D3.3-Meta-Kernel Layer Module Integrated IT-2

Document Identification			
Status	Final	**Due Date**	28/02/2025
Version	1.0	**Submission Date**	28/02/2025

Related WP	WP3	Document Reference	D3.3
Related Deliverable(s)	D3.1, D3.2, D2.4	Dissemination Level (*)	PU
Lead Participant	IBM	Lead Author	Kalman Meth
Contributors	ATOS, NCSRD, PSNC, L-PIT, SUITE5, XLAB, ENG, UPC, BSC, TUBS, NKUA, CRF, IBM	Reviewers	XLAB, ATOS Francesco D'Andria (ATOS) Hrvoje Ratkajec (XLAB)

Keywords:
Cloud, Edge, IoT, MetaOS, development, integration

This document is issued within the frame and for the purpose of the ICOS project. This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101070177. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

(*) Dissemination level: **(PU)** Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page). **(SEN)** Sensitive, limited under the conditions of the Grant Agreement. **(Classified EU-R)** EU RESTRICTED under the Commission Decision No2015/444. **(Classified EU-C)** EU CONFIDENTIAL under the Commission Decision No2015/444. **(Classified EU-S)** EU SECRET under the Commission Decision No2015/444.

Document Information

List of Contributors	
Name	Partner
Francesc Lordan	BSC
Alex Barceló	BSC
Marc Michalke	TUBS
Admela Jukan	TUBS
Fin Gentzen	TUBS
Marla Grunewald	TUBS
Gabriele Giammatteo	ENG
Maria Antonietta Di Girolamo	ENG
Nikola Markovic	ENG
Manuel Gallardo	ATOS
Alberto Llamedo	ATOS
Carlos Sánchez	ATOS
Jordi Garcia	UPC
Xavier Masip-Bruin	UPC
Andreu Català	UPC
Kalman Meth	IBM
Kfir Toledo	IBM
Menelaos Zetas	NKUA
Anastasios Giannopoulos	NKUA

Document History			
Version	Date	Change editors	Changes
0.1	20/12/2025	IBM	First draft version of ToC.
0.2	27/01/2025	All	Main content for all sections
0.3	03/02/2025	All	Additional content for all sections
0.4	10/02/2025	BSC, IBM	Added tracking of functionalities + cleanup
0.5	14/02/2025	BSC, IBM	Adding missing tests information
0.6	16/02/2025	UPC, IBM	Updated Application Descriptor
0.7	21/02/2025	BSC	Address issues from internal review
0.8	28/02/2025	IBM	Draft version for quality check
1.0	28/02/2025	ATOS	Final version to be submitted

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Kalman Meth (IBM)	28/02/2025
Quality manager	Carmen San Roman (ATOS)	28/02/2025
Project Coordinator	Francesco D'Andria (ATOS)	28/02/2025

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
API	Application Programming Interface
CC	Cloud Continuum
CL	ClusterLink
CLI	Command-Line Interface
CRD	Custom Resource Definition (Kubernetes term)
CRUD	Create/Read/Update/Delete
D&PE	Distributed and Parallel Execution
DM	Deployment Manager
DX.Y	Deliverable number Y belonging to WP X
E2E	End-to-End
EC	European Commission
gRPC	(Google) Remote Procedure Call
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ICOS	IoT to Cloud Operating System
IT-x	Development Iteration x
JM	Job Manager
MM	Match Maker
OCM	Open Cluster Management (software)
OTLP	OpenTelemetry Protocol
RBAC	Role-Based Access Control
REST	REpresentational State Transfer
SCA	Security Configuration Assessment
SUC	System Use Case
TA	Telemetry Agent
TC	Telemetry Controller
TE	Topology Exporter
TG	Telemetry Gateway
UI/UX	User Interface/User Experience
URL	Uniform Resource Locator
xDS	x Discovery Service
WP	Work Package

Executive Summary

This document titled “D3.3 Meta-Kernel Layer Module Integrated (IT-2)” is a report on the final iteration of the design, development, and implementation of ICOS Meta-Kernel Layer Module. It serves as a key milestone in the project, consolidating the advancements made in previous phases and providing a comprehensive overview of the module's integration readiness for Work Package 5 (WP5).

The report reviews the functionalities derived from the project’s requirements, detailing the implementation status of each component in alignment with the overall system objectives. By building upon the architectural foundation established in previous deliverables—*“D2.4 - ICOS Architectural Design (IT-2)”* [2], *“D3.1 Meta-Kernel Layer Module Integrated (IT-1)”* [3], and *“D3.2 Meta-Kernel Layer Module Developed (IT-2)”* [4]—this document offers deeper insights into the internal implementation and integration of the Meta-Kernel components.

To foster collaboration, transparency, and wider adoption, the code for the ICOS Meta-Kernel Layer Module is released as open-source on the project's official GitHub repository (https://github.com/icos-project). It is distributed under a non-viral license, ensuring that developers and organizations can freely integrate, modify, and extend the module without restrictive obligations.

1 Introduction

1.1 Purpose of the document

This document provides a detailed report on the final iteration of the design, development, and implementation of the ICOS Meta-Kernel Layer Module. It complements the publicly available code, which has been released on the project's official GitHub repository (https://github.com/icos-project), and is ready for integration into Work Package 5 (WP5). As a key milestone in the project, this report consolidates progress from previous phases, offering a comprehensive overview of the module's current state, functionality, and integration readiness.

1.2 Relation to other project work

This document uses results of earlier documents: "D2.2 ICOS Architectural Design (IT-1)" [1], "D2.4 ICOS Architectural Design (IT-2)" [2], "D3.1 Meta-Kernel Layer Module Integrated (IT-1)" [3], and "D3.2 Meta-Kernel Layer Module Integrated (IT-2)" [4]. Changes and additions from these previous documents are recorded in this document.

1.3 Structure of the document

The content in this document is divided in 4 major chapters:

Chapter 2 describes in a nutshell the goals achieved by the Meta-Kernel layer. It recalls all the functionalities that the Meta-Kernel layer module was expected to deliver and describes the current status of their implementation.

Chapter 3 presents the Application Manifest that is used to describe an application to ICOS. This is the basic data structure that is provided to the ICOS System to enable it to perform its distributed deployment.

Chapter 4 presents the components of the Meta-Kernel layer of the ICOS System and how they work together to achieve deployment across multiple clusters.

Chapter 5 wraps up the document and extracts some conclusions.

2 Functionalities overview

Chapter 3 of Deliverable D2.2 [1] analysed the ICOS System from a user perspective, identifying key functionalities expected by users and breaking them down into fundamental interactions that provide tangible benefits. Each identified basic functionality was mapped to a corresponding System Use Case (SUC). Table 1 gathers those System Use Cases where the Meta-Kernel layer module is wholly or partially responsible for the implementation of the described functionality and presents the final status of its implementation.

Table 1 System Use Cases where the Meta-Kernel layer is involved

ID	System Use Case	Description	Status
Continuum Management			
SUC_CC_1	Node On-Boarding	Allows to add a new node to an ICOS Cloud Continuum	Achieved
SUC_CC_3	Configure Node	Allows to specify the required configuration to ensure that the new node can join the CC	Achieved
SUC_CC_4	Join the Cloud Continuum	Allows to make the new node start communicating with other nodes in the Cloud Continuum	Achieved
SUC_CC_5	Visualize Cloud Continuum Topology	Allow the user to graphically visualize the topology of a Cloud Continuum	Achieved
SUC_CC_6	Install and Configure ICOS Discovery Service	Allows to set-up a new Discovery Service for an ICOS Cloud Continuum	Achieved
Runtime Management			
SUC_RT_1	Define Application	Allows the definition a new application and all the details for its deployment	Achieved
SUC_RT_2	Define Application Deployment Descriptor	Allows to describe the deployment of an application (e.g., services and interconnections between them)	Achieved
SUC_RT_3	Define Application Requirements and Policies	Allows to define a set of non-functional requirements and policies that ICOS should optimize while running the application	Achieved
SUC_RT_4	Deploy Application	Allows to deploy an application in an ICOS Instance	Achieved
SUC_RT_5	Deploy in a different or multiple Controllers	Allow to deploy the application in an ICOS instance composed of multiple Controllers based on user requirements and matchmaking results	Pending
SUC_RT_6	See Application Logs and Status	Review the status and the logs of all the application components	Achieved

SUC_RT_7	See Application and Resources Performance Data	See and query the performance metrics related to the resources that are hosting the application and the metrics generated by the application itself	Achieved
SUC_RT_8	Review Events and Alerts	Review all the events and alerts generated from the runtime, security, and intelligence layer about the application optimization	Achieved
SUC_RT_9	Apply suggested deployment optimizations	Apply suggestions generated by the system to optimize the deployment of the application	Achieved
SUC_RT_10	Apply recovery actions	Apply recovery actions generated by the system	Achieved
SUC_RT_11	Delete Application	Un-deploy and remove an application	Achieved

With the exception of SUC_RT_5, the Meta-Kernel layer successfully supports all defined System Use Cases. The Application Manifest, introduced in Chapter 3, facilitates the definition of new applications by specifying deployment details, including service configurations and interconnections, while also enabling runtime optimization policies (SUC_RT_1, SUC_RT_2, and SUC_RT_3). The remaining Use Cases are implemented through interactions between various components of the layer, as outlined in Chapter 0. Many of these interactions have been previously documented in architecture deliverables (D2.2 [1] and D2.4 [2]) and in reports on the Meta-Kernel implementation (D3.1 [3], D3.2 [4]). Section 4.2 provides updates on selected interactions and details their final implementation status.

Despite discussions and the design of support for multiple Controllers, as detailed in Deliverable D2.4 [2], SUC_RT_5 was not implemented. The development efforts were prioritized towards the implementation of the other System Use Cases, which were deemed more critical for the overall functionality of the Meta-Kernel layer. As a result, while the conceptual framework for SUC_RT_5 exists and is described in Section 4.10 of D2.4 [2], its implementation was not completed.

3 Application Manifest

ICOS is a technology agnostic Meta Operating System. This section defines the minimalist syntax required to provide ICOS with an agnostic Application Descriptor manifest to run on the ICOS system. The Application Manifest serves as a critical input to the Meta-Kernel layer since it defines the application's components, the dependencies among them and the policies that will drive its deployment. Although the Application Manifest was already introduced in D3.1 [3] Section 4.2.2, its syntax has undergone significant changes to accommodate the new functionalities of the layer.

3.1 Application Model

An application is made of a set of **components** and the interactions between them. Different entities that do not make sense separately (for instance a Deployment and the Service exposing it) should be defined as part of the same component. Therefore, they will run in the same resources (which could be a list of nodes in the same cluster). The service exposes the deployed component to facilitate communication with other components of the application.

Note that having two manifests that have requirements in the same component does not make sense, because in fact we created the component so that we can group together entities that do not need computational resources with entities that need them. (Aka. Docker compose does this grouping and calls it a service).

3.2 Overall structure of the application descriptor

The application descriptor has three mandatory sections: **components**, **dependencies** and **manifests** as shown in the example from Snippet 1.

```
name: uc4-app-emds
description: "the description"
components:
  - name: house-component
    type: docker
    manifests:
      - name: house-manifest
dependencies:
 # producer/consumer components information
manifests:
 # List of manifests in the specified format
```

Snippet 1: Example of the mandatory sections for the application descriptor

3.2.1 Components Section

Components are application objects that can be executed at different nodes of the continuum. For each component, we need to specify the following details:

- **name:** The component's name.
- **type:** The orchestration engine for which the manifests are prepared (Kubernetes, Docker or both).
- **manifests:** A list of manifests belonging to this component.

There are different cases to define the application components, each with specific manifests describing how to deploy the component on a different orchestration engine. Below are the examples of different cases.

Case 1: Two components with `kubernetes` type (single manifest each), as shown in Snippet 2. In this case, Kubernetes nodes will be selected for both the components.

```
name: my application
description: "The description of application"
components:
  - name: component-1
    type: kubernetes
    manifests:
      - name: comp-1-manifest
  - name: component-2
    type: kubernetes
    manifests:
      - name: comp-2-manifest
```

Snippet 2: Example of manifest with two kubernetes-type components

Case 2: Two components with `docker` type (single manifest each) as shown in Snippet 3. In this case, Docker nodes will be selected for both the components

```
name: my application
description: "The description of application"
components:
  - name: component-1
    type: docker
    manifests:
      - name: comp-1-manifest
  - name: component-2
    type: docker
    manifests:
      - name: comp-2-manifest
```

Snippet 3: Example of manifest with two docker-type components

Case 3: Two components with mixed types (`kubernetes`, `multiple`) as shown in Snippet 4. The "`multiple`" type indicates that both, Kubernetes and Docker, manifests are provided for that specific component. However, if one of the component types is `kubernetes`, the rest of the component will be deployed on a Kubernetes cluster. It is not allowed to mix component types.

```
name: my application
description: "The description of application"
components:
  - name: component-1
    type: kubernetes
```

```
    manifests:
      - name: comp-1-manifest
  - name: component-2
    type: multiple
    manifests:
      - name: comp-2-manifest-kubernetes
      - name: comp-2-manifest-docker
```

Snippet 4: Example of a manifest with a component with multiple types

Case 4: Two components with multiple type (two manifest each) as shown in Snippet 5. In this example, `component-1` and `component-2` are both flexible and can be deployed either on Kubernetes or on Docker nodes.

```
name: my application
description: "The description of application"
components:
  - name: component-1
    type: multiple
    manifests:
      - name: comp-1-manifest-kubernetes
      - name: comp-1-manifest-docker
  - name: component-2
    type: multiple
    manifests:
      - name: comp-2-manifest-kubernetes
      - name: comp-2-manifest-docker
```

Snippet 5: Example of manifest with many components with multiple type

The components section includes the list of components and two optional fields: Requirements and Policies.

3.2.1.1 Requirements

Requirements section specifies the requested resources for the application. It can apply to the entire application, in which case the section will appear at the same level as the components, or to individual components, in which case it will be specified under the component it applies to. Note that the former option has not been considered. As each component may have different resource requirements and specifying requirement per component level allows to allocate most suitable resources rather than applying a uniform allocation The latter option, where the requirements are specified inside each component, is considered as shown in the example in Snippet 6.

```
name: my_application
description: "my app description"
components:
```

```
  - name: component-1
    type: kubernetes
    manifests:
      - name: comp-1-manifest
    requirements:
      architecture: intel
      cpu: 0.5
      memory: 200Mi
      nodelabel:
        key: node-label-key
        values:
          - node-label-robot1
          - node-label-robot2
      devices: squat.ai/video
```

Snippet 6: Example of a manifest indicating the requirements for a component

Accepted requirements are:

- architecture: intel/x86_64 or arm64/aarch64
- cpu: number of cpu cores requested.
- memory: amount of memory requested; the unit can be specified (following Kubernetes syntax).
- devices: devices that are necessary to be present for the component to run.
- nodelabel: often the component is intended to run on specific nodes of the ICOS system (e.g., specific robots, specific houses, specific cars…). In this case a (key, value) pair can be specified to identify the nodes. The value can be a list of nodes, which implies that replicas of the component will be deployed in all of the specified nodes.

A note on defining the requirements for the ICOS header: In Kubernetes, the smallest deployable units of computing are the Pods, so resources are allocated to Pods, whereas in ICOS the resource allocation unit is the **component**. Since each container has its own resource needs, the user (application developer) needs to abstract (add up) the needs of each container (which can be part of a workload management controller such as Deployment, ReplicaSet, Job in Kubernetes, etc) that belong to the same component. Similarly, if a component contains several manifests, each with its own list of requirements, the requirements for each manifest should be merged in the header, as shown in the example from Snippet 7.

```
manifests:
  - name: comp-1-manifest-1
    manifest:
      apiVersion: apps/v1
      kind: Deployment
      metadata:
        name: busybox-1
        namespace: demo
      labels:
        app: busybox-1
```

```
      spec:
        selector:
          matchLabels:
            app: busybox-1
        template:
          metadata:
            labels:
              app: busybox-1
          spec:
            containers:
              - name: busybox-1-container
                image: busybox-1
                command: ["sleep", "100"]
                resources:
                  requests:
                    cpu: "0.5"
                    memory: "2Mi"


manifests:
  - name: comp-1-manifest-2
    manifest:
      apiVersion: apps/v1
      kind: Deployment
      metadata:
        name: busybox-2
        namespace: demo
      labels:
        app: busybox-2
      spec:
        selector:
          matchLabels:
            app: busybox-2
        template:
          metadata:
            labels:
              app: busybox-2
          spec:
            containers:
              - name: busybox-2-container
```

```
                image: busybox-2
                command: ["sleep", "100"]
                resources:
                  requests:
                    cpu: "0.5"
                    memory: "3Mi"
```

Snippet 7: Example of manifest with a component with multiple manifests each with its own requirements

The implications of placing two manifests in the same component is that their requirements will be merged, and a suitable target will be searched that matches ALL the requirements from both manifests. Therefore, it is understood that if two manifests can run apart from each other, they should belong to different components so that the matchmaking process has more options.

Enforcement of requirements (scheduling or runtime): When are requirements taken into account? In the current version, they are considered only at scheduling time. However, some of the requirements could also be monitored and enforced at runtime. In any case, the remediation action to take would be defined by ICOS (i.e. migrate).

3.2.1.2 Policies

Policies are more dynamic than requirements. They are rules that can be defined by the user. Like requirements, policies can be defined inside a component or outside if they apply to all applications. The syntax for the policies section is presented in Snippet 8.

```
policies:
  - name: policy-name
    type: policy-type
```

Snippet 8: Syntax of the policies section

The rest of the dictionary entries depend on the type of policy. Supported types are:

- Security: Defines security requirements.
- Scheduler: Specifies scheduling strategies.
- Custom: User-defined policies for runtime.

Snippet 9 illustrates an example of a security-type policy.

```
policies:
  - name: securitypolicyexample
    type: security
    level: low    # or medium or high
    remediation_action: none # or migrate
```

Snippet 9: Example of definition of a security-type policy

In this type of policy, the "`level`" value specifies the minimal security level for the application to run: `low`, `medium` or `high`. The system will define the thresholds. The different security level determines how hardened an ICOS node is against potential threats. The security level of an ICOS node is evaluated using the SCA (Security Configuration Assessment) metric. The most secure node is the one with the highest SCA score (an integer ranging from 0 to 100). This policy is considered both during scheduling and at runtime.

Those policies with type "scheduler" - e.g., the policy defined in -- are checked during scheduling and defines the scheduling strategy to drive the deployment of the components.

```
policies:
  - name: scheduling-scoring
    type: scheduler
    performance: 0.5
    energy: 0.3
    security:0.2
```

Snippet 10: Example of the definition of ascheduler-type policy

Additionally, custom policies can be defined by the user for instance to consider application-specific metrics. In this case, the metric and remediation action are provided by the user, as shown in Snippet 11, and the system verifies it at runtime.

```
policies:
  - name: my-policy
    type: custom
    fromTemplate: compss-under-allocation
    remediation: scale-up
    variables:
      thresholdTimeSeconds: 120
      compssTask: provesOtel.example_task
```

Snippet 11: Example of the definition of a custom-type policy

More details about the fields of the policy are provided in the Annex section 1.1.2.12 of the deliverable D3.2 [4].

3.2.1.3 Communication

This section of the application descriptor specifies relationships between components of the application, as shown in example in Each component has:

- `outgoing`: which specify the service the component provide to another component.
- `incoming`: which specify the service that the component depends on from another component.

```
name: HelloworldClusterlink
description: "Helloworld application to test clusterlink"
components:
  - name: hello
    type: kubernetes
    manifests:
      - name: hello-deployment
      - name: hello-configmap-script
    communication:
      - incoming: world-svc

  - name: world
```

```
    type: kubernetes
    manifests:
      - name: world-deployment
      - name: world-configmap-script
      - name: world-configmap-requirements
      - name: world-svc
    communication:
      - outgoing: world-svc
```

Snippet 12. It outlines how the interactions between components need to be defined in terms of communication between them (which is necessary if the components run in different clusters, so that ClusterLink can be set up). Each component needs to define its communication needs within its definition. This information will also be used during scheduling to find a more accurate allocation.

Each component has:

- `outgoing`: which specify the service the component provide to another component.
- `incoming`: which specify the service that the component depends on from another component.

```
name: HelloworldClusterlink
description: "Helloworld application to test clusterlink"
components:
  - name: hello
    type: kubernetes
    manifests:
      - name: hello-deployment
      - name: hello-configmap-script
    communication:
      - incoming: world-svc

  - name: world
    type: kubernetes
    manifests:
      - name: world-deployment
      - name: world-configmap-script
      - name: world-configmap-requirements
      - name: world-svc
    communication:
      - outgoing: world-svc
```

Snippet 12: Example of the definition of a component communication

3.2.2 Manifests

The following types of manifests are supported:

- `docker`: A Docker compose manifest.
- `kubernetes`: A Kubernetes manifest.
- `multiple`: Means both Kubernetes and dcompose manifests are provided. This gives more flexibility to the ICOS system to schedule the components on any resource, regardless of the orchestration engine. Whereas if a component is specified type `kubernetes`, it can only be scheduled on Kubernetes clusters.

Snippet 14 illustrates how to create ICOS header from docker compose manifest depicted in Snippet 13.

It is important to note that, in the current ICOS implementation, it is not allowed to mix component types. That is, if one of the components is of type `kubernetes`, the rest should be either `kubernetes` or `multiple`. In this case, the application will be scheduled on Kubernetes clusters.

```
manifests:
  - name: comp-1-manifest
    manifest:
      version: '3.8'
      services:
        frontend:
          image: example/webapp
          deploy:
            resources:
              reservations:
                cpus: '0.5'
                memory: 200M
```

Snippet 13: Manifest implementing a component using docker compose

```
components:
  - name: component-1
    type: docker
    manifests:
      - name: comp-1-manifest
    requirements:
      cpu: 0.5
      memory: 200M

manifests:
  - name: comp-1-manifest
    manifest:
      # Docker compose manifest like the one contained in Snippet 13
```

Snippet 14: Example of components section using the docker compose manifest in Snippet 13

4 Meta-Kernel Layer Module Design

As shown in Figure 1 (a merged and updated version of Figures 6 and 13 presented and discussed in D2.4 [2]), the Meta-Kernel Layer is now present in both the ICOS Controller and the ICOS Agent.

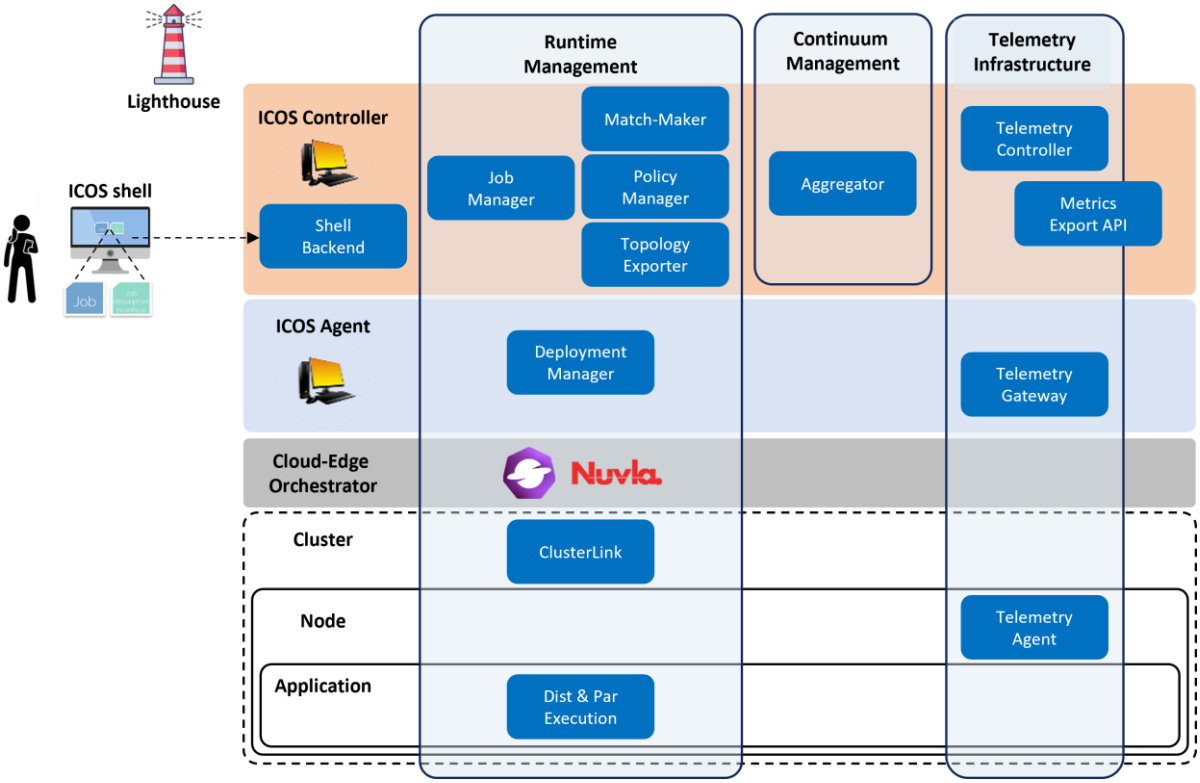

Figure 1: The ICOS Meta-Kernel Layer

This chapter focuses on the functionalities and interactions of the ICOS components and it concentrates on the changes and additions from previous documents D3.1 [3], D3.2 [4], and D2.4 [2].

For each component, we describe the basic functionality including interactions with other components, we point to other documents (e.g. D2.4, D3.1, D3.2) where more details are available, we indicate how testing was performed, the public repository where the code can be found and the license applied to it.

4.1 Components

4.1.1 Shell

The ICOS Shell consists of two different interfaces that a user can use to interact with the ICOS system. These are namely the Command Line interface (CLI) and the graphical user interface (GUI). Both use the ICOS Lighthouse as a reference point to obtain the address of an ICOS controller and then connect to this controller's API to issue further commands. While different in their appearance, they both utilize the same endpoints of the Shell Backend.

The respective Shell Client also takes care of managing the user's authentication by saving the auth token and attaching it to every request, allowing for proper authorization and authentication through all ICOS components.

The following figure shows all currently possible interactions of the Shell Client and Shell Backend with all connected components, except for the Lighthouse.

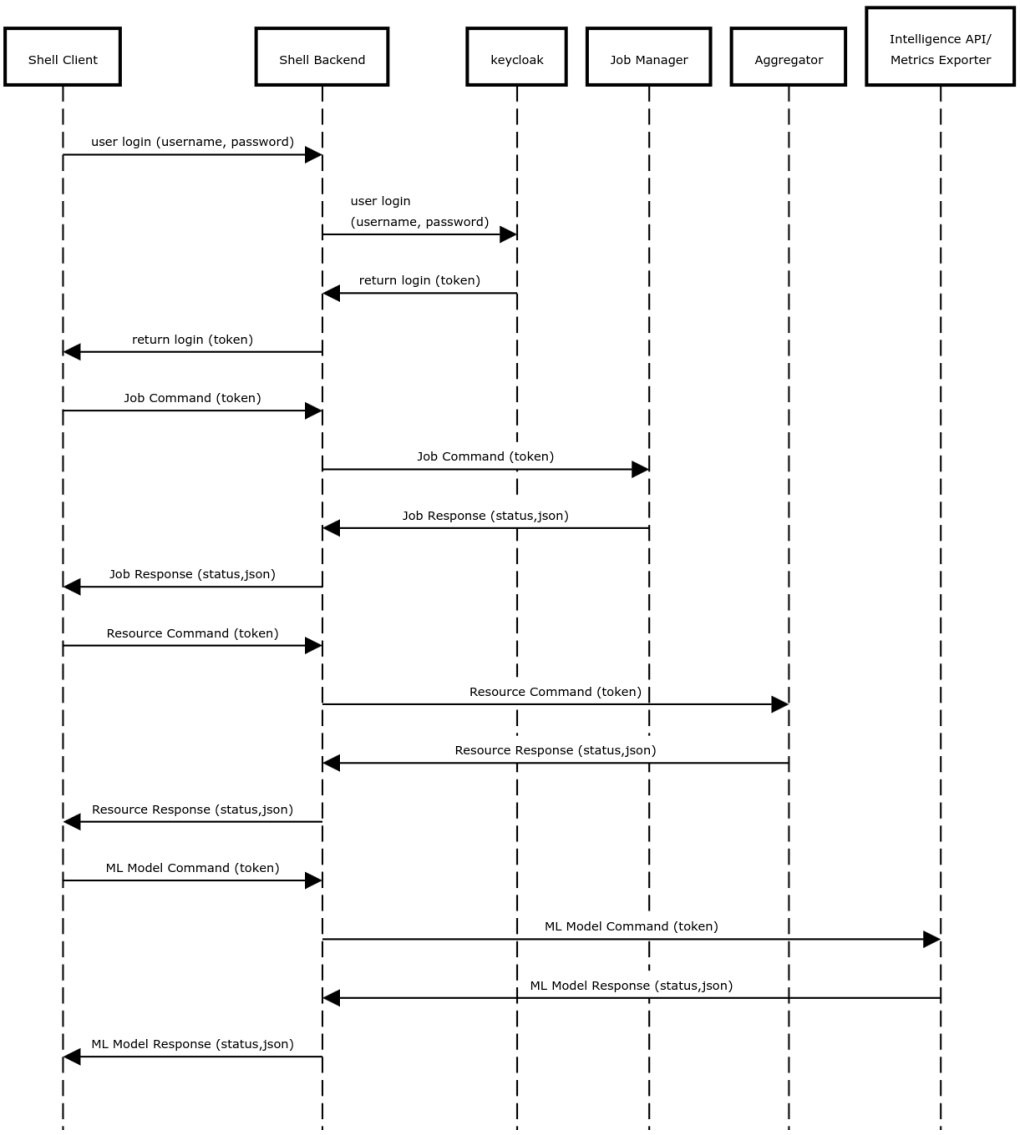

Figure 2: Shell interaction

4.1.1.1 Testing and Validation

Shell Client is tested through unit tests at build time as well as integration tests after every major change to the code.

4.1.1.2 Code

The code of the Shell component can be found within the backend in the Shell repository of the ICOS Github: https://github.com/icos-project/Shell. The component is released under the Apache license 2.0.

4.1.2 Shell Backend

As shown in the diagram of the previous section, the Shell Backend offers interactions with the components Job Manager, Aggregator and Metrics Exporter of the Intelligence API as well as Keycloak[1] for authentication management. The Shell Backend exposes a single API with dedicated API endpoints that define possible interactions with these components and then forwards the calls to the corresponding component endpoints.user api.

Depending on the responsible component, the Shell Backend forwards the respective requests to the appropriate microservice. Furthermore, the Shell Backend regularly connects to the Lighthouse in a keepalive manner to update its registration. Running as a containerized application, the Shell Backend supports configuration through either a configuration file, environment variables, or a mixture of both.

Outside of these externally provided settings, the Shell Backend itself remains fully stateless, moving all state management tasks to the Shell Client. A full list of all API endpoints and available functionalities has been provided in D3.2 [4].

4.1.2.1 Testing and Validation

Shell Backend is tested through unit tests at build time as well as integration tests after every major change to the code.

4.1.2.2 Code

The code of the Shell Backend component can be found within the backend in the Shell repository of the ICOS Github: https://github.com/icos-project/Shell . The component is released under the Apache license 2.0.

4.1.3 Lighthouse

The Lighthouse acts as a controller repository storing the addresses of all registered controllers and removing inactive ones. Any device or client can then use this repository to retrieve a list of available controller addresses to connect to. Controllers need to be authorized to add their address to the Lighthouse. If they remain inactive for more than a configured timeout, they are removed from the list. Since the Lighthouse shares the same code base as the Shell Backend, the configuration mechanisms remain the same. The API endpoints have been described in D3.1 [3].

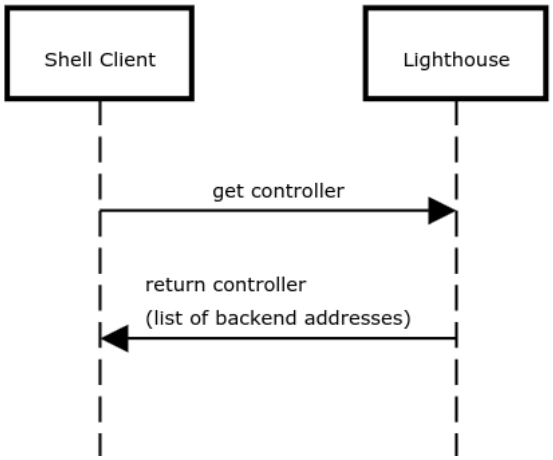

Figure 3: Lighthouse Interaction

[1] https://www.keycloak.org/

4.1.3.1 Testing and Validation

Lighthouse is tested through unit tests at build time as well as integration tests after every major change to the code.

4.1.3.2 Code

The code of the Lighthouse component is published in the lighthouse-registration-service repository within ICOS' Github: https://github.com/icos-project/lighthouse-registration-service. The component is released under the Apache license 2.0.

4.1.4 Job Manager

The Job Manager serves as a persistence level meta-orchestrator. It receives new applications and operates to collect the allocation for each of the jobs, passing the information into a database. It offers different endpoints to allow other components to obtain information about the jobs and manage them according to business needs. It serves as the main source of truth for the Deployment Managers (see Section 4.1.8) which collect jobs and apply according to the results from the allocator.

4.1.4.1 API Endpoints

The list of API endpoints was first introduced in Section 2.2.5.1 from D3.1 [3], and extended during IT-2 as described in Section 2.1.4 from D3.2 [4]. For this last version, the previously defined endpoints (in Table) are no longer supported and are replaced by the ones in Table.

All URIs are relative to http://${CONTAINER_ADDRESS}

Table 2: Job Manager's removed API Endpoints

Endpoint	HTTP Method	Request Body	Description
/jobmanager/resources/status/{job_uuid}	GET		Get resource status
/jobmanager/resources/status/	PUT	Resource entity and condition	Update resource status
/jobmanager/jobs/executable/{orchestrator}/{owner_id}	GET		Get job by state

Table 3: Job Manager's new API endpoints

Endpoint	HTTP Method	Request Body	Description
/jobmanager/status/{job_id}	GET		Get resource status
/jobmanager/status/	PUT	Json with job or jobgroupid entity and newstatus	Update resource status
/jobmanager/jobs/executable/{orchestrator}/{agent_id}	GET		Get job by state

4.1.4.2 Testing and Validation

The Job Manager has been developed using interfaces to facilitate mocking of specific resources. It has a set of unit tests that can be run with the standard tooling of golang programming language. Apart from that, the main validation were conducted by port-forwarding the resources from the testbed, to be capable of detecting issues in the local run of the service. The validation then was checked with a helloworld application following the application description of the project. In case of specific integration tests, a custom helloworld application was developed and tested in the same way with port-forward connectivity to the staging testbed.

4.1.4.3 Code

The source code of the JobManager is publicly available in the job-manager repository of the project's Github: https://github.com/icos-project/job-manager. The component is released under the Apache license 2.0.

4.1.5 Matchmaker

The Matchmaker gets the application descriptor YAML file from the Job Manager and calls the aggregator to get the ICOS topology (JSON). It extracts the resources required for each component of the user application, matches them with available resources and finds the best possible solution. It then responds to the Job Manager for the deployment of applications on targeted nodes as shown in the Figure 4.

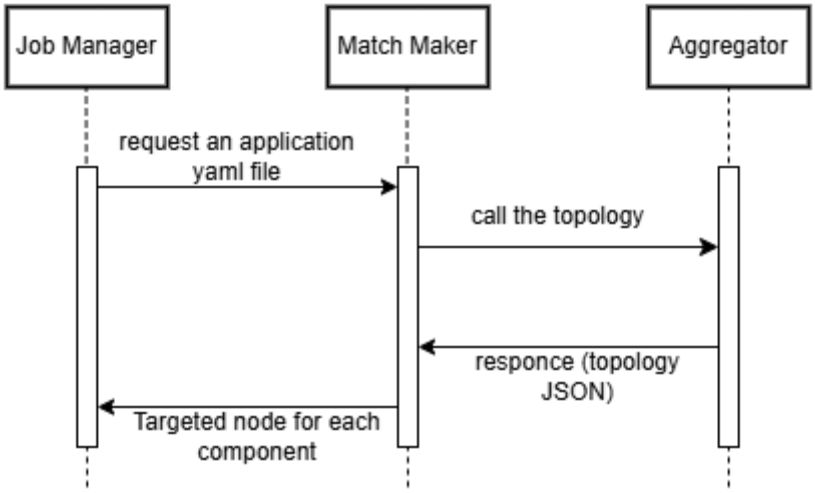

Figure 4 Matchmaking Interaction

The Matchmaker algorithm remains the same as described in D3.2 [4] with the following steps and an additional **rescheduling** functionality.

1. Filtering
2. Scoring
3. Affinity
4. SelectBest
5. Rescheduling

Matchmaking also considers the policies defined by users in the policies section of the ICOS application header. Matchmaking evaluates user demands, such as specifying preferences for performance (e.g., 50%), security (e.g., 30%), and energy consumption (e.g., 20%). This information is utilized by the scoring algorithm to prioritize node selection based on the user's specific preferences.

4.1.5.1 Testing and Validation

The matchmaking algorithm is tested using an **integration test** to validate the integration of different steps inside the matchmaking i.e., filtering, scoring, affinity, select-best and rescheduling.

The integration test combines multiple functionalities:

- Read the application descriptor (YAML) and convert it to JSON.
- Reading the topology data (JSON).
- Execute the matchmaking logic, producing accurate results for target node selection based on the requirements.

4.1.5.2 Code

The repository Match-Making of ICOS' public Github (https://github.com/icos-project/Match-Making) contains the implementation of the service and algorithm of the Match-maker. The component is released under the Apache license 2.0.

4.1.6 Policy Manager

As reported in the deliverable "D2.2 ICOS Architecture Design (IT-1)" [1], the role of Policy Manager (aka Dynamic Policy Manager) is the management of the policies (technical and business performance) and the detection and prediction of violations of such policies in the running application.

Mainly, Policy Manager interacts with:

- Monitoring and Logging component for the creation and sending of the policy (configured to the Monitoring and Logging component) and
- Job Manager: it notifies the Policy Manager each time an application undergoes actions such as creation, stopping, starting, updating, or deletion. Specifically, as reported in section 2.2.2.1 of the deliverable D3.2 [4] the integration of the policies is part of the functionalities of the Job Manager.

The Policy Manager component has been implemented during IT-2, according to the design previously described in D3.1 [3] and D3.2 [4].

No improvements or functionalities are implemented in the last period about the integration of the Policy Manager (PM) with the other ICOS components. The main achievements about the integration with the other ICOS components of this module are implemented in the IT-2 and provided in section 3.3.1 and section 3.3.2 of the deliverable "D3.2 Meta-Kernel Layer Module Developed (IT-2)" [4].

In the last period of the project, the focus of the activity concerns PM GUI. The implementation of the GUI started in the IT-2 and the first version of it has been provided in the section 3.3.3, GUI, "D3.2 Meta-Kernel Layer Module Developed (IT-2)" [4])

The scope of the PM GUI is to have a user-friendly access to the policies.

The main achievements for the PM GUI (Figure 5 and Figure 6) pertain to the following functionalities (more details are reported in the deliverable D5.3 [6]):

- Activate / Deactivate Policies

  A new feature allows logged-in users to activate or deactivate an existing policy directly from the GUI. This action is integrated with the Policy Manager backend and updates the policy status accordingly.

- Policy Deletion

  Users can now delete a policy from the GUI. A confirmation dialog ensures the user is aware of the permanent removal.

- Policy Update

  Users can edit the thresholds and the variables of an existing policy.

- Enhanced User Feedback

  Success and error notifications have been introduced to provide immediate visual feedback when creating, deleting, or modifying (activating/deactivating) policies.

- Refined Keycloak Integration

  Building upon the initial Keycloak integration, the GUI now clearly differentiates guest (read-only) and user (privileged) modes, ensuring only authorized users can modify or delete policies.

- Improved UI / UX

  Light/Dark mode toggle for better user experience under different lighting conditions. Tooltips on action icons (view, activate/deactivate, delete) for clarity.

- Cleaner Codebase & Configurability

  Streamlined code structure (custom hooks for notifications, enhanced error handling).

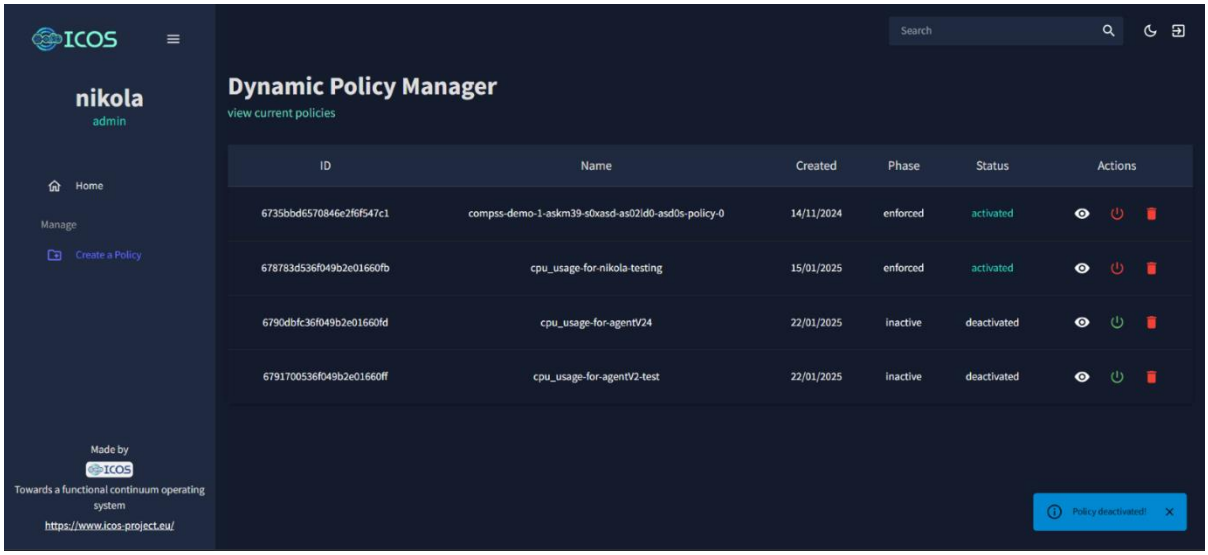

Figure 5: Dynamic Policy Manager GUI, home

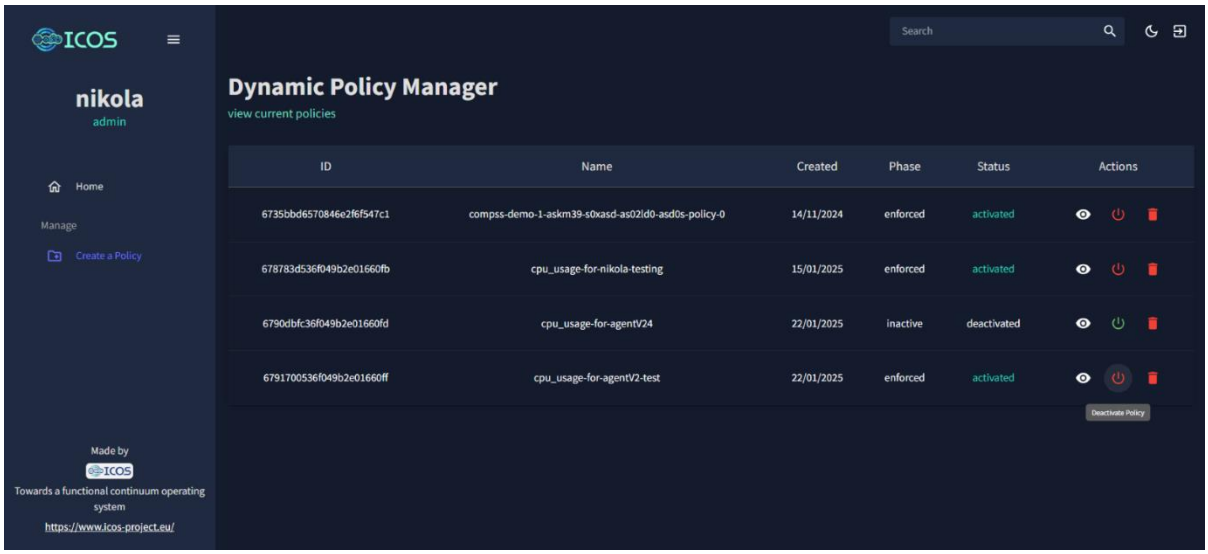

Figure 6: Dynamic Policy Manager GUI List of policies

4.1.6.1 API Endpoints

With respect to the previous reports, the only relevant changes in the API of the component are reported in Table 4. All URIs are relative to http://${CONTAINER_ADDRESS}.

Table 4 Policy Manager API updates

Endpoint	HTTP Method	Request Body	Description
/registry/api/v1/icos/app	POST	App descriptor and violation notifications endpoint	Starts monitoring a new application
/registry/api/v1/policies/{id}	PUT		Stops monitoring an application

4.1.6.2 Testing and Validation

The Policy Manager component has a CI/CD pipeline that includes multiple testing phases executed for each commit in the code base. The tests are of four types:

- code formatting and linting: checks code formatting and guidelines (e.g. Dockerfile and Helm file linting, secrets in the code);
- code quality: checks the quality of the code using SonarQube agents. This step assesses the overall quality of code in many different areas (e.g. readability, maintainability, security);
- Vulnerability Scanning of generated Docker Images using Trivy;
- Unit tests based on PyTest. Currently the test suite comprises 46 tests.

In addition, during the development manual REST APIs tests and integration tests are executed using the project's testbeds.

4.1.6.3 Code

The source code and tests for the Policy Manager are publicly accessible at the dynamic-policy-manager repository of the Github of ICOS: https://github.com/icos-project/dynamic-policy-manager.

4.1.7 Topology Exporter

The Topology Exporter is a new component incorporated to the architecture of the Meta-Kernel layer. Its purpose is to notify application-level containers about events related to the deployment of the application. Thus, application components can react to those events and change their configuration. This is especially needed for distributed computing libraries with no automatic resource discovery. An example of this kind of library is the distributed and parallel execution component. Its agent-based runtime system orchestrates the execution of many tasks on top of a distributed infrastructure; however, the neighbouring agents must be manually set up by the application deployer.

Rather than keeping an internal representation of the topology of each application and updating it upon being notified of the elements deployed or undeployed for each application, the Topology Exporter periodically obtains the topology of the system from the Aggregator. Upon receiving the response for the topology query, the Topology Exporter parses its content to discover the list of deployed elements corresponding to each component of each monitored instance of the application. When the whole

response has been parsed, the list of elements is published through a Zenoh[2] bus so that any application subscribed onto it can be notified about changes in its topology.

Instead of discovering applications and components during the parsing of the response and notifying all of them, the component does some filtering of this information and monitors only some specific applications. In order to start/stop monitoring an application, the Topology Exporter offers a REST API consisting of three methods:

4.1.7.1 API Endpoints

All URIs are relative to http://${CONTAINER_ADDRESS}

Table 5: Topology Exporter API Endpoints

Endpoint	HTTP Method	Request Body	Description
/	GET		Healthcheck test
/applications/	GET		List of monitored applications
/applications/{app_id}	PUT	Application Manifest	Starts monitoring a new application
/applications/{app_id}	DELETE		Stops monitoring an application

4.1.7.2 Testing and Validation

The Topology Exporter component contains several unit tests to verify its proper behavior. To facilitate the verification of the component, the other component of the Meta-Kernel layer on which the Topology Exporter depends (Aggregator) has been mocked and a new service that is able to return a pre-made topology has been developed. Upon a query for the status of the infrastructure, this mocked aggregator reads the desired status of the testbed from a file; therefore, by dynamically changing the content of the file, we can simulate the status of the infrastructure.

The unit tests for the component starts a docker compose that starts and creates several containers: 1) the mock Aggregator, 2) the Topology Exporter, 3) all the necessary containers to deploy Zenoh acting as the bus to report topology changes to the application, and 4) several containers emulating a deployed application. These application containers subscribe to the bus for notifications on a specific application deployment and component, and whatever notification is received is printed to the standard output.

The conducted test deploys this testing infrastructure and submits several requests to the Topology Exporter, so it starts monitoring several applications. Once the test has been set up, it continues by changing several times the file read by the mock Aggregator, and, on every change, it checks the logs of the application containers to verify that they received the desired notifications.

4.1.7.3 Code

The source code and tests for the Topology Exporter are publicly accessible at the dynamic-policy-manager repository of the Github of ICOS: https://github.com/icos-project/topology-exporter. The component is released under the Apache license 2.0.

[2] https://github.com/eclipse-zenoh/zenoh

4.1.8 Deployment Manager

The Deployment Manager collects jobs from the Job Manager and acts by deploying them when the status of the job is "executable". There are two different Deployment Managers depending on the target orchestration that the job is intended for: Open Cluster Manager (OCM) and Nuvla.

The OCM Deployment Manager acts over the executable jobs for the OCM orchestrator and uses the OCM resources to propagate deployments under a specific OCM orchestrator. For this it uses the resource Manifestwork[3] which uses standard Kubernetes resources. The Deployment Manager consists of one container with two different executables, one that periodically checks for synchronization and execution of jobs (sidecar) and another that actually implements the deployment with OCM wrapper functions.

The Nuvla Deployment Manager translates ICOS deployment jobs into application and deployment definitions that can be processed by the Nuvla orchestrator. Once the deployment is set to the START state, the NuvlaEdge Agent running on the edge node detects the new deployment and applies it within the local cluster.

4.1.8.1 API Endpoints

All URIs are relative to http://${CONTAINER_ADDRESS}

Table 6: Deployment Manager API endpoints

Endpoint	HTTP Method	Request Body	Description
/deploymanager	GET		Home route
/deploymanager/healthz	GET		Healthcheck test
/deploymanager/execute/	GET		Execute jobs
/deploymanager/status/	GET		Get Status
/deploymanager/status/sync	GET		Sync Status

4.1.8.2 Testing and Validation

A mock for the Job Manager has been created that helps implementing the appropriate methods for deploying OCM resources. The status sync feature involved several parts, so the validation was performed in a similar way locally by port forwarding some necessary ports from the staging testbed.

4.1.8.3 Code

The code implementing each flavor of the Deployment Manager is located in a different repository of the ICOS Github.

The Nuvla Deployment Manager is in the deployment-manager-nuvla repository: https://github.com/icos-project/deployment-manager-nuvla. The driver is released under the Apache license 2.0.

The code for the OCM driver and its sidecar container can be respectively found at the ocm-description-service (https://github.com/icos-project/ocm-description-service) and ocm-description-sidecar (https://github.com/icos-project/ocm-description-sidecar). Both ocm-driver components are released under the Apache license 2.0.

[3] https://open-cluster-management.io/docs/concepts/work-distribution/manifestwork/

4.1.9 ClusterLink

ClusterLink simplifies the connection between services that are located in different domains, networks, and cloud infrastructures. ClusterLink runs on Kubernetes-enabled clusters. The ClusterLink deployment operator allows easy deployment of ClusterLink to a Kubernetes cluster.

ClusterLink setup operations (create, deploy, delete) are performed during cluster deployment using the ClusterLink CLI command. Once the ClusterLink operator is deployed on a cluster, operations (peer, export, import, policies) are performed through Custom Resource Definition objects (CRDs). The creation and manipulation of CRDs is done through the Kubernetes API. ICOS components (e.g. Deployment Manager) identify which services need to be exported/imported in a cluster, and apply the appropriate CRDs to perform the export/import operations.

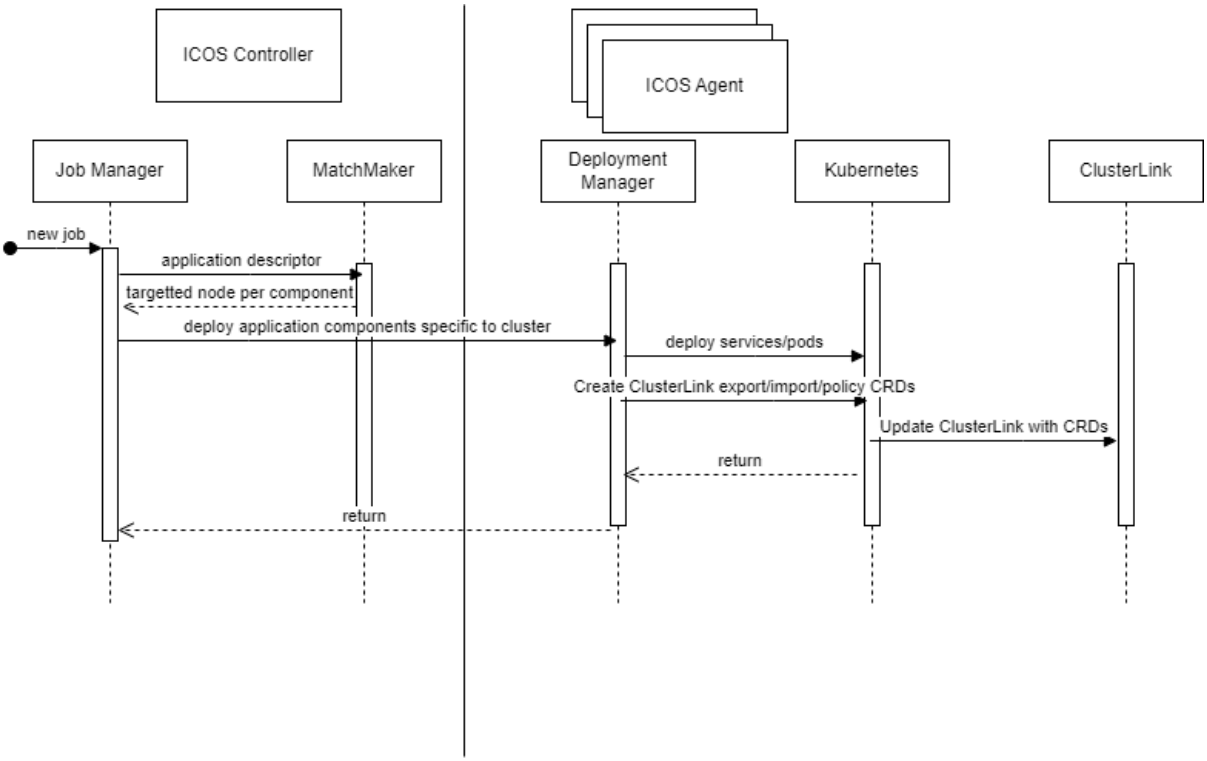

Figure 7 ClusterLink interaction

Details of the design of how ClusterLink works can be found in D3.2 [4] Section 3.1.1 and https://clusterlink.net/docs/latest/concepts/.

Examples of the CRD formats can be found at https://clusterlink.net/docs/main/tasks/operator/.

4.1.9.1 Testing and Validation

The ClusterLink project has a CI/CD pipeline that runs tests for each commit. The testing includes three types:

1. **Code Format**: This step checks for strict code formatting and runs linter checks to preserve the correct style, thus removing style warnings and errors in the project.
2. **Unit Tests**: ClusterLink's unit tests are based on the Go testing framework, enabling fast checks for specific units such as the ClusterLink operator, ClusterLink control plane controllers (authorization, XDS, CRDs), ClusterLink dataplane controllers, the policy agent, and various utilities used by ClusterLink components.

3. **End-to-End (E2E) Tests**: The ClusterLink project runs E2E tests to verify the correctness of the system. This framework uses Kind[4] and the testify/suite package, a testing library for Go, to test different end-to-end scenarios, such as CLI commands, ClusterLink operators, correct usage of CRDs, proper implementation of the policy agent for access control rules, and various load balancing scenarios. Additionally, the ClusterLink project performs performance tests to ensure that the local environment is not negatively impacted and can handle high-speed traffic.

4.1.9.2 Code

The code of the component is available at https://github.com/clusterlink-net/clusterlink. The component is released under the Apache license 2.0.

4.1.10 Orchestrator Edge Cloud

This component is external to ICOS. It acts as an aggregator and orchestrator service of the Cloud and Edge resources. It provides cloud and edge device management and application orchestration capabilities on top of the resources it aggregates. For more details see D3.1 [3] section 2.2.11. The project uses two different orchestrators: OCM and Nuvla.

Open cluster management (OCM) works as a centralized reference for Kubernetes resources using Manifestwork. With the internals of OCM, any edge cluster will pull resources and apply them when reading Manifestwork in the corresponding namespace. The OCM handshake warrantees that there are no leaks of important cluster information by utilizing Kubernetes RBAC mechanisms.

Nuvla translates ICOS deployment jobs into application and deployment definitions that can be processed by the Nuvla orchestrator. Once the deployment is set to the START state, the NuvlaEdge Agent running on the edge node detects the new deployment and applies it within the local cluster.

4.1.11 Distributed and Parallel Execution

The purpose of the component is to parallelize and distribute the workload of an application. Each application container runs a process (the D&PE Agent) providing a REST API that allows requesting the execution of a function or python script and managing the resources onto which its workload can be offloaded. The API of the component offered to application developers remains the same as described in D3.1 [3] Section 2.2.8.1.

Additionally, in order to automatically discover the resources onto which a D&PE Agent can offload tasks, the component has been extended with a sidecar process that subscribes to a specific topic on Zenoh. Through this subscription, this process expects to receive a description of the topology of the application with information of which pods have been deployed on each cluster. After receiving this information, this sidecar process checks the current configuration of the Agent and finds out which pods/services have been newly added to the deployment and which currently configured nodes have been removed from the deployment. Any of these changes is notified through the REST API of the D&PE Agent to reconfigure the pool of resources onto which the Agent can offload tasks to distribute the workload.

[4] https://kind.sigs.k8s.io/

4.1.11.1 Testing and Validation

The Distributed and Parallel Execution component undergoes several unit tests as already described in D3.1 [3] Section 4.2.3. Additionally, a new test has been added to verify that the component is able to react to the notifications arriving from the Zenoh bus. Using a docker compose, this test raises a container that pushes through the bus several different topologies. The test verifies that every time that a new topology is pushed through the bus, the resource pool of the D&PE agent has been modified accordingly.

4.1.11.2 Code

The code of the Distributed and Parallel Execution component is available in ICOS' dp-exec repository: https://github.com/icos-project/dp-exec. The component is released under the Apache license 2.0.

4.1.12 Aggregator

The Aggregator service provides a comprehensive knowledge of the resource infrastructure of the ICOS ecosystem. This information includes both static and dynamic properties such as available computing resources, performance, availability, eco-efficiency and peripheral hardware.

It is implemented as a web service written in Go language, exposing an API called by the Matchmaking service. The response of the Aggregator is given following the ICOS Infrastructure Taxonomy, describing the characteristics of the ICOS infrastructure. This infrastructure taxonomy, as outlined in D3.1 [3] Annex 6.1.5, has remained consistent with no significant changes since its initial description. To get this information, the service queries the Telemetruum Hub (Thanos) component to retrieve telemetry data and restructure it according to the ICOS taxonomy.

The interaction of these components is described as the following flow diagram:

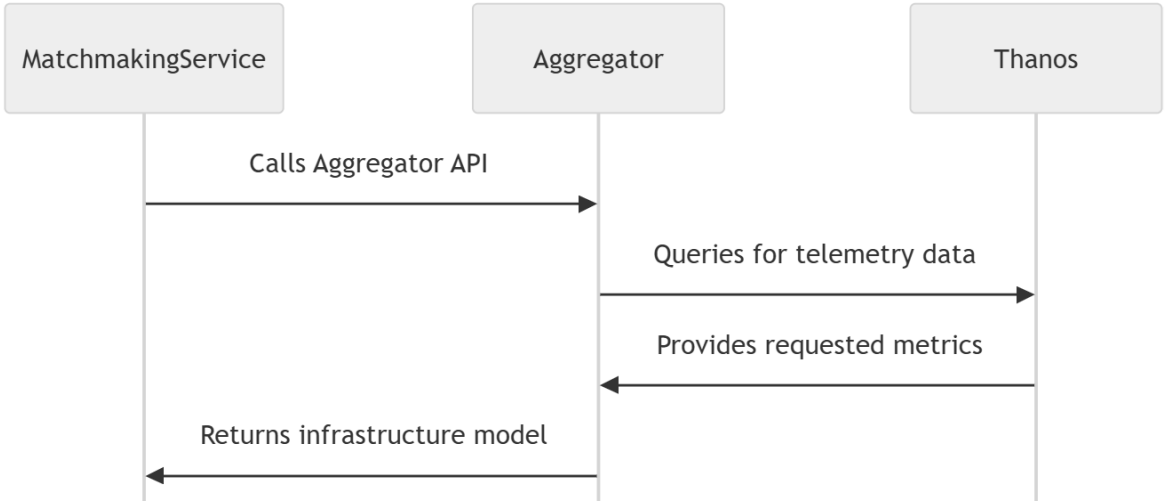

Figure 8 Aggregator interaction

In IT-2, the logging, debugging and error handling of the Aggregator were improved through a code refactoring that eases the development, deployment and maintenance process of the service.

More details about the Aggregator may be found in Deliverables D2.4 [2] and D3.1 [3].

4.1.12.1 Testing and Validation

The testing carried out for the component includes functional and integration testing to verify the correct functionalities of the service and its interactions with other ICOS components. Additionally, the output of the Aggregator was validated against the ICOS Testbed to ensure consistency with the actual state of the infrastructure.

4.1.12.2 Code

The repository Aggregator of ICOS' public Github (https://github.com/icos-project/Aggregator) contains the source code implementing the Aggregator. The component is released under the Apache license 2.0.

4.1.13 Telemetry Controller

The role of the telemetry controller is to store data and provide interfaces to query and visualise it; it runs in the ICOS Controller (see Figure 1). Further details about the implementation are provided in Section 2.1.16 of the deliverable D3.2 [4]. No significant changes have occurred since then.

4.1.13.1 Testing and Validation

The Telemetry module has a CI/CD pipeline that includes multiple testing phases executed for each commit in the code base. There are mainly three types of tests:

- code formatting and linting: checks code formatting and guidelines (e.g. Dockerfile and Helm file linting, secrets in the code);
- code quality: checks the quality of the code using the SonarQube agents. This step assesses the overall quality of code on many different areas (e.g. readability, maintainability, security);
- Vulnerability Scanning of generated Docker Images using Trivy.

In addition, during the development manual REST APIs tests and integration tests are executed using the project's testbeds. Unit testing is not extensively used for this component mainly because a) the majority of modules are composed from third-party software and b) given the extremely distributed nature of the component, the effort required to properly isolate single functionalities mocking other modules to create proper unit tests was judged not convenient for the project.

4.1.13.2 Code

The source code of the Telemetry Controller is available at ICOS' Telemetry-Controller repository: https://github.com/icos-project/Telemetry-Controller.

4.1.14 Telemetry Gateway

The Telemetry Gateway exposes an API to receive metrics from the Telemetry Agents using the OpenTelemetry Protocol (OTLP) format through HTTP and gPRC. The same protocols are used to send data to the Telemetry Controllers. The Telementry Gateway remained the same as in IT-2, with further details provided in Section 2.1.15, of the deliverable D3.2 [4].

4.1.14.1 Testing and Validation

Concerning testing, for the Telemetry Gateway the same type of tests described for the Telemetry Controller component in section 4.1.13.1 have been applied.

4.1.14.2 Code

The code of the component has been released in ICOS public repository (https://github.com/icos-project/telemetry-gateway). The component is released under the Apache license 2.0.

4.1.15 Telemetry Agent

The role of the Telemetry Agent is to collect data from the system and the applications and to send it to the Telemetry Controller; it runs in the infrastructure nodes.

The Telemetry Agent remains the same as in IT-1, D3.1 [3], with one key architectural change: the Telemetry Agent now sends data to the new Telemetry Gateway instead of directly to the Telemetry Controller. At the implementation level, additional metrics and plugins have been introduced in the IT-2, with further details provided in section 2.1.14 of the deliverable D3.2 [4].

4.1.15.1 Testing and Validation

Concerning testing, for the Telemetry Agent the same type of tests described for the Telemetry Controller component in section 4.1.13.1 have been applied.

4.1.15.2 Code

The code of the Telemetry Agent is published in the Telemetry-Agent (https://github.com/icos-project/Telemetry-Agent) and Telemetruum-Agent (https://github.com/icos-project/Telemetruum-Agent) repositories within ICOS' Github. The code of both parts is released under the Apache license 2.0.

4.1.16 Metrics Export API

The Metrics Export API extends ICOS with advanced monitoring and predictive analytics capabilities by integrating real-time telemetry data with machine learning-based forecasting. It acts as a bridge between telemetry sources (Prometheus, Thanos, Grafana) and the Intelligence Layer, enabling dynamic metric creation, predictive analytics, and model training. The API facilitates automated metric tracking, allowing system components to react proactively to workload variations and anomalies.

By leveraging the ICOS Aggregator and Intelligence Layer, the Metrics Export API supports static metric provisioning for system-wide monitoring and dynamic metric generation based on application-specific requirements. This ensures that infrastructure-level metrics (e.g., CPU, Memory, Energy consumption) and predictive models (e.g., load forecasting, anomaly detection) are seamlessly integrated into the ICOS ecosystem.

Authentication and access control are managed through Keycloak[5], ensuring secure interactions between components. More details on the architecture, API workflows, and integrations with ICOS telemetry and intelligence modules can be found in D4.2 [5].

4.2 Additional Comments on Integrated Solution

4.2.1 Job Management

The Job Manager component is the core module of the ICOS Controller. It is responsible for comprehensive runtime management, encompassing control, persistence, and coordination among ICOS components. This integration ensures that all job-related processes are efficiently orchestrated within the ICOS framework, supporting overall reliability and coherence.

The Job Manager comprehensive lifecycle management CRUD operations (Create, Read, Update, Delete) ensure the consistent and efficient execution of Jobs. These operations oversee the state of individual Jobs within the ICOS ecosystem, thereby maintaining system integrity and performance. By managing the Job lifecycle, the Job Manager ensures that Jobs transition smoothly through various states while adhering to the specified policies and requirements. This structured approach to job execution enhances the adaptability and responsiveness of the ICOS architecture, fostering a more integrated and automated operational environment.

The integration with the Policy Manager component is a critical aspect of the Job Manager's functionality. The Job Manager notifies the Policy Manager each time an application undergoes actions such as creation, stopping, starting, updating, or deletion. This notification mechanism ensures that all policy-related considerations are continuously monitored and enforced throughout the application lifecycle.

[5] https://www.keycloak.org/

Additionally, the Job Manager plays a key role in responding to policy non-compliance detected by the Policy Manager. Upon receiving a non-compliance request, the Job Manager initiates a remediation process by updating the job's status with the necessary metadata required by the Deployment Manager to take appropriate action. These actions may include modifying job parameters, triggering scaling operations, applying security measures, or reallocating deployments to optimize system performance. To facilitate communication, the Job Manager provides a callback response URL in case of non-compliance, ensuring prompt and structured handling of policy violations.

By actively responding to policy violations, the Job Manager ensures compliance across the ICOS environment, strengthening its resilience and adherence to predefined operational standards.

4.2.1.1 Enforcement of policies

The orchestration of an application is adapted based on system conditions and events using pre-defined policies. This integration involves the Job Manager, Dynamic Policy Manager, and the Logging and Telemetry subsystem. When a new application is deployed, the Job Manager calls the Dynamic Policy Manager, which parses the Application Descriptor and activates the relevant policies based on data collected at the edge. It also sets up alerting rules to notify when the data does not meet the policy conditions. During runtime, the Dynamic Policy Manager receives notifications if metrics from the edge application violate the established rules. If a violation occurs, it informs the Job Manager, which then takes actions (e.g., migrating the application or scaling it) to restore compliance with the policies.

The enforcement of policies has been implemented during IT-2. During the last period no improvements or functionalities are implemented; the details of this component are described in Section 2.2.6, Enforcement Policies, of the deliverable D3.2 [4].

4.2.1.2 Component rescheduling

After the successful deployment of the application by the Job Manager through Deployment Manager, the Policy Manager monitors the current targeted nodes for policy violations (e.g., a node's security score violation). If a violation is detected, this information (component with node name) is sent to the Job Manager. The Job Manager notifies the Matchmaker about the execution of rescheduling (migration of component to feasible node), providing the following details:

- Current setup of the deployed application components.
- Details of the violated components and nodes.
- The original application Yaml file.

The Matchmaker then reschedules the targeted node based on the updated information from the Job Manager. The rescheduling loop, as described in the flow diagram Figure 9, covers the process from initial deployment to rescheduling.

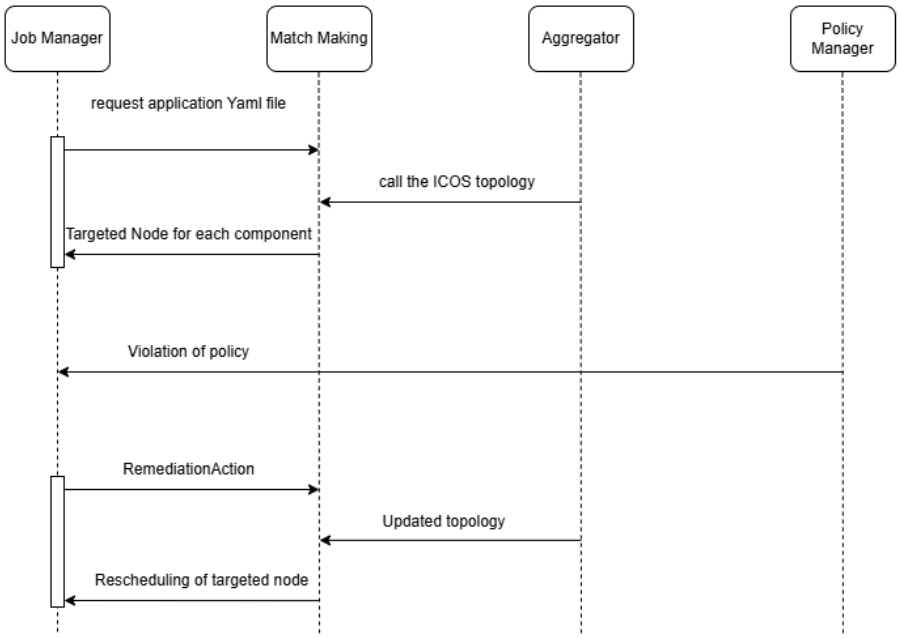

Figure 9 Rescheduling loop

4.2.2 Job Deployment

The Deployment Manager in the ICOS project plays a central role in managing the deployment of user applications within the edge-to-cloud continuum. It is responsible for interfacing with the Job Manager to retrieve and execute deployment jobs based on predefined rules and conditions. Unlike other components in the system, the deployment manager does not expose any HTTP endpoints but rather operates in a pull-based mode, consuming jobs from the Job Manager interface.

Once a job is retrieved, the Deployment Manager processes and validates it before executing it within the Orchestrator Edge Cloud. The job must meet specific state and locking conditions for execution, ensuring synchronization and consistency within the ICOS deployment workflow. This approach enables dynamic and automated deployment orchestration across heterogeneous infrastructures, supporting various technologies and deployment environments.

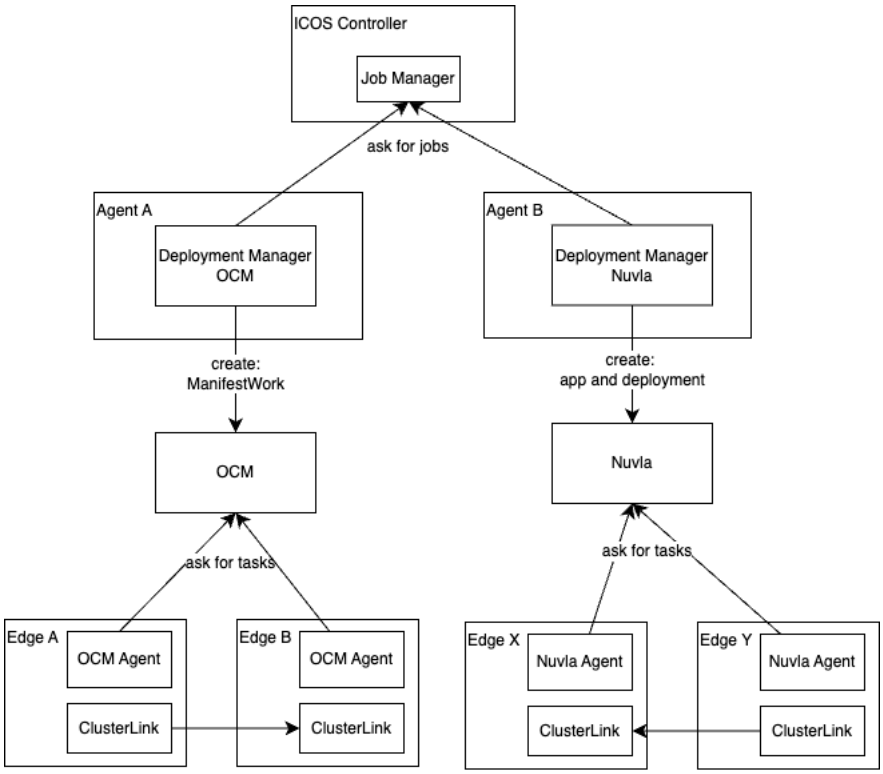

Figure 10 Deployment Manager interactions

4.2.2.1 Interaction with underlying orchestrators

The Deployment Manager interacts with underlying orchestrators such as Open Cluster Management (OCM) and Nuvla, which are responsible for managing Docker and Kubernetes-based clusters and edge/cloud resources. It works by pulling deployment jobs from the Job Manager and then depending on the underlying orchestrator type (OCM or Nuvla), the Deployment Manager:

OCM:

translates ICOS deployment jobs into ManifestWork objects can be processed by the OCM orchestrator. Once the ManifestWork is created, the Open Cluster Management Agent running on the edge node detects the new deployment and applies it within the local cluster.

Nuvla:

translates ICOS deployment jobs into application and deployment definitions that can be processed by the Nuvla orchestrator. Once the deployment is set to the START state, the NuvlaEdge Agent running on the edge node detects the new deployment and applies it within the local cluster.

This mechanism ensures that deployments are effectively managed and monitored in distributed environments. The interaction with these orchestrators allows ICOS to support scalable and flexible application deployments across various computing environments. This enables a seamless execution on both cloud and edge resources. The ICOS Deployment Managers for OCM and Nuvla are written in Golang and Python correspondingly.

4.2.2.2 Multi-Cluster Setups

In the cases where applications running on different clusters (typically in the edge-to-edge scenario) need to connect to each other, the continuum should be able to provide the corresponding overlay networking options as well as the applications discovery possibilities. In the project this is achieved using ClusterLink developed by IBM. The solution works on Kubernetes only.

The action of setting up of the cluster-to-cluster connections using ClusterLink is done during the edge onboarding process. This includes the decision on how clusters could connect on the network level (e.g., one of the clusters can reach the other via IP:PORT; a relay can be used (e.g., VPN), etc.) and then deploying the ClusterLink components on each of the clusters and letting them establish the peer connections. After the multi-cluster setup is done, the deployment of the applications by the Deployment Managers takes into account the pre-created cluster mesh and works on the level of the exporting/importing the corresponding services required by the applications. For the details on the ClusterLink deployment and operation see section 4.1.9.

4.2.3 Logging and Telemetry

The main achievements are implemented in the IT-2 and provided in the Section 2.2.5 of the deliverable D3.2 [4]. Specifically, the Figure 10 of the deliverable D3.2 [4] and provided below (Figure 11) shows the main technologies integrated to realize the telemetry components:

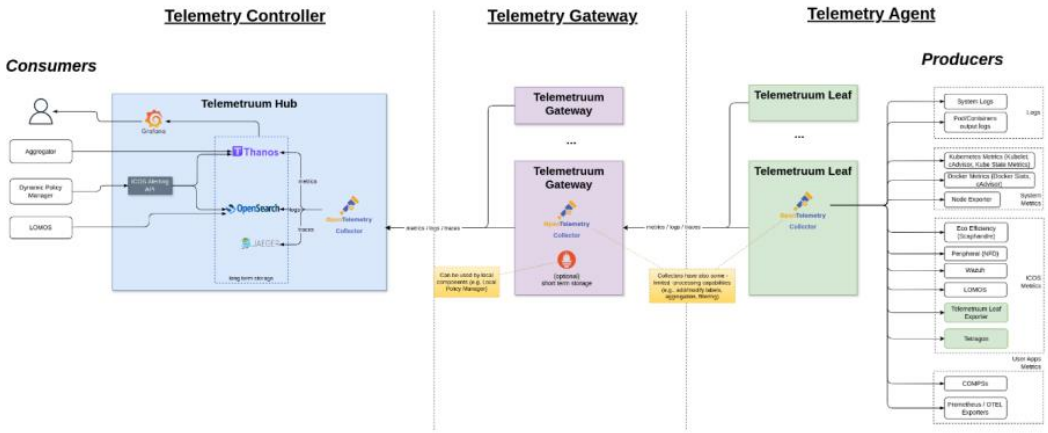

Figure 11 Logging and Telemetry technologies

During the last period, the Telemetruum[6] Leaf Exporter component has been updated to improve the stability and report additional data (e.g., node geographical location, custom labels, workloads statuses).

[6] Telementruum Leaf is the software module that implements the Telemetry Agent architectural component: https://www.icos-project.eu/docs/Concepts/Functionalities/observability/.

4.2.4 Topology changes notification to the Application

The sequence diagram in Figure 12 illustrates how the ICOS Meta-Kernel layer notifies applications so they can react and be reconfigured automatically when there is any change on the deployment of the application.

Distributed applications may span across multiple clusters, either under the domain of one single controller or many. Although we could implement a protocol to share this information directly among ICOS Agents, potential race conditions during the deployment of the containers may complicate the solution. A simpler alternative to this coherence protocol is to control this information at the Controller level.

Within the Controller, there are two potential sources from where to obtain this information. The first option is that every time that the Job Manager orders the deployment of some component, it notifies the change to the application. However, if there's any error during the deployment, the information may be inconsistent. The second source of information that can be used is the Aggregator which gathers all the information collected through the telemetry infrastructure. Despite the delay for gathering the data, if the information is in the Aggregator, the components have already been deployed. Our solution leverages the information from the Aggregator.

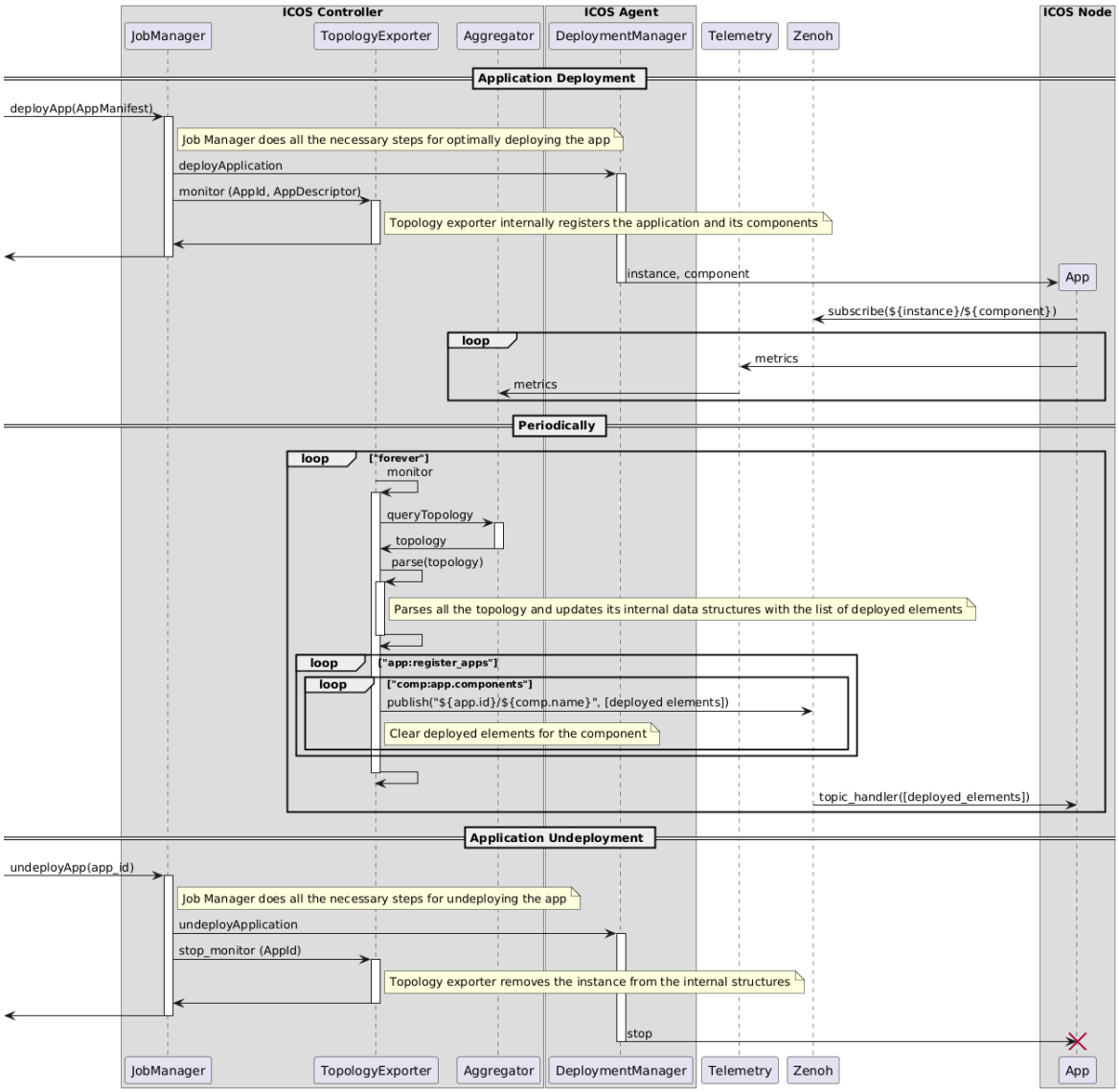

Figure 12 Sequence diagram for notifying topology changes to application

The Aggregator is a reactive component; it collects the data and responds to queries done by other components; therefore, it won't be aware of any changes in application deployments. In order to overcome this lack of proactiveness of the Aggregator a new component has been added to the ICOS Controller: the Topology Exporter. This component queries the application topology from the Aggregator and notifies the application about its current configuration.

To reduce the number of applications whose topology is being monitored, the Job Manager lets the Topology Exporter know when a new application is being deployed so it starts monitoring. Likewise, when the user requests to stop an application, it also notifies the Topology Exporter, so it stops monitoring the application deployment.

For exposing metaOS level information at application level, we created a data bus where the Topology Exporter publishes the deployed elements. Not all the applications require to know the topology of its deployment. Thus, it makes sense that application changes are only notified on a subscription basis. When a container of an application requiring to know that information starts, it can subscribe to a specific topic on this bus to receive the notifications.

5 Conclusions

The ICOS project successfully concluded the development and integration of the ICOS Meta-Kernel Layer module. The code, which is ready for integration into Work Package 5 (WP5), has been publicly released on the project's official GitHub repository (https://github.com/icos-project) under open-source and non-viral license.

By leveraging the architectural groundwork established in prior deliverables — D2.2 [1], D2.4 [2] —, this report provides a structured analysis of the module's evolution in alignment with the system's objectives.

The report introduces a comprehensive specification of the Application Manifest, a critical feature that enables vertical application owners to harness the full capabilities of ICOS efficiently. The Application Manifest facilitates seamless interaction with the Meta-Kernel Layer, offering developers a standardized mechanism to deploy and optimize their applications within the ICOS framework.

The document also describes significant updates to components compared to the progress documented in previous iterations — D3.1 [3], and D3.2 [4]. These updates encompass enhancements in performance optimization, stability improvements, and refinements to the integration mechanisms, ensuring better alignment with system-wide requirements. To support further development and collaboration, the report also includes a reference to the official ICOS project repository, where the latest version of the module's source code is made available and the license describing the terms and conditions for use, reproduction, and distribution of the component. Additionally, this document provides an in-sight of the necessary updates and design refinements to ensure a smooth integration of the whole Meta-Kernel Layer module.

All components have been implemented, tested, integrated and deployed on the ICOS testbed. Results of running on the integrated system are to be reported in deliverables "D5.3 Third ICOS Release: Complete ICOS version" [6] and "D6.11 Final ICOS Product release" [7].

Efforts will continue beyond this final iteration to further refine the Meta-Kernel Layer Module by:

- addressing any remaining bugs
- optimizing performance and improving resource consumption of each individual component
- enhancing the reliability and stability of the integration ensuring seamless operation within the fully integrated Meta-Kernel.

As discussed in Section 2, the implementation of inter-controller communication, will not be addressed during the lifetime of this project. While the current focus has been on refining and integrating the Meta-Kernel Layer Module, the development of mechanisms to facilitate communication between different controllers within the system has been scoped out for future work. The implementation of inter-controller communication will be considered as a key area for future development, building upon the solid foundation laid by the current module and its integration within the ICOS framework.

6 References

[1] ICOS D2.2 – *"ICOS Architecture Design (IT-1)"*. Giammatteo, Gabriele. May 2023 https://www.icos-project.eu/files/deliverables/D2.2_ICOS_Design_v1.0.pdf,

[2] ICOS D2.4 – *"ICOS Architectural Design (IT-2)".* García, Jordi. July 2024 https://www.icos-project.eu/files/deliverables/D2.4-architectural-design-it2.pdf

[3] ICOS D3.1 – *"Meta-Kernel Layer Module Integrated (IT-1)".* Skaburskas, Konstantin. October 2023. https://www.icos-project.eu/files/deliverables/D3.1_Meta_Kernel_Module_IT-1_v1.0.pdf

[4] ICOS D3.2 – *"Meta-Kernel Layer Module Developed (IT-2)"*. Lordan, Francesc. October 2024 https://www.icos-project.eu/files/deliverables/D3.2-Meta-Kernel_Layer_Module_Developed_IT-2_v1.0.pdf

[5] ICOS D4.2 – *"Data Management, Intelligence and Security Layers (IT-2)"*, In preparation.

[6] ICOS D5.3 – *"Third ICOS Release: Complete ICOS version"*, In preparation.

[7] ICOS D6.11 – *"Final ICOS Product release"*, In preparation.