Runtime Provisioning Sequence Diagram

4. Secure and Trusted Execution of Computing Environments

Threat Model

In order to support the risk analysis presented in D2.1, we performed a threat assessment on the framework architecture. Modelling threats on the COGNIT framework aims to identify, communicate, and understand threats and mitigations on COGNIT assets. The threat model represents the information that affects the security of the framework, providing a view of the system and its environment from the security point of view. In this document, we follow the OWASP Threat Modeling methodology proposed by OWASP ThreatDragon[5][6]. We chose this tooling for its simplicity and high level approach[7], compared with other tools such as Microsoft TMT[8]. The threat model is described in the following subsections:

- **External Dependencies**: components that are not part of the application's code but could endanger it (external cloud storage, …)
- **Access Points**: interfaces that allow potential attackers to communicate with the program
- **Trust Levels**: application's access privileges to external entities
- **Data Flow Diagram**: diagram showing the flows between actors, processes and data stores, as well as the trust boundaries of the framework.

External Dependencies

External dependencies are components that are not part of the application's code but could endanger it. The development team may not have control over these things, but the organisation usually still has influence over them. When looking at external dependencies, the production environment and requirements should be taken into account first. External dependencies are documented as follows:

1. **ID**: A unique ID assigned to the external dependency.
2. **Description**: A textual description of the external dependency.

ID	Description
1	**External sources** : Data from external backend storages used by Device Clients application and by the Serverless Runtimes (e.g. an S3 cloud storage)
2	**Dedicated external user authentication** : drivers used to leverage additional authentication mechanisms or sources of information about the users (e.g. LDAP, OIDC).

Table 4.1. Threat model - External dependencies

[5] https://owasp.org/www-community/Threat_Modeling
[6] https://www.threatdragon.com
[7] Bygdas, Erlend & Jaatun, Lars & Antonsen, Stian & Ringen, Anders & Eiring, Erlend. (2021). Evaluating Threat Modeling Tools: Microsoft TMT versus OWASP Threat Dragon. 1-7. 10.1109/CyberSA52016.2021.9478215.
[8] https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool

Access Points

Access points specify the interfaces that allow potential attackers to communicate with or provide data to the program. Access points are necessary for an application to be attacked by a potential attacker. Access points are documented as follows:

1. **ID**: A unique ID assigned to the access point. This will be used to cross-reference the access point with any threats or vulnerabilities that are identified.
2. **Name**: A descriptive name identifying the access point and its purpose.
3. **Description**: A textual description detailing the interaction or processing that occurs at the access point.
4. **Trust Levels**: The level of access required at the access point. These will be cross-referenced with the trust levels defined later in the document.

ID	Name	Description	Trust Levels
1	**Device Client**		
1.1	Public API	Module used by users to communicate with the COGNIT platform	(1) Device developer (2) Device Client software
1.1.1	Serverless Runtime context	Represents the Serverless Runtime context and provides runtime operations.	(1) Device developer (2) Device Client software
1.1.2	COGNIT config modules	Enables the developer to configure the endpoint and credentials to connect to the COGNIT platform instance.	(1) Device developer (2) Device Client software
2	**Provisioning Engine**		
2.1	REST API	Exposes methods to Device Client to control the lifecycle of Serverless Runtimes, the only object managed by a Provisioning Engine.	(3) Device Client user (4) Provisioning Engine administrator (5) Provisioning Engine software
3	**Serverless Runtime**		
3.1	Public API	Module used by users to communicate with the Serverless Runtime	(3) Device Client user (7) Serverless Runtime user
3.1.1	FaaS resources API/v1/FaaS	Actual environment where the offloaded function will be executed, and linked to it may exist a DaaS	(3) Device Client user (7) Serverless Runtime user
3.1.2	DaaS Resources API/v1/DaaS	In charge of hosting any data that might need to be stored for the	(3) Device Client user (7) Serverless Runtime user

		correct execution of the offloaded function	
4	**Cloud-Edge Manager**		
4.1	Metrics REST API	Allows the Cloud-Edge Manager to receive metrics, related to the performance, of the FaaS Runtime (e.g. average execution time, number of executions per second) and DaaS Runtime (e.g. IOPS, available free capacity) pushed by serverless Runtime	(10) Monitoring agent user
4.2	Serverless Runtime deployment plan API	OpenNebula XML-RPC API and OneFlow API, expose the requirements via REST API to AI-Enabled Orchestrator for deployment plan.	(11) Monitoring AI-Enabled Orchestrator user
4.3	COGNIT environment authentication system	Provides an authentication system based on username and password or using asymmetric cryptography techniques such as TLS	(3) Device Client user (6) Provisioning Engine special user (8) Cloud-Edge Manager administrator
5	**AI-Enabled Orchestrator**		
5.1	REST API	REST API exposes to the cloud edge manager, the method of placing the Serverless Runtime on the available cloud-edge continuum resources	

Table 4.2. Threat model - Access points

Assets

Assets are documented in the threat model as follows:

1. **ID**: A unique ID is assigned to identify each asset. This will be used to cross-reference the asset with any threats or vulnerabilities that are identified.
2. **Name**: A descriptive name that clearly identifies the asset.
3. **Description**: A textual description of what the asset is and why it needs to be protected.
4. **Trust Levels**: The level of access required to access the access point is documented here. These will be cross-referenced with the trust levels defined in the next step.

ID	Name	Description	Trust Levels
1	**Device Client**		
1.1	Private API	Module encapsulating the internal	(2) Device Client software

		logic to interact with the COGNIT platform	
1.1.1	Provisioning Engine client	Module implementing the REST client and the data models to interact with the Provisioning Engine	(2) Device Client software
1.1.2	Serverless Runtime client	Module implementing the REST client and the data models to interact with the Serverless Runtime	(2) Device Client software
1.1.3	FaaS serializer	Module use to serialize the offloaded functions into a string that can be sent	(2) Device Client software
2	**Provisioning Engine**		
2.1	Translation module	Converts the Serverless Runtime Description Object into VM Templates that can be submitted to the Cloud/Edge Manager.	(5) Provisioning Engine software
2.2	Cloud/Edge Manager client	Handles all communication between the Provisioning Engine and the Cloud-Edge Manager to create and manage the lifecycle of the Serverless Runtimes.	(5) Provisioning Engine software
3	**Serverless Runtime**		
3.1	Private API	Module encapsulating the internal logic for the execution of the task offloading function	(8) Serverless Runtime software
3.1.1	FaaS models	Provide the data structures needed for the requests and internal communication between function calls.	(8) Serverless Runtime software
3.1.2	FaaS parser	It is responsible for serialising the offloaded functions and for deserializing the results returned from the Serverless Runtime.	(8) Serverless Runtime software
3.1.3	Logger	Provides its own log structure based on different levels of logging.	(8) Serverless Runtime software
3.1.4	FaaS manager	Responsible for adding an asynchronous task offloading and managing its execution status.	(8) Serverless Runtime software

3.1.5	Dask-based event loop	Provides parallel task execution and scalability of data processing applications.	(8) Serverless Runtime software
3.1.6	C executor	Groups all the logic needed to execute C language with Cling, and interactive C interpreter.	(8) Serverless Runtime software
3.1.7	Py executor	Groups the logic needed to execute Python language.	(8) Serverless Runtime software
4	**Cloud-Edge Manager**		
4.1	Edge Cluster provisioning	Based on OpenNebula OneProvision	(9) Cloud-Edge Manager administrator (10) Cloud-Edge Manager software
4.2	Serverless Runtime deployment	will be performed using VM and OneFlow deployments in OpenNebula	(6) Provisioning Engine special user (10) Cloud-Edge Manager software
4.3	Metrics	will be collected using Prometheus Server and OpenNebula monitoring	(10) Cloud-Edge Manager software (11) Monitoring agent user (12) Monitoring AI-Enabled orchestrator user
4.4	Scheduler	Will query the AI-Enabled Orchestrator API to get placements for the Serverless Runtimes	(10) Cloud-Edge Manager software (12) Monitoring AI-Enabled Orchestrator user
4.5	Provider Catalogue	Contains a list of resource providers available in the cloud-edge continuum.	(6) Provisioning Engine special user (10) Cloud-Edge Manager software
5	**AI-Enabled Orchestrator**		
5.1	Orchestrator pod	Placement recommendation, system state recording, metrics DB	(14) Cloud-Edge Manager software user (12) Monitoring AI-Enabled Orchestrator user
5.2	AI plug-ins	Backend AI service providing placement (training, verification), workload prediction, long-term planning, energy usage prediction	(13) AI-Enabled Orchestrator administrator

5.3	Orchestrator pod	UI for the orchestrator and plugins	(13) AI-Enabled Orchestrator administrator

Table 4.3. Threat model - Assets

Trust Levels

The application's access privileges to external entities are represented by the trust levels. The assets and access points are cross-referenced with the trust levels. This enables us to specify the privileges or access permissions needed to engage with each asset as well as those needed at each access point. Trust levels are documented in the threat model as follows:

1. **ID**: A unique number is assigned to each trust level. This is used to cross-reference the trust level with the access points and assets.
2. **Name**: A descriptive name that allows you to identify the external entities that have been granted this trust level.
3. **Description**: A textual description of the trust level detailing the external entity who has been granted the trust level.

ID	Name	Description
1	Device administrator	An administrator authentified to the Device and has logged in using valid credentials and performs configuration.
2	Device Client software	Software running on the client device with access to all internal modules. Software is use case specific
3	Device Client user	User authentified to the COGNIT environment using the authentication system based on username and password provided by the Cloud-Edge Manager (through Provisioning Engine). This user can communicate with the Provisioning Engine, Serverless runtime, and Cloud edge manager
4	Provisioning Engine administrator	An administrator authentified to the Provisioning Engine and has logged in using valid login credentials, he performs configuration.
5	Provisioning Engine software	Software running on the Provisioning Engine with access to all internal modules.
6	Provisioning Engine special user	User authentified to the Cloud-Edge Manager using security mechanisms such as TLS 1.3 (certificate) and able to perform deployment operations on the Cloud-Edge Manager
7	Serverless Runtime user	User authentified on an external source
8	Serverless Runtime software	Software running on the Serverless Runtime with access to all internal modules.
9	Cloud-Edge Manager administrator	An administrator authentified to the Cloud-Edge Manager and has logged in using valid login credentials and perform configuration.

10	Cloud-Edge Manager software	Software running on the Cloud-Edge Manager with access to all internal modules.
11	Monitoring agent user	User authentified to the Cloud-Edge Manager monitoring API using to push collected metrics
12	Monitoring AI-Enabled Orchestrator user	User authentified to the Cloud-Edge Manager monitoring API using to pull collected metrics
13	AI-Enabled Orchestrator administrator	An administrator authentified to the AI-Enabled Orchestrator and has logged in using valid login credentials, he performs configuration.
14	Cloud-Edge Manager software user	User authentified to the AI-Enabled Orchestrator to get placements for the Serverless Runtimes

Table 4.4. Threat model - Trust levels

Data Flow Diagram

The following figures were produced using the ThreatDragon tool and use the STRIDE methodology. It illustrates a high-level threat model. It highlights the threats between the different components and more particularly the Access Points described previously.

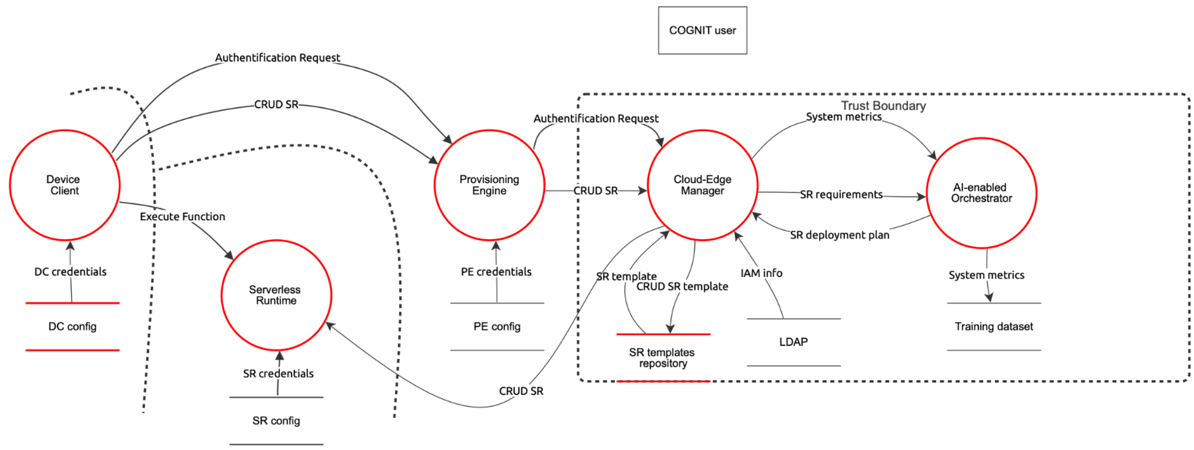

Figure 4.1. Threat model diagram

Figure 4.2 illustrates threat modelling more specific to the client device. The threats identified concern modules communicating with actors external to the Device Client.

On the one hand the COGNIT configuration module, which could be spoofed in order to allow an attacker to recover the credentials allowing access to a client device and thus cause an information disclosure.

On the other hand the Provisioning Engine client and Serverless Runtime client module which could be jammed in order to prevent them from communicating with the other components of the framework, and thus cause a denial of service.

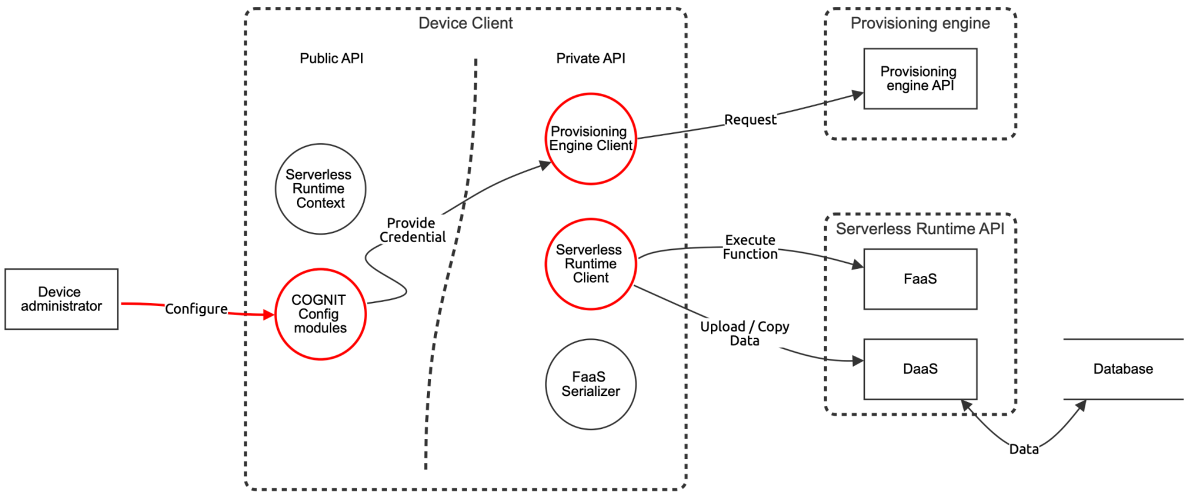

Figure 4.2. Device Client Threat model diagram

Figure 4.3 illustrates threat modelling more specific to Serverless Runtime. The threats identified concern the FastAPI Rest Server allowing the Device Client to communicate with the Serverless Runtime, but also certain internal modules.

Requests sent to serverless Runtime API services (Daas & Faas) are subject to MITM attacks.

On the other hand, executors allowing the execution of C and Python code are subject to attacks of the "Memory inspection" (Information disclosure) in order to recover information but also "Leverage Resource" by executing malicious code.

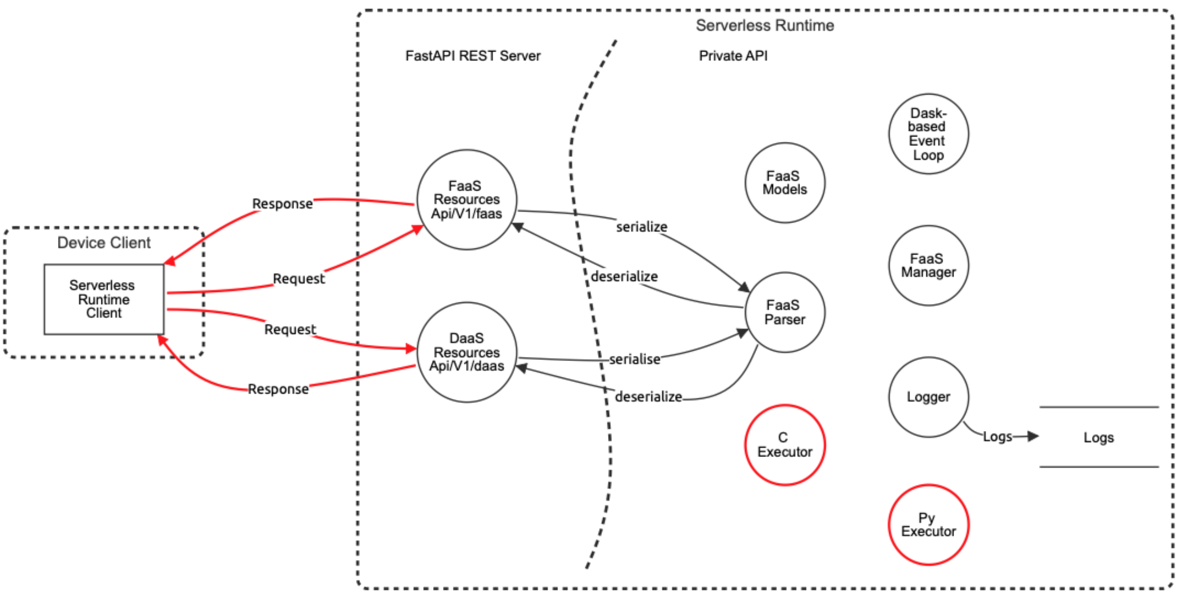

Figure 4.3. Serverless Runtime Threat model diagram

Figure 4.4 illustrates threat modelling more specific to the Provisioning Engine. The threats identified concern the Rest API allowing the Device Client to communicate with the Provisioning engine. Requests sent to the Rest API are subject to MITM attacks. On the other hand, these Rest APIs are subject to DDOS attacks, causing service unavailability.

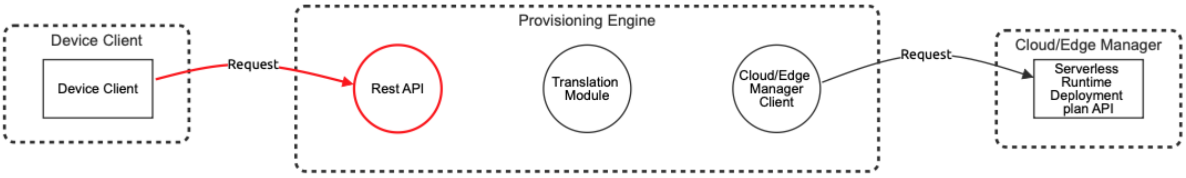

Figure 4.4. Provisioning Engine Threat model diagram

[SR6.1] Advanced Access Control

Description

Based on the risk analysis in D2.2, and the threat model of the above section, we have identified methods to exploit, detect and remediate overly permissive namespace[9] access defaults in a multi-tenant context. The use of default namespaces makes application of RBAC and other controls more difficult and is generally considered a vulnerability[10][11][12]. This default is insecure because inattentive users will not specify a namespace to deploy their resources to, those will be deployed in the default namespace where a malicious actor will more easily be able to find vulnerabilities and attack them.

Detection and mitigation

Detection of this vulnerability can be done using vulnerability scanning tools, those implement checks for specific security properties. In this case, the security policy of the COGNIT framework needs to specify that deployment of workloads to default namespaces is not allowed, or alternatively that default namespaces should not be permitted at all[13]. To that effect, we identified the OpenSCAP[14] open source ecosystem that provides security policy enforcement through the Security Content Automation Protocol (SCAP). The SCAP detection check interrogates the Cloud-Edge Manager API for the existence of a default namespace, and triggers a security compliance error if that namespace is present, notifying the platform administrators and security operators. The check should take place at least each time the framework is instantiated, when the framework is updated, and ideally on a schedule.

Once the vulnerability is detected, remediation can take place. The default namespace will be deleted, and if it contains existing workloads those should be quarantined in a

[9] Namespaces are a mechanism to share and segregate resources in a multi-tenant context. Namespaces allow users to attach authorization and policy to subsections of a cloud, separating them logically. OpenNebula's Virtual Data Center (VDC), OpenStack's Neutron Network Namespaces, or Kubernetes Namespaces are examples of this mechanism.
[10] CIS Kubernetes v1.24 Benchmark v1.0.0 L2 Master / Tenable 5.7.4 The default namespace should not be used
[11] OpenNebula VOneCloud - Multi Tenancy - removing default VDC prevents accidental deployment in default VDC
[12] CWE-1188: Insecure Default Initialization of Resource
[13] https://www.cisecurity.org/controls/v7 2.10 - Physically or Logically Segregate High Risk Applications - ensure the default namespace/VDC does not exist
[14] https://www.open-scap.org

sandboxed environment for forensics analysis. This remediation is performed using the Vacsine tool using OASIS CACAO[15] remediation playbooks.

In Figure 4.5, the Tenant A user deploys a workload (e.g. a virtual machine or a function) without specifying the target namespace. The Cloud-Edge Manager has the following insecure default configuration: when a workload has no target namespace, it is deployed on the "default namespace". This renders the workload vulnerable to malicious workloads deployed in the default namespace, that can for example listen to the workload communications or try to compromise it.

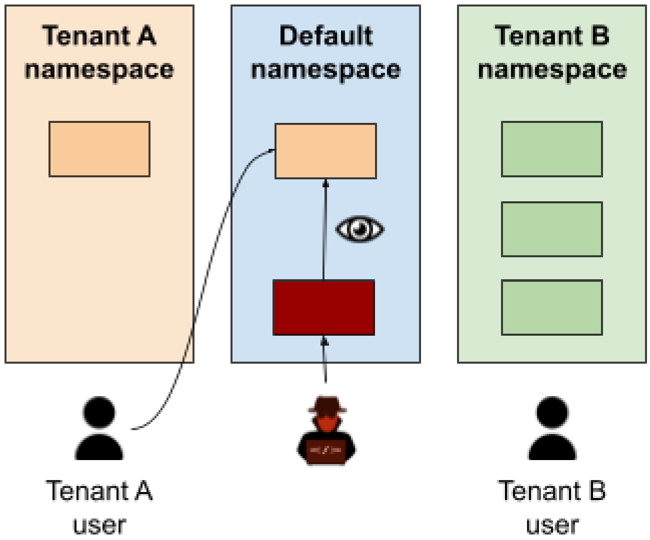

Figure 4.5. Side-channel attack on the Cloud-Edge Manager exploiting a insecure default

[SR6.2] Confidential Computing

Description

A "memory inspection" attack consists of recovering a secret stored in memory. Storing and erasing these secrets is a difficult problem when facing an attacker who can gain unrestricted physical access to the underlying hardware[16]. This is particularly problematic in an edge context where the edge devices are easily accessible. Recent examples of this kind of vulnerability include the Meltdown and Spectre attacks as well as the Heartbleed (CVE-2014-0160) vulnerability.

Based on the risk analysis in D2.2, and the threat model of the above section, we have identified methods to protect the framework against attacks that exploit those vulnerabilities: the use of confidential computing techniques can mitigate this risk. A secure CPU enclave is used to process sensitive data. The contents of the enclave, including the data being processed and the methods used to handle it, are invisible to and

[15] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=cacao

[16] Jonathan Valamehr, Melissa Chase, Seny Kamara, Andrew Putnam, Dan Shumow, Vinod Vaikuntanathan, and Timothy Sherwood. 2012. Inspection resistant memory: architectural support for security from physical examination. SIGARCH Comput. Archit. News 40, 3 (June 2012), 130–141. https://doi.org/10.1145/2366231.2337174

unknown to anyone outside of the permitted programming code. AMD SEV-SNP[17] and Intel TDX[18] are new hardware extensions developed to provide trusted execution.

In order to validate the effectiveness of this security control at reducing the risk, memory inspection tools can be used to try to extract digital artefacts from volatile memory (RAM). Confidential computing will prevent an attacker from inspecting the memory in order to extract confidential information. A concrete example in the COGNIT framework would be the attacker trying to obtain authorization tokens for the provisioning engine by inspecting an edge device memory.

[17] https://www.amd.com/en/developer/sev.html
[18] https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html