A Cognitive Serverless Framework for the Cloud-Edge Continuum

D5.11 COGNIT Framework - Demo - b

Version 1.0

23 May 2025

Abstract

COGNIT is an AI-Enabled Adaptive Serverless Framework for the Cognitive Cloud-Edge Continuum that enables the seamless, transparent, and trustworthy integration of data processing resources from 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 provides both a demonstration of some of the capabilities of the COGNIT Framework using the COGNIT testbed hosted by RISE.

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:	WP5. Adaptive Serverless Framework Integration and Validation
Nature:	DEM: Demonstrator, Pilot, Prototype
Dissemination Level:	PU: Public
Version:	1.0
Contractual Date of Delivery:	31/03/2025
Actual Date of Delivery:	23/05/2025
Lead Author:	Thomas Ohlson Timoudas (RISE), Joel Höglund (RISE)
Authors:	Monowar Bhuyan (UMU), Aritz Brosa (Ikerlan), Daniel Clavijo (OpenNebula), Marco Mancini (OpenNebula), Alberto P. Martí (OpenNebula), Idoia de la Iglesia(Ikerlan), Fátima Fernández (Ikerlan), Constantino Vázquez (OpenNebula), Pavel Czerny (OpenNebula).
Status:	Submitted

Document History

Version	Issue Date	Status[1]	Content and changes
0.1	15/04/2025	Draft	Initial Draft
0.2	15/05/2025	Peer-Reviewed	Reviewed Draft
1.0	23/05/2025	Submitted	Final Version

Peer Review History

Version	Peer Review Date	Reviewed By
0.2	15/05/2024	Antonio Álvarez (OpenNebula)

Summary of Changes from Previous Versions

This is the first version of Deliverable D5.11

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

Executive Summary

This deliverable, D5.11, presents the demonstration of the latest release (3.0) of the COGNIT Framework, corresponding to COGNIT architecture v2. It demonstrates several capabilities of the new release of the COGNIT Framework, using 3 scenarios:

1. **Offloading a function using the COGNIT Device Client:** This scenario demonstrates how to offload a function using the updated COGNIT Device Client, highlighting the changes in the second version (v2) of the architecture, compared to the first version. The v2 architecture enhances serverless functionality by decoupling infrastructure management from application execution, significantly simplifying function offloading.
2. **Horizontal scaling:** The report investigates the scalability of the COGNIT Framework in response to changes in workload, comparing both reactive and proactive scaling strategies. The demonstration shows how reactive scaling responds to increasing CPU usage, highlighting its limitations in handling natural fluctuations in CPU usage. The demonstration illustrates how proactive scaling anticipates increases in workload based on historical data, to preemptively optimize resource allocation and maintain responsiveness.
3. **Workload optimization**: The demonstration shows how different scheduling policies within an Edge Cluster can be used to achieve different goals, for example maintain CPU balance across hosts or minimize the total energy consumption.

The results presented in this report demonstrate the enhanced flexibility, efficiency, and adaptive capacity of the COGNIT Framework in dynamic cloud-edge environments.

This deliverable has been released at the end of the Fourth Research & Innovation Cycle (M27), and will be updated with one more incremental release in M33.

Table of Contents

Abbreviations and Acronyms

AI	Artificial Intelligence
IP	Internet Protocol
SR	Serverless Runtime
VM	Virtual Machine
YAML	Yaml Ain’t a markup language

1. Introduction

The initial version of the COGNIT Framework Demo (Deliverable D5.10), released in M15, includes both a demonstration of how to deploy the COGNIT Framework on a target infrastructure, and a demonstration of the COGNIT Framework in an operational environment using the COGNIT testbed.

Since there were no substantial modifications to the OpsForge tool, this document contains only a demonstration of the release 3.0 of the COGNIT Framework, corresponding to the COGNIT architecture v2.

To show the capabilities of the new release of the COGNIT Framework, we consider the following 3 scenarios:

1) A Device requesting to offload a function using the new COGNIT client.
2) The scalability of the COGNIT Framework according to changes in the workload (i.e. number of functions to be offloaded), comparing both reactive and proactive approaches.
3) The optimization of the workload (i.e. migrations of Serverless Runtimes within an Edge Cluster) according to different scheduling policies: CPU balance and energy consumption.

This document is structured accordingly, as follows. Section 2 describes the setup of the testbed used for the different scenarios. Section 3 shows how to offload a function using the COGNIT client. Section 4 shows how the COGNIT Framework addresses the scalability of Serverless Runtimes when there are changes in the workloads, using both a reactive and proactive approach. Section 5 shows how the workload can be optimized according to different scheduling policies: CPU balance and energy consumption. Finally, Section 6 concludes the document.

2. EdgeCluster setup

The cluster consists of two hosts: sm07 and sm15, as reported in Table 2.1.

Table 2.1. The characteristics of the hosts related to the Edge Cluster

Host ID	Host Name	CPUs	Memory [GB]
0	sm07	32	1032
1	sm15	256	773

The two hosts present different characteristics in terms of CPU and Energy Consumption. Figure 2.1 shows the measured power consumption of both hosts, expressed as a function of CPU usage. The host sm15 has lower power consumption for the same amount of CPU usage than sm07, as shown in Figure 2.1.

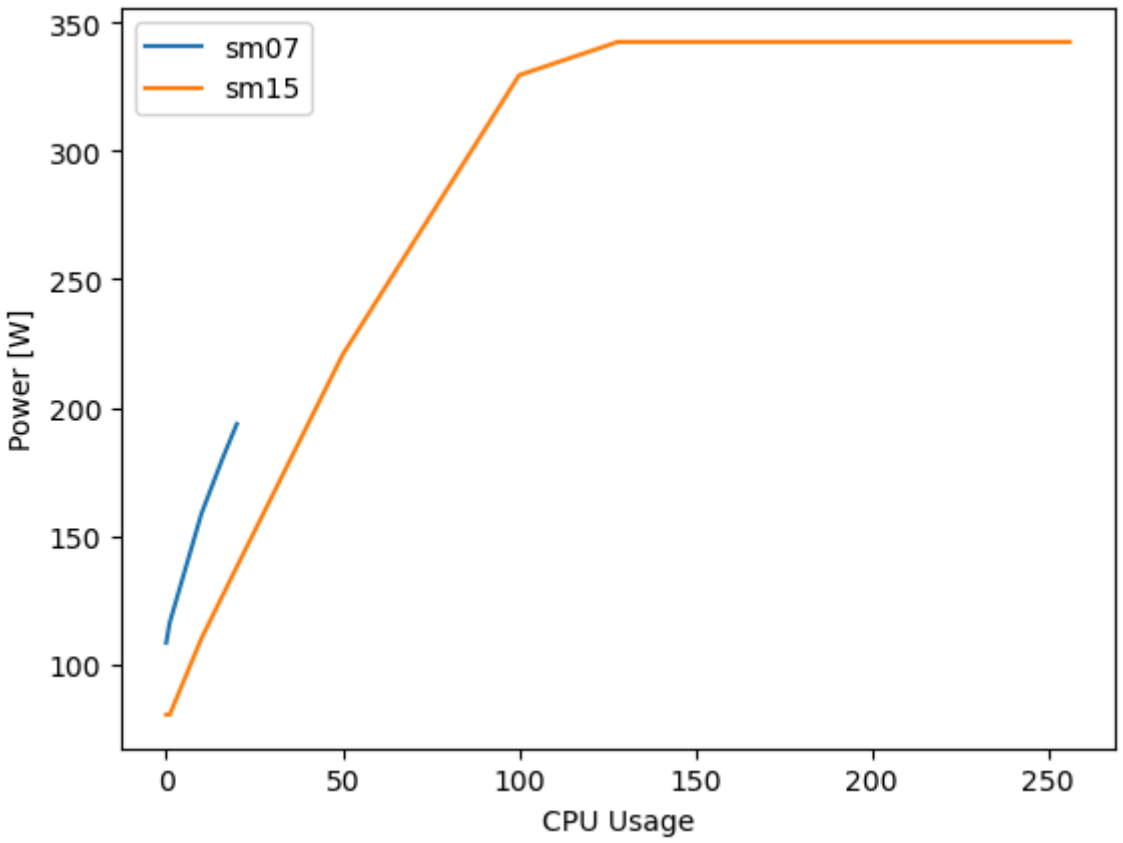

Figure 2.1. Power consumption comparison for the hosts of the Edge Cluster.

3. Offloading a Function using the COGNIT Device Client

From the Device Client standpoint, unlike in the first version of the architecture, the v2 architecture completely decouples the management of infrastructure from the user application, giving the system a real serverless behaviour, meaning that the user won't need to think about the underlying infrastructure to offload the required function. It will only set certain functional requirements (or App requirements), which will be a secondary part of the decision-making on the orchestration.

In order to be able to offload a function, the first step is to set the COGNIT instance the user is willing to work with (in this demo's case, the Testbed deployed in Luleå for COGNIT). To define a COGNIT instance, it is needed to provide the user's credentials and endpoint of the COGNIT Framework. The credentials are provided in a YAML file with the following structure:

```
Python
api_endpoint: "address:port"
#Example api_endpoint
```

```
#api_endpoint: "https://cognit-lab-frontend.sovereignedge.eu"
credentials: "<user>:<pass>"
#Example credentials
#credentials: "oneadmin:<oneadmin_pass>"
```

Once the COGNIT instance is defined, it is necessary to provide some requirements to the COGNIT Framework in order to comply with the user's needs. These requirements are used by the orchestrator who would need to match them when placing the resources to perform the execution of the function (in its particular context, that is, on this particular instance of the Device Client). As of now, the user can set the following application requirements:

```JSON
APP_REQS = {
      "FLAVOUR": "NAME_OF_FLAVOUR",
      "MAX_FUNCTION_EXECUTION_TIME": 15.0,
      "MAX_LATENCY": 45,
      "MIN_ENERGY_RENEWABLE_USAGE": 75,
      "GEOLOCATION": "LOCATION_STRING"
}
```

The Flavour field defines the type of VM on which the user wants to execute the function. This Flavour allows customizing the function execution environment based on the user needs, in terms of dependency libraries to be used by the offloaded function, needed computing resources by this execution (CPU, memory, storage, networking definition). This Flavour needs to be defined in advance of the execution of the function, and deployed within the infrastructure that the user is targeting.

On the one hand, the “MAX_FUNCTION_EXECUTION_TIME” parameter should be a soft limit (based on the policies imposed from infrastructure side) that allows the user to set a limit for the acceptable time elapsed between the moment that the request to offload a function is sent from the user application, and the moment it receives the result from the function.

On the other hand, “MAX_LATENCY” defines the maximum latency expected by the user to the Edge Cluster where the function will be executed. This is relevant for network-bound user applications, as it allows the user to specify to the orchestrator a soft limit for the placement of the Edge Cluster where the function will be executed. It is an optional requirement.

Linked to the “MAX_LATENCY” field, and as an optional requirement “GEOLOCATION” specifies the geolocation of the device, it becomes mandatory if the “MAX_LATENCY” field is set.

Moreover, the "MIN_ENERGY_RENEWABLE_USAGE" allows the user setting its requirements in terms of renewable energy usage in terms of computing resources to be used when executing the function.

The selected YAML file with the credentials and endpoint is used during the instantiation of the "DeviceRuntime" class whereas the user requirements are applied using the "init" method from the resulting "DeviceRuntime" instance. These requirements can be updated whenever the user wants by using the "update_requirements" function. The following code lines exemplify this instantiation and initialization of requirements:

```python
# Instantiate a device Device Runtime
my_device_runtime = device_runtime.DeviceRuntime("config.yml")
my_device_runtime.init(APP_REQS)
```

The result of the execution of these lines is the creation of a "my_device_runtime" object. This object is what is called a context of a Device Client, meaning that it will have a requirement ID associated within the COGNIT instance specified in the configuration file, which may change with time, but will be unique for this context until it is terminated:

At this point, the user is able to communicate with the COGNIT Framework with specific execution requirements. For demonstration purposes let's define a Machine Learning workload that will have the function to be offloaded:

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

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

    # Fit a linear regression model
     slope, intercept, r_value, p_value, std_err = stats.linregress(x_values,
y_values)

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

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

```
    return predicted_y
```

Using the “call” method of the “my_device_runtime” object the user will be able to offload this function with the requirements specifications provided.

```
Python
# Offload and execute ml_workload function
result = my_device_runtime.call(ml_workload, 10, 5)
```

The output of the execution from the user standpoint may look like this:

```
Python
--------------------------------------------------
Requirements: {'FLAVOUR': 'FlavourV2', 'MAX_FUNCTION_EXECUTION_TIME': 15.0,
'MAX_LATENCY': 45, 'MIN_ENERGY_RENEWABLE_USAGE': 75, 'GEOLOCATION': 'IKERLAN
ARRASATE/MONDRAGON 20500'} UPDATED!
------------------------------------------------------------
Predicted Y: ret_code=<ExecReturnCode.SUCCESS: 0> res=array([13.64170134,
19.20454152, 24.76738169, 30.33022186, 35.89306203]) err=None
Execution time: 3.608775 seconds
------------------------------------------------------------
```

It’s important to note the simplicity of the API of the Device Client in comparison to what was showcased in M15, where v1 of the architecture was showcased.

In the v2 architecture, in order to give the COGNIT’s offloading mechanism a pure serverless behavior, the Device Client has been decoupled into two parts:
- Device Client: Extremely simple API exposed to the user to allow them to offload functions. That is, by the use of two methods; *.init* and *.call* the users are able to perform the request with their preferences in terms of function execution.
- Device Runtime: All the mechanisms working in the background (in a transparent manner for the user) so the communication with COGNIT’s different elements is done as expected, so the defined application requirements can be considered correctly by the orchestrator, and the function and parameters are handled properly within the COGNIT Framework.

This way all the complexity of the actions’ logic is handed over to the Device Runtime, and the user does not need to take care of anything on the infrastructure side at execution time of the application.

This same behavior is the one that follows the C version of the Device Client. However in order to comply with limitations on memory footprint set by the software requirement 1.4 (SR1.4: Low memory footprint for constrained devices, shown in D3.4), some of the functionalities have been simplified.

4. Scalability

For this scenario, we are considering how the COGNIT Framework reacts to changes in the workloads (i.e. the frequency of offloaded functions requested) by increasing and decreasing the number of Serverless Runtimes (i.e. horizontal scaling).

In particular, for this scenario, we compare two different approaches for scaling the number of Serverless Runtimes of a particular flavor, using CPU usage as the metric:

- Reactive horizontal scaling based only on CPU usage observations.
- Proactive horizontal scaling based on both observations and forecasts of CPU usage.

To evaluate these approaches, the experimental setup is configured as follows:

- **Initial State**: The experiment commences with an initial deployment of 1 Serverless Runtime (SR), equipped with 5 vCPUs.
- **Monitoring**: The system continuously monitors the CPU usage of these VMs every 10 seconds.
- **Proactive Strategy Detail**: For the proactive scaling strategy, predictions leverage a 1-minute forecast horizon.

The core horizontal scaling policy, common to both reactive and proactive evaluations, is defined by these parameters:

- **Trigger Condition**: An automatic scale-out operation is initiated if the average CPU utilisation across the active SRs surpasses 80%. (For a 5-vCPU SR, this means an average usage equivalent to 4 vCPUs being fully utilized per SR).
- **Evaluation Frequency**: The trigger condition is assessed every 5 seconds, based on the average CPU usage across the active SRs.
- **Cooldown Period**: Following any scaling action, a 30-second cooldown is enforced to allow system stabilization before further elasticity adjustments can be made.

The workload applied during this scenario is designed to progressively stress the system, requiring it to dynamically scale out to maintain performance. It features a linear increase in the frequency of function execution requests over time.

- **Load Progression**: The stress test begins by generating a certain number of function requests corresponding to a CPU load of 10%, then incrementally increases this load in discrete steps of 10%.
- **Step Duration**: Each load level is maintained for 1 minute before escalating to the next.

To illustrate the differences between the two scaling strategies, we will examine their behavior under a common CPU utilization threshold, which is set at 80% of the total

available CPU capacity. This threshold provides a practical buffer against complete resource saturation.

4.1 Reactive Scaling Performance

The chart below in Figure 4.1 depicts the system's performance under a reactive scaling policy. With this approach, new Virtual Machines are provisioned only when the actual average CPU utilization across active VMs surpasses the 80% threshold.

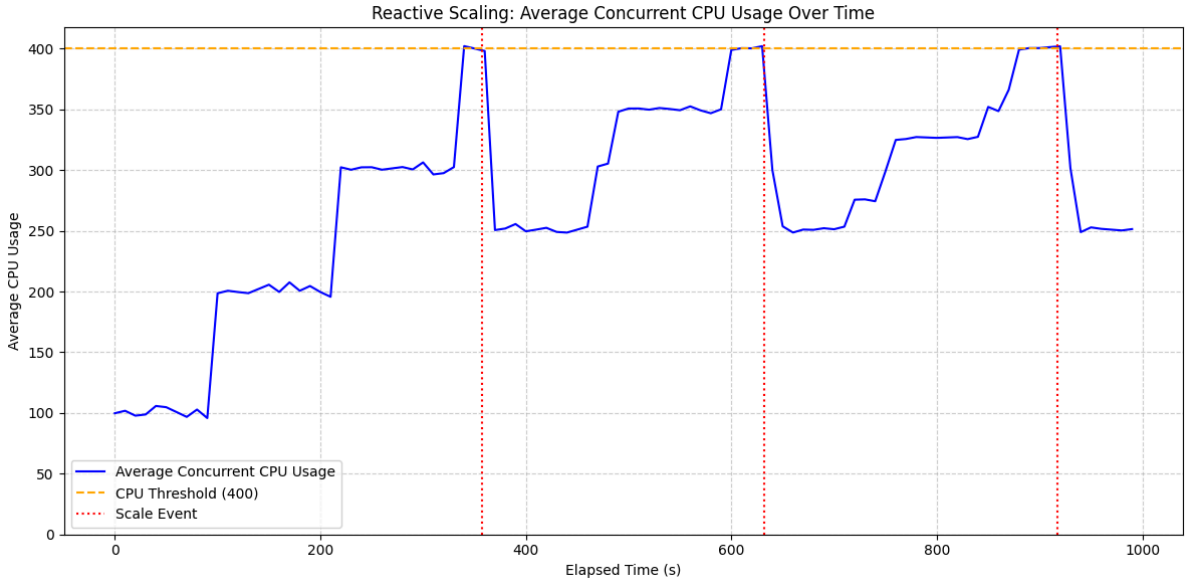

Figure 4.1. Average CPU Usage of 5 vCPU Serverless Runtimes as a function over time using the reactive scaling strategy

As this chart illustrates, the reactive system effectively responds to increasing load by adding resources once the predefined CPU limit is breached. However, this inherent delay means the system might experience periods of high CPU utilization, potentially impacting responsiveness, before scaling actions take effect. While setting a lower threshold in a reactive model might seem like an intuitive way to avoid resource saturation by triggering scaling actions earlier, this approach often leads to premature scaling and, consequently, overprovisioning. The challenge lies in the reactive system's inherent lack of foresight into future demand. Consider instead the chart below in Figure 4.2.

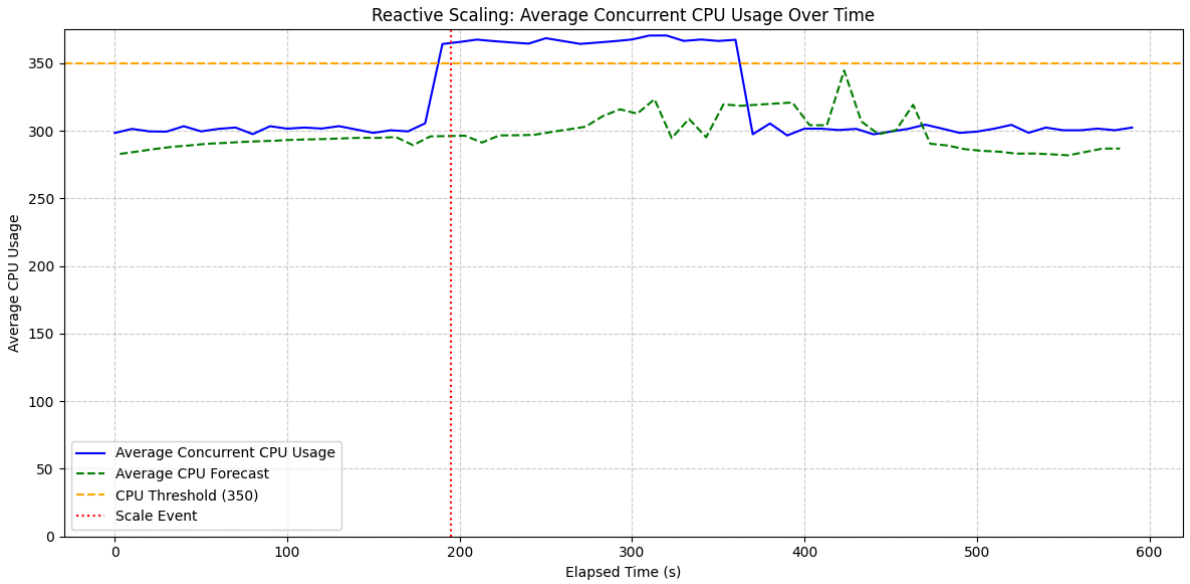

Figure 4.2. Illustration of how higher evaluation frequency in a reactive scaling strategy can trigger premature scaling due to short-lived fluctuations in average CPU Usage

The Average Concurrent CPU Usage (solid blue line) shows natural fluctuations before sharply rising to cross the CPU Threshold (dashed orange line at 350), triggering a Scaling Event (vertical red dotted line). This demonstrates how reactive scaling responds immediately to threshold breaches, even when they might be temporary fluctuations rather than sustained demand increases.

Importantly, the forecasted CPU usage (dashed green line) remains steady below the threshold throughout this period. Under a proactive scaling policy, this temporary spike would not have triggered resource allocation, as the forecast algorithm correctly identified it as a transient event rather than a sustained trend.

This comparison highlights a fundamental limitation of reactive scaling: without predictive capabilities, the system must either risk resource saturation with higher thresholds or face overprovisioning with lower ones. The reactive approach cannot distinguish between momentary spikes and genuine demand increases, potentially leading to inefficient resource allocation when temporary threshold crossings trigger unnecessary SR provisioning.

4.2 Proactive Scaling Performance

The subsequent chart in Figure 4.3 showcases the system's behavior when employing a proactive scaling strategy. In this model, scaling decisions are driven by forecasts derived from historical CPU usage patterns and anticipated future demand. A new SR is instantiated as soon as the predicted CPU usage is projected to exceed the 80% threshold.

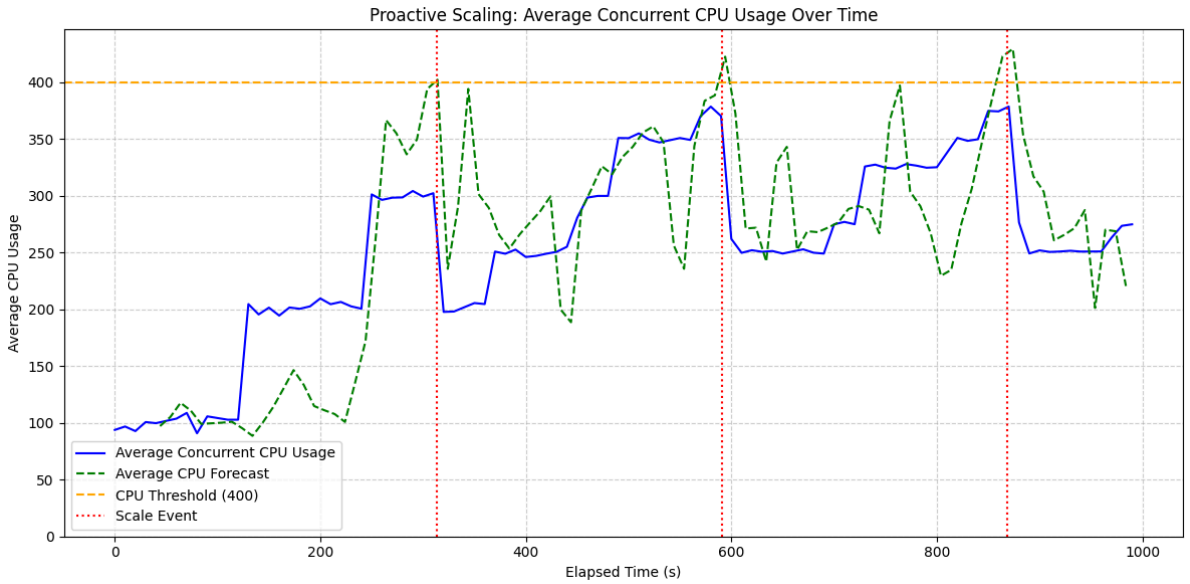

Figure 4.3. Average CPU Usage of 5 vCPU Serverless Runtimes as a function over time using the proactive scaling strategy

The proactive approach, as seen in this chart, aims to preemptively adjust resources. By anticipating load increases, the system can initiate scaling actions before the CPU utilization actually hits the critical threshold. This anticipatory scaling helps maintain consistent system responsiveness, minimizing periods of high resource contention and reducing potential waiting times for function execution. The goal is to more closely match resource allocation with true demand, leading to a more optimized and efficient use of resources compared to a purely reactive approach.

5. Workload optimization

In this scenario, we are going to demonstrate how scheduling policies affect the distribution of the Serverless Runtimes within an Edge Cluster by considering different criteria: CPU usage and energy consumption.

In order to demonstrate this, we start with the initial state described in Table 5.1 and shown in Figure 5.1. Four Serverless Runtimes (SR) are allocated on the host sm15, while sm07 is empty. Each Serverless Runtime requests 2 GB of memory. Two SRs are large (IDs 352 and 353) and need 64 CPU cores each. One mid-size SR (ID 354) requires 14 CPU cores and one small SR (ID 355) needs 2 CPU cores. In this experiment, the CPU usage of each SR is approximately equal to its requested number of CPU cores.

Table 5.1. Properties and initial allocation of the SRs.

SR ID	Requested Memory [GB]	Requested CPU Cores [-]	Current Allocation (Host name)
352	2	64	sm15

353	2	64	sm15
354	2	14	sm15
355	2	2	sm15

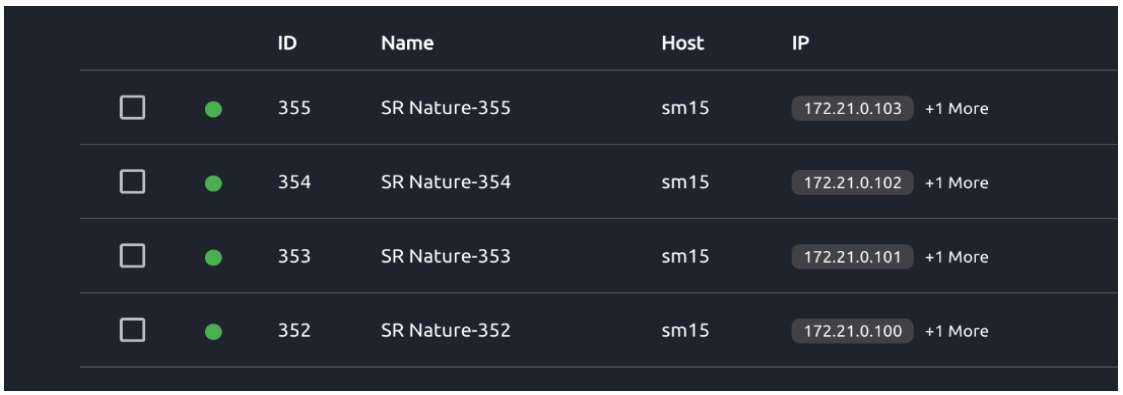

Figure 5.1. Initial allocation of the SRs.

5.1 CPU Usage Balancing

With the CPU usage balancing policy, the AI-Enabled Orchestrator will aim to keep CPU utilization as balanced as possible across the hosts. To accomplish this, we set the AI-Enabled Orchestrator to use as scheduling policy the CPU balancing.

In order to balance the cluster according to the CPU usage, the AI-Enabled Orchestrator migrates the mid-sized SR 354 to the host sm07, as shown in Figure 5.2.

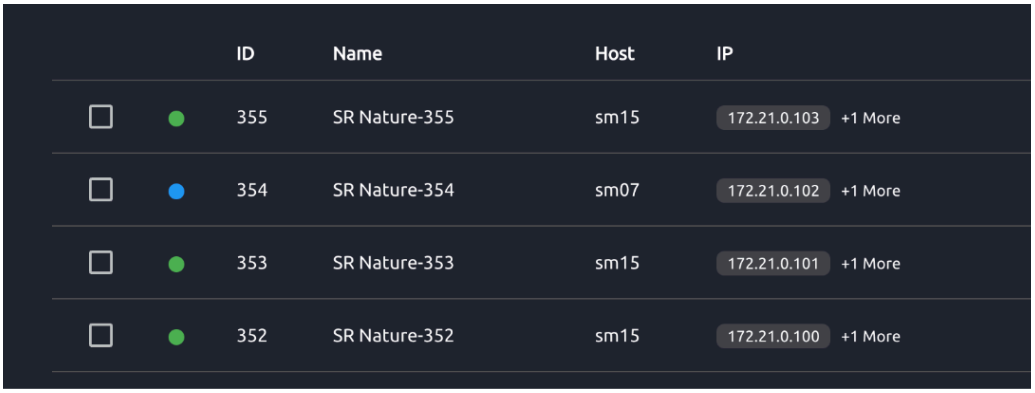

Figure 5.2. SR allocation after applying the CPU balancing policy

5.2 Energy Consumption Reduction

If we want to apply the energy saving policy to the running SRs, we need to set the AI-Enabled Orchestrator policy to energy saving, and then run the optimization.

In this case, the solution will be different, and all SRs will be allocated to the host sm15, i.e. the SR with the ID 354 should migrate back to sm15. This is because sm15 is more energy efficient than sm07 and has enough capacity to host all the SRs at the same time.

Figure 5.3 shows the result of applying the energy saving policy.

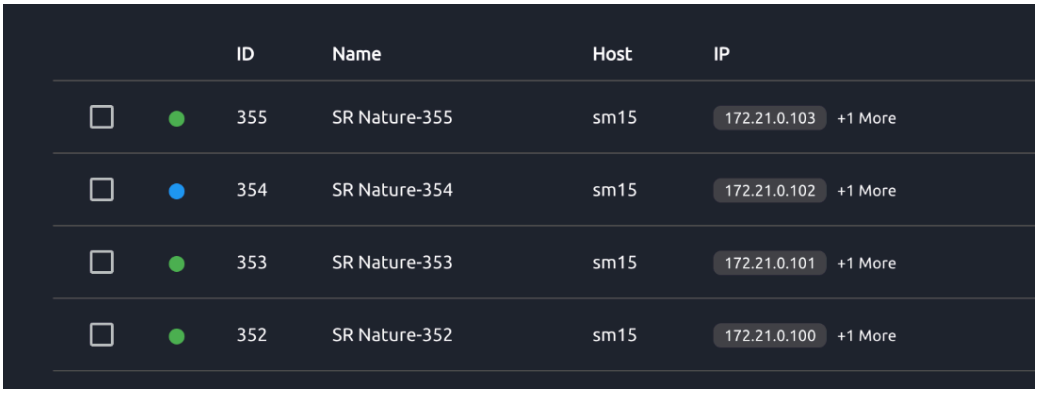

Figure 5.3 SRs allocation after applying the energy saving policy

6. Conclusions

This document demonstrates the capabilities of the release 3.0 of the COGNIT Framework, related to the COGNIT Architecture v2.0, according to different scenarios: offloading a function from a device, scalability of the framework according to changes in the frequency and load of function requests, and optimization of the Serveless Runtime workloads within an Edge Clusters due to different scheduling policies (cpu usage and energy consumption).