D.3.2 Report on the application LCM in the CECC – Final

Document Summary Information

Project Identifier	HORIZON-CL4-2022-DATA-01. Project 101093129		
Project name	Agile and Cognitive Cloud-edge Continuum management - Initial		
Acronym	AC^3		
Start Date	January 1, 2023	End Date	December 31, 2025
Project URL	www.ac3-project.eu		
Deliverable	D.3.2 Report on the application LCM in the CECC – Final		
Work Package	WP3		
Contractual due date	30/06/2025	Actual submission date	XX/XX/XX
Type	R — Document, report	Dissemination Level	PU
Lead Beneficiary	FIN		
Responsible Authors	Abdelhak KADOUMA, A. RASHEED, and Ibrahim AFOLABI (FIN)		
Contributors	A. Kadouma, A. Bakare, O. Oladeji, A. Rasheed, I. Afolabi (FIN), D. Klonidis, A. Valantasis (UBI), A. MELIANI M. Mekki (EUR), M. Gurdiel, K. Ramantas (IQU), E. Dritsas (ISI), D. Amaxilatis, N. Tsironis (SPA), A. Podimata, G. Koulopoulos, V. Moulos (UPR)		
Peer reviewer(s)	V. Moulos (UPR), Souvik Sengupta (IONOS)		

AC^3 project has received funding from European Union's Horizon Europe research and innovation programme under Grand Agreement No 101093129.

Revision history (including peer reviewing & quality control)

Version	Issue Date	% Complete	Changes	Contributor(s)
V0.1	13/03/2025	0%	Initial Deliverable Structure and ToC	A. Kadouma, A. Rasheed, I. Afolabi (FIN), D. Klonidis (UBI)
V0.2	26/03/2025	10%	Final Deliverable Structure	A. Kadouma, A. Rasheed, I. Afolabi (FIN), D. Klonidis (UBI),
V0.3	31/03/2025	20%	Editing of Section 3	A. Kadouma (FIN), A. Rasheed, I. Afolabi, A. Bakare
V0.4	01/04/2025	24%	Restructuring Chapters	D. Klonidis (UBI)
V0.5	10/04/2025	30%	Editing Section 4	A. Kadouma (FIN), A. Rasheed, I. Afolabi
V0.6	14/04/2025	35%	Updating Sections 3 and 4	A. Kadouma (FIN), D. Klonidis (UBI)
V0.7	23/04/2025	45%	Editing Section 2 and 5	A.meliani (EUR), D,Klonidis (UBI)
V0.8	24/04/2025	50%	Editing Section 5	A.meliani (EUR),
V0.9	02/05/2025	60%	Editing Section 5	J, Redondo (IQU)
V0.9.1	20/05/2025	70%	Updating Section 2	D.klonidis (UBI), A.Kadouma (FIN)
V0.9.2	26/05/2025	75%	Editing Section 4	A.Kadouma (FIN), A. Rasheed (FIN), I. Afolabi (FIN)
V0.9.3	28/05/2025	80%	Editing Section 5	A.Meliani (EUR), J.Redondo (IQU)
V0.9.5	02/06/2025	85%	Editing Section 3	M.Vrettos (UPR),
V0.9.7	05/06/2025	90%	Editing Section 1, 2, 3, & 6	M.Vrettos (UPR), D.klonidis (UBI), A.kadouma (FIN), I.koulopoulos (UPR
V0.9.9	10/06/2025	95%	Proofreading	I. Afolabi (FIN), A. Kadouma (FIN), A. Rasheed (FIN), A. Bakake (FIN), O. Oladeji (FIN), D. Klonidis (UBI)
V1.0	11/06/2025	100%	Final draft sent for internal review	A.Kaouma (FIN), A. Rasheed (FIN), I. Afolabi (FIN), A. Bakare (FIN), O. Oladeji (FIN), D.Klonidis (UBI)

© AC^3 2023

Disclaimer

The content of this document reflects only the author's view. Neither the European Commission nor the HaDEA are responsible for any use that may be made of the information it contains.

While the information contained in the documents is believed to be accurate, the authors(s) or any other participant in the AC^3 consortium make no warranty of any kind with regard to this material including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Neither the AC^3 consortium nor any of its members, their officers, employees or agents shall be responsible for or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein.

Without derogating from the generality of the foregoing neither the AC3 Consortium nor any of its members, their officers, employees or agents shall be liable for any direct or indirect or consequential loss or damage caused by or arising from any information advice or inaccuracy or omission herein.

Copyright message

© AC^3 Consortium. This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.

Table of Contents

1 Executive Summary 11
2 Introduction 12
2.1 Overview – purpose and objectives 12
2.2 Mapping AC3 Outputs 12
2.3 Integration with other initiatives within the AC3 Project 15
2.4 Deliverable Overview and Report Structure 15
3 Implemented application LCM functionalities and mapping in the AC3 platform 17
3.1 Overview of the application-related innovative processes in the AC3 CECC 17
3.2 Mapping to the updated AC3 architecture 17
4 Intent-based models, mechanisms and modules for application deployment 20
4.1 Introduction 20
4.2 Overview of the design and implementation plan 20
4.3 Application Descriptor Model 21
4.4 Formulation of the application descriptor from the GUI 23
4.5 KPIs collection and presentation 25
4.6 Adaptation layer for LCM 30
5 Application profile model mechanism 33
5.2 Introduction 33
5.3 Overview of the design and implementation plan 33
5.4 Related works 34
5.5 Application Profile Model 35
5.5.1 Application Metadata 36
5.5.2 Application Consumers 36
5.5.3 Microservices and Data Sources 36
5.5.4 Service Level Agreements (SLAs) 36
5.6 Data Collection 37
5.7 Prediction Algorithm 38
5.8 Testing and results 40
5.9 Future Perspectives: Enhancing APM with Generative AI and Log Analytics 42

6 Service migration mechanisms 44
6.1 Resiliency Focused Proactive migration for Stateful Microservices in Multi-Cluster Containerized Environments 44
6.1.1 Related work 44
6.1.2 Solution 44
6.1.3 Resource Discovery and Resource Exposer 45
6.1.4 Resource Orchestrator (RO) (AI-Based LCM) 45
6.1.5 Cluster Manager (LMS) 46
6.1.6 Predictor 46
6.1.7 Checkpointer Operator 46
6.1.8 Workflow 46
6.1.9 Testing and results 49
6.1.10 Before the migration 50
6.1.11 During the Migration 53
6.2 Reinforcement Learning Cache-aided Service Migration Algorithm 58
6.2.1 Introduction 58
6.2.2 Migration cache-aided architecture 59
6.2.3 Workflow 60
6.2.4 Performance evaluation 61
7 Conclusions 65
8 References 66

© AC3 2023

List of Figures

Figure 1: Mapping of Application related processes to the overall AC3 architecture including also the directly linked processes 18

Figure 2: High-level structure of the application descriptor 21

Figure 3: Example configuration of a frontend microservice, including general details, environment variables, microservice-specific SLAs, and deployment constraints 22

Figure 4: Sample of the networking graph from the descriptor, detailing the connection dependencies, protocols, ports, and SLAs for each link between microservices 23

Figure 5: Translating High-Level Applications Specifications Workflow 24

Figure 6: Architecture Design workflow 27

Figure 7: Rule configuration example 30

Figure 8: mapping between one of the use case 2 microservices configuration in the App Descriptor to its equivalent in the NSD format 31

Figure 9: Example of the application translator mapping for use case 2. Adaptation between AC3 application descriptor and the MAESTRO service order as a custom manifest work 32

Figure 10: Holistic Representation of the Application Profile Model 35

Figure 11: APM Data collection 37

Figure 12: R2 Scores for each target for each tested algorithm 41

Figure 13: Stateful applications LCM Architecture 45

Figure 14: System components workflow 48

Figure 15a: Model's MSE, RMSE, MAE, and MAPE across Different Window Sizes for the memory predictors models 51

Figure 16b: Model's MSE, RMSE, MAE, and MAPE across Different Window Sizes for the cpu predictors models 52

Figure 17: Average Inference Time (Seconds) for both resource models 53

Figure 18: Average time, required to complete checkpointing, image creation, and image pushing processes for different container sizes and container images on the source computing nodes in both migration scenarios 55

Figure 19: Multi-Clusters Scheduling techniques comparison 57

Figure 20: Application restore times with varying disk configuration 58

Figure 21: Migration cache-aided architecture 59

Figure 22: Migration process workflow 60

Figure 23: Network Environment 61

Figure 24: Communication Diagram of the different platforms 61

© AC3 2023

Figure 25 Latency comparison between different migration policies 63
Figure 26: Migration comparison between different migration policies 63
Figure 27: Average number of apps per edge server. 64

List of Tables

Table 1: Adherence to AC3 GA deliverable and tasks descriptions 12
Table 2: List of the KPIs provided 28
Table 3: Regressions algorithms Pros and Cons 39
Table 4: Disk performance metrics 49
Table 5: Model's performance metrics 53
Table 6: Simulation Setup for Migration Environment 62

Glossary of terms and abbreviations used

Abbreviation / Term	Description
AC^3	Agile and Cognitive Cloud edge Continuum management
AES	Advanced Encryption Standard
AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARIMA	Autoregressive Integrated Moving Average
AppD	Application Descriptor
CA	Certificate Authority
CECC	Cloud Edge Computing Continuum
CECCM	Cloud Edge Computing Continuum Manager
CI/CD	Continuous Integration (CI) and Continuous Delivery (CD)
CPU	Central Processing Unit
CRUD	Create, Read, Update, and Delete
DCAT-AP	DCAT Application profile for data portals in Europe
DDoS	Distributed Denial of Service
DQN	Deep Q-Networks
ETSI	European Telecommunications Standards Institute
GUI	Graphical User Interface
GXFS	Gaia-X Federation Services
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IDS	International Data Spaces
IDSA	International Data Spaces Association
IoT	Internet of Things
JSON	JavaScript Object Notation

© AC^3 2023

LCM	Life Cycle Management
LSTM	Long short-term memory
ML	Machine Learning
MQTT	MQ Telemetry Transport
PaaS	Platform as a Service
QoS	Quality of Service
RBAC	Role-Based Access Control
SLA	Service Level Agreement
OWL	Web Ontology Language
WAF	Web Application Firewall
YAML	Yet Another Markup Language

1 Executive Summary

The transition toward Cloud–Edge Continuum Computing (CECC) requires a fundamental shift in how applications are developed, deployed, and managed across diverse and distributed infrastructures. This shift calls for new models and tools capable of accurately interpreting user intent, converting Service Level Agreements (SLAs) into practical policies, and dynamically managing microservices with minimal human involvement. The AC³ project addresses this need by developing a CECC Management (CECCM) framework designed to automate and optimize application deployment, resource allocation, and lifecycle management (LCM).

The objective of Deliverable D3.2 is to present the implementation and integration of models, mechanisms, and tools that enable SLA-aware, intent-driven deployment and lifecycle management of microservices-based applications across the Cloud–Edge Computing Continuum (CECC). These solutions build upon the conceptual foundations introduced in Deliverable D3.1 ("Report on Application Life Cycle Management in the CECC – Initial"), moving from design concepts to validated, deployable implementations.

Work Package 3 (WP3) contributes to the AC3 CECC Management (CECCM) framework by developing the core architectural elements that enable seamless application modeling, profiling, and migration. This deliverable reports on:

- **Refinement of the Descriptor Model** (Task T3.1) – Expanding the initial design to support comprehensive application metadata, consumer profiles, microservice definitions, data sources, and deployment context, with integrated reasoning via the Ontology and Semantic Reasoner (OSR).
- **Implementation of the Application Profile Model (APM)** (Task T3.2) – Defining a unified, ontology-based model for static and dynamic application profiling, supported by AI/ML algorithms for workload prediction, SLA compliance monitoring, and proactive resource management.
- **Development of Migration Strategies** (Task T3.3) – Creating algorithms for migrating stateful microservices while ensuring data integrity, SLA adherence, and optimal placement across cloud–edge–far edge infrastructures.
- **Integration with CECCM Orchestrators** – Enabling descriptor translation for deployment through both the NFVO-based LiSO orchestrator and the MAESTRO orchestrator (as described in D2.3), ensuring compatibility and interoperability across heterogeneous infrastructure management systems.
- **Validation and Experimentation** – Presenting simulation and testbed results demonstrating the effectiveness of the models and mechanisms, with metrics on resource efficiency, SLA adherence, and adaptability under varying workloads.
- **Collaboration with WP5** – Ensuring integration of WP3 outputs into the broader CECCM interoperability layer, facilitating end-to-end lifecycle management across the AC³ architecture.

By combining these elements, D3.2 delivers a complete and validated toolkit for autonomous, SLA-compliant application lifecycle management in federated cloud–edge environments, paving the way for scalable, resilient, and sustainable CECC deployments.

2 Introduction

This section provides an overview of the work presented in this deliverable, outlining its primary purpose, key objectives, and connections to related project activities and deliverables. It highlights how this deliverable builds upon and continues the work previously described in deliverable D3.1, "Report on the Application LCM in the CECC – Initial," and outlines the expected outcomes mapped to specific tasks defined in the Grant Agreement (GA).

2.1 Overview – purpose and objectives

Deliverable D3.2 aims to finalize and validate the Life Cycle Management (LCM) mechanisms and tools for microservices-based applications within the Cloud–Edge Computing Continuum (CECC). While these mechanisms were initially introduced in Deliverable D3.1: "Report on Application Life Cycle Management in the CECC – Initial," this deliverable extends and concretizes those concepts through detailed implementation, simulation, and validation. In particular, D3.2 introduces new contributions in the form of a refined ontology-based Application Profile Model, the development and evaluation of machine learning–based profiling techniques, the implementation of intent-to-descriptor translation workflows, and advanced strategies for stateful service migration. These developments were not only conceptualized but also technically realized and evaluated, marking a significant step forward from D3.1. Thus, D3.2 serves both as a continuation and as a demonstration of novel results supporting the AC3 project's vision for intelligent, adaptive, and SLA-compliant application management in CECC environments.

Specifically, D3.2 addresses three primary tasks outlined in the GA:

- Task 3.1 encompasses the definition of Service Level Agreement (SLA) models and intent-based frameworks for automated microservices-based application deployment. It includes translating performance intents into executable policies and strategies and extending the application composition model to integrate comprehensive data management information.
- Task 3.2 focuses on developing Application Profile Models to effectively manage the lifecycle of microservices applications. This involves defining and constructing detailed application profiles, collecting and monitoring relevant profiling information, and employing Machine Learning (ML) algorithms to predict application behavior dynamically based on gathered monitoring data.
- Task 3.3 addresses challenges related to the migration of stateful microservices, developing algorithms designed to achieve optimal trade-offs between maintaining application SLAs and efficiently utilizing infrastructure resources. This ensures uninterrupted service continuity and optimized resource allocation in the dynamic CECC environment.

2.2 Mapping AC^3 Outputs

The purpose of this section is to map AC^3 Grant Agreement commitments, both within the formal Deliverable and Task description, against the project's respective outputs and work performed as shown in Table 1: Adherence to AC3 GA deliverable and tasks descriptions.

Table 1: Adherence to AC3 GA deliverable and tasks descriptions

AC^3 GA Component Title	AC^3 GA Component Outline	Respective Document Chapter(s)	Justification

DELIVERABLE			
D3.2 Report on the application LCM in the CECC – Final			
TASKS	**Description**	**Section in the deliverable**	**Justification**
T3.1 SLA and intent-based models for microservice-based applications deployment	SLA model definition: Revisiting and updating SLA models specifically for microservice-based applications within federated CECC environments, incorporating critical parameters like CPU, RAM, throughput, latency, service availability, and response time. Definition of microservice-based applications for automatic deployment: Updating application definitions to explicitly support automated deployment across cloud, edge, and far-edge locations, considering specific latency and regional proximity requirements. Translation of performance requirements/intents into corresponding policies/strategies: Developing an intent-based provisioning module that translates user-defined performance requirements (KPIs) into actionable resource allocation policies, allowing users to prioritize performance intents transparently. Extension of the application composition model to include information on data management: Enhancing existing application composition models, incorporate comprehensive data management	Section 3	Presented the components for the user place as well as the Application & Resource Management of the AC3 Framework architecture.

	details, supporting the specific needs of microservice-based deployments across the federated CECC.		
T3.2 Application profile models	Definition and construction of application profiles for managing application lifecycle: Identifying and defining key profile elements, including application consumers, constituent microservices, and relevant data sources, addressing aspects such as traffic types, generation patterns, and eligibility for far-edge deployment. Machine learning algorithms for predicting application profiles: Implementing explainable ML models to predict future application behavior based on monitored data, emphasizing a balance between model accuracy and explainability to gain actionable insights into application characteristics.	Section 4	Have a unified model that caters to LCM functionalities and ML Algorithms to predict application behavior.
T3.3 Service migration of stateful microservices	Algorithms for stateful microservice migration: Developing algorithms specifically designed to handle migration challenges of microservices requiring persistent storage or volumes, ensuring seamless continuity of service during lifecycle management operations. Trade-off between application SLA and infrastructure resources: Implementing decision-making strategies within migration algorithms to balance application performance constraints (such as latency) with the effective	Section 5	Present techniques that will assist LCM in place management.

	utilization of infrastructure resources at edge and far-edge nodes, informed by detailed application profiling. Efficient microservice image management: Utilizing machine learning and AI-based methods to predict optimal timing for transferring microservice images to edge locations, enabling efficient resource management and ensuring minimal service downtime during migrations.		
T3.1, T3.2, T3.3 with WP5 perspective	(Extensions related to overall LCM functionality and module interfacing)	Section 2, 3, 4 and Section 5	Solutions that can be integrated to the LCM

2.3 Integration with other initiatives within the AC3 Project

The work reported in this deliverable is closely interconnected with other AC^3 project activities, particularly those in Work Packages WP2, WP4, and WP5. WP2 provides the overarching architecture framework that underpins the LCM mechanisms described here. Meanwhile, WP4 contributes critical resource management functionalities, including underlying monitoring and resource discovery mechanisms that support the profiling and prediction models explained in this deliverable. Last but not least, WP5 offers essential feedback and integration validation, particularly guiding the enhancements made during this final phase, based on insights from Task T5.1.

2.4 Deliverable Overview and Report Structure

In this section, we outline the structure of Deliverable D3.2, providing a clear description of the contents of each part. Section 2 describes the LCM process within the overall AC^3 architecture, highlighting its core functionalities and their integration into the broader AC^3 framework. Section 3 introduces the application descriptor composition mechanism, detailing the methods used to translate high-level, user-defined application intents into deployable descriptors, along with the components involved in the semantic process.

Section 4 presents the AI-based application profiling mechanism. It explains the process of dynamically profiling applications using AI-driven techniques, including the collection and monitoring of profiling data, and the deployment of machine learning algorithms for predictive analysis.

Section 5 discusses the AI-based lifecycle management functions, particularly emphasizing intelligent placement and runtime migration strategies for stateful microservices. It includes experimental validation results demonstrating the effectiveness of these solutions.

Finally, Section 6 concludes the deliverable by summarizing the key outcomes and insights from the implementations and experiments reported, providing a foundation for future development and deployment in the AC³ project.

© AC³ 2023

3 Implemented application LCM functionalities and mapping in the AC3 platform

This section provides an overview of the key AC^3 Cloud-Edge Continuum Configuration Management (CECCM) framework platform functionalities, focusing on those residing on the User Plane and the upper part of the Management Plane. These functionalities include the application descriptor composition mechanism (Section 3), the AI-based application profiling mechanism (Section 4), and the AI-based LCM mechanisms for placement and service migration (Section 5). These functionalities are mapped to the overall architecture, as this has been defined in the final updated version presented in D2.2.

3.1 Overview of the application-related innovative processes in the AC3 CECC

The processes related to the application management in the AC^3 CECCM are defined through the user and management plane functionalities in D2.2 sub-section 3.1 and described in detail in D2.2 sub-sections 3.3 and 3.4, respectively. The key innovations in support of these processes are as follows:

- **Intent-based application descriptor composition**: This process covers the functionalities for capturing user-driven application intents and composing the Application Descriptor. Through the Northbound API (GUI or directly through REST APIs), the end users can define application goals in the form of deployment requirements and SLAs, as well as the related microservices and their interdependencies. The main novelty lies in the additional definition of data requirements, particularly in the combined service and data request processes handled through the Ontology and Semantic Reasoner (OSR). The OSR effectively interprets these intents, querying the Catalogue for available services and data sources. The OSR validates, enriches, and transforms this information into a standardized and rich content YAML Application Descriptor, ready for use by any AI-based LCM, either directly or through adapters.
- **AI-Based application profiling & prediction**: This process involves creating and utilizing AI-based Application Profiles directly from the requested application descriptors. This serves as a core AI-based tool that highlights the smart capabilities of the AC^3 framework. It continuously operates on the fly, collecting real-time monitoring data from the infrastructure. This data is then fed into AI/ML models (e.g., LSTM, Random Forest) to build dynamic profiles. These profiles go beyond simple tracking, generating applications, and SLA-oriented optimal predictions related to traffic patterns and potential performance bottlenecks. This predictive capability is crucial for enabling proactive management, providing the AI-based LCM with the necessary intelligence for optimized, timely decisions.
- **AI-Based Lifecycle Management (LCM) with Advanced Migration**: This process includes the AI-based LCM's core decision-making and execution functions, particularly its advanced service migration capabilities. Using the Application Descriptor and predictive profiles, the LCM manages the entire application lifecycle, including initial placement and runtime adjustments, such as scaling. Its key intelligent feature is the ability to dynamically relocate microservices, supporting stateful migration (through container checkpointing to preserve data) and proactive mobility-induced migration (leveraging Reinforcement Learning and caching to maintain QoS for mobile users). The LCM implements these complex migration strategies by interfacing with the Decision Enforcement and Adaptation Layer, ensuring efficient and resilient application management across the CECC.

3.2 Mapping to the updated AC3 architecture

The mapping of the application LCM-related processes to the updated AC^3 architecture is presented in Figure 1: Mapping of Application related processes to the overall AC3 architecture including also the directly linked processes. At the Application Gateway level, the intent-based application descriptor composition is implemented within the

Otology and Semantic Aware Reasoner (OSR) module. The end-user application intents are fed from the AC³ GUI, delivered under the WP5 activities. A critical step in the overall process is the interaction with the Catalogue module, which is built upon the Piveau framework[1], enabling semantic indexing, efficient query execution, and rich metadata storage, while serving both the secure data and service collection and cataloguing. The implementation details related to the catalogue and its interfaces are included in deliverable D3.4. The OSR interacts with the catalogue to enrich the application description composition with data-related information (location, type, authentication, etc.) as well as service repository information.

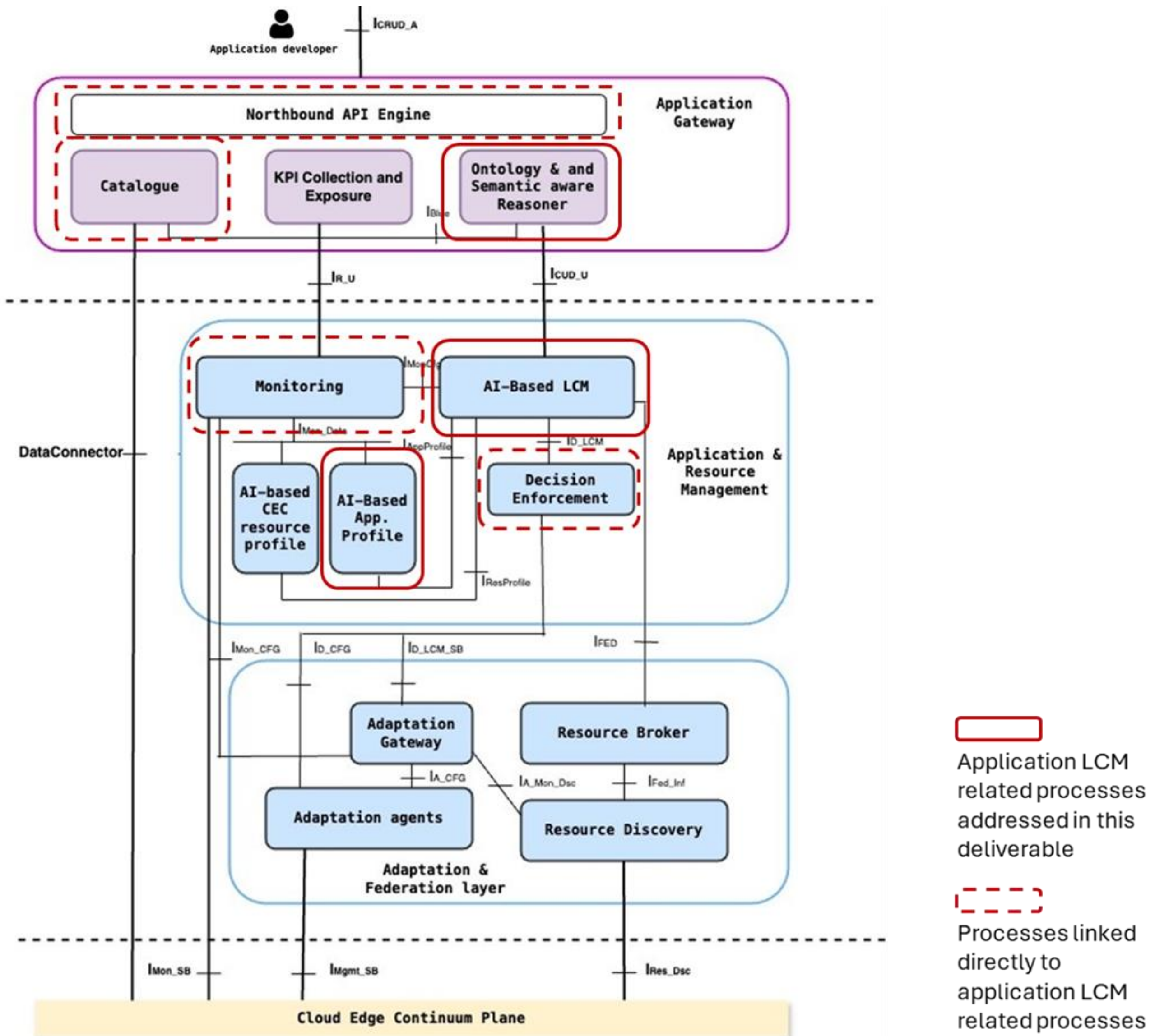


Figure 1: Mapping of Application related processes to the overall AC3 architecture including also the directly linked processes

[1] https://doc.piveau.de/general/introduction/

Figure 1 Mapping of Application related processes to the overall AC3 architecture including also the directly linked processes

On the Application and Resource Management layer, the AI-based LCM is the module that orchestrates the application-specific functionalities for deployment and runtime management phases. The core of the LCM processes depends on the utilized service orchestrator (for example, EUR's LiSO[2] and UBI's MAESTRO[3], both described in D2.3 sub-section 5.1, and deployed under WP5 integration activities). An adaptation layer is developed at the input interface of the LCM module to adapt the application descriptor to the required service management fields of each orchestrator. At this level, the AI-based application profiling & prediction processes are implemented within the Application Profile module. This includes the advanced AI-based models for application-specific predictions based on monitoring information, as well as the application of SLAs. It should be noted that the resource-related decisions are handled within WP4 activities and reported in D4.2. Finally, the AI-based Advanced Migration processes are implemented and can be handled as part of the core LCM mechanism and specifically the engine responsible for the run-time management decisions.

[2] https://www.eurecom.fr/fr/publication/7487

[3] https://themaestro.ubitech.eu/

4 Intent-based models, mechanisms and modules for application deployment

This section details the mechanisms, tools, and architecture that enable the intent-based application deployment workflow in the AC³ framework. It describes how application descriptors are constructed from high-level, user-defined intents, which are semantically processed by the Ontology and Semantic Reasoner (OSR) and then structured into a deployment-ready format for integration into the CECC infrastructure. This section focuses on the composition pipeline involving the Graphical User Interface (GUI), the OSR, and the Piveau Catalog[4], without addressing the downstream lifecycle management (handled in other AC³ modules such as Application Profile Model (APM) and LCM, which are addressed separately in this deliverable).

4.1 Introduction

In AC³, the deployment of microservices-based applications across the Cloud-Edge Continuum is guided by a high-level, intent-based approach. Rather than requiring application developers to manually configure deployment parameters or low-level infrastructure details, the system allows users to specify the desired outcomes of their applications, while intelligent reasoning modules handle the implementation. This section introduces the application descriptor composition mechanism that operationalizes this paradigm.

The main objective of this mechanism is to support the definition, validation, and transformation of high-level application specifications into structured deployment artifacts, referred to as Application Descriptors. These descriptors encode metadata, microservices, networking configurations, data source requirements, and SLA policies in a YAML format that can be consumed by the downstream orchestration plane.

To achieve this, AC³ relies on a coordinated integration between the Graphical User Interface (GUI), which serves as the entry point for application developers; the Ontology and Semantic Reasoner (OSR), which handles intent interpretation, validation, and enrichment; and the Piveau Catalog, which serves as the registry for reusable service and data descriptors. The collective outcome is a standardized and validated deployment specification, generated through semantically enhanced composition logic, that reflects user intent and is compatible with the CECC resource orchestration backend.

This section describes the architecture, data flows, and design logic implemented for this intent-based application descriptor composition mechanism. It excludes the runtime management and optimization logic handled by the APM and LCM and focuses solely on pre-deployment descriptor generation and its upstream interfaces.

4.2 Overview of the design and implementation plan

The implementation of the intent-based deployment framework within AC³ follows a structured approach that transforms high-level user-defined application intents into validated and deployable artifacts. This framework integrates three main components: the Graphical User Interface (GUI) for input collection, the Ontology and Semantic Reasoner (OSR) for semantic processing, and the Piveau-based catalogs for service and dataset discovery. The outcome is a standardized YAML-based application descriptor, which is later translated into platform-specific deployment artifacts, such as NSDs for NFVO/LiSO and Kubernetes manifests for MAESTRO. This workflow ensures that user-defined intents are semantically enriched, optimized for the CECC environment, and seamlessly integrated into the deployment pipeline.

[4] https://piveau.ac3-project.eu/

4.3 Application Descriptor Model

The Application Descriptor Model is a structured framework designed to represent the deployment requirements of microservices-based applications within the Cloud–Edge Continuum (CECC). While this structure has been refined and validated within the scope of D3.2, it builds upon the conceptual model first introduced in Deliverable D3.1 and leverages established descriptor principles from ETSI NFV (Network Service Descriptor) and Kubernetes manifest structures. This ensures alignment with industry standards while extending them to address CECC-specific needs such as distributed microservice placement, SLA-awareness, and semantic integration through the Ontology and Semantic Reasoner (OSR).

The model consists of several sections, as shown in Figure 2 It includes general information about the application (name, version, metadata), microservice configurations, a networking graph illustrating dependencies between microservices, and SLA requirements that must be met throughout the application lifecycle—such as service availability, latency targets, and throughput requirements—depending on the application type.

```
ApplicationName: "Surveillance System"
Version: "1.0.0"

Microservices_configuration:

Networking_graph:

Global_SLA:
  ServiceAvailability: "99.9%"
  MaxLatency: "500 ms"
  MaxResponseTime: "Low"
  DataThroughput: "High"
```

Figure 2: High-level structure of the application descriptor

The Microservices Configuration section contains detailed specifications for each microservice that constitutes the application. Figure 3: Example configuration of a frontend microservice, including general details, environment variables, microservice-specific SLAs, and deployment constraints illustrates an example configuration for the frontend microservice. This configuration includes information such as the microservice name and version, the container image to be used, and the resource allocation required for execution. Resources such as CPU, memory, and GPU are explicitly defined to ensure that the infrastructure can meet the application's performance demands. Additionally, this section includes Service Level Agreement (SLA), which differs from the global application SLAs by focusing on individual service requirements. To further refine the deployment process, the Microservices Configuration section also specifies environment variables, which provide runtime parameters necessary for the microservices to function correctly. These variables may include database credentials, API endpoints, and configuration parameters that allow dynamic adaptability based on the infrastructure and operational conditions. Moreover, deployment constraints are defined to determine whether a microservice should be placed in a cloud, edge, or far-edge environment. This capability is particularly useful in CECC architectures, where computational workloads must be distributed efficiently to optimize performance, reduce latency, and minimize bandwidth consumption.

```yaml
Microservices_configuration:
  - MicroserviceName: "frontend"
    Version: "1.0"
    Image: "capy8ra/ac3-uc2-frontend:latest"
    ID: "frontend"
    Dependencies:
      - "backend"
      - "deepstream"
    ResourceRequirements:
      Cpu: "1 vCPU"
      Memory: "2Gi"
    MicroservicesSLAs:
      ServiceAvailability: "99.9%"
      MaxResponseTime: "Low"
      DataThroughput: "High"
    ReplicaCount: "1"
    EnvironmentVariables:
      - Name: "VITE_USER_SERVICE_URL"
        Value: "http://172.21.16.156:30033/api/v1"
      - Name: "VITE_USER_SERVICE_BASE_URL"
        Value: "http://172.21.16.156:30033"
      - Name: "VITE_REACT_APP_SITE_KEY"
        Value: "XXX"
    apiEndpoint: "http://frontend.fingletek.com"
    apiPort: "5173"
    Protocol: "HTTP/REST"
    InternetAccess: "true"

    GeographicalArea:
      Region: "London-west"
      LocationType: "cloud"
```

Figure 3: Example configuration of a frontend microservice, including general details, environment variables, microservice-specific SLAs, and deployment constraints

Beyond defining microservices, the Networking Graph section of the descriptor is responsible for mapping the communication dependencies between microservices. This section explicitly outlines how microservices interact with one another by specifying source-destination relationships, the protocols used for communication, and the ports through which data is transmitted. This structured approach ensures that inter-service connectivity is secure, reliable, and aligned with application-level networking requirements.

The Networking Graph also incorporates SLA constraints related to network performance, including latency, bandwidth, and error rate thresholds. These constraints guarantee that the network infrastructure is capable of handling the required data transmission speeds while maintaining a low failure rate.

An example of a networking graph is presented in Figure 4: Sample of the networking graph from the descriptor, detailing the connection dependencies, protocols, ports, and SLAs for each link between microservices, which illustrates a structured representation of microservice interconnections. The figure details the connections between microservices, including protocols, ports, and associated SLAs that define communication parameters between services.

```yaml
Networking_graph:
  - Source: "backend"
    Destination: "db"
    Protocol: "TCP"
    Port: "5432"
    ConnectionSLAs:
      Latency: "Less than 500 ms"
      Availability: "99.9%"
      Bandwidth: "High"
      ErrorRate: "Less than 1%"
  - Source: "frontend"
    Destination: "deepstream"
    Protocol: "TCP"
    Port: "8585"
    ConnectionSLAs:
      Latency: "Less than 500 ms"
      Availability: "99.9%"
      Bandwidth: "High"
      ErrorRate: "Less than 1%"
  - Source: "frontend"
    Destination: "backend"
    Protocol: "TCP"
    Port: "8000"
    ConnectionSLAs:
      Latency: "Less than 500 ms"
      Availability: "99.9%"
      Bandwidth: "High"
      ErrorRate: "Less than 1%"
```

Figure 4: Sample of the networking graph from the descriptor, detailing the connection dependencies, protocols, ports, and SLAs for each link between microservices

4.4 Formulation of the application descriptor from the GUI

The process of formulating the application descriptor in AC³ begins with the user defining high-level application requirements through a guided and intuitive graphical interface. This interface provides a structured and modular experience, divided into seven main sections that correspond to different aspects of application deployment. Users are not required to be familiar with low-level deployment logic or orchestration syntax; instead, they engage with clear and descriptive forms that abstract away the complexity of distributed application configuration. For experienced users or integrators, the platform also supports the upload of structured JSON files, which follow the same schema as the form and contain all the necessary deployment metadata. Whether the input is submitted through the form or via JSON, the information is processed by the Ontology and Semantic Reasoner (OSR) to generate a validated, deployment-ready descriptor (Figure 5: Translating High-Level Applications Specifications Workflow).

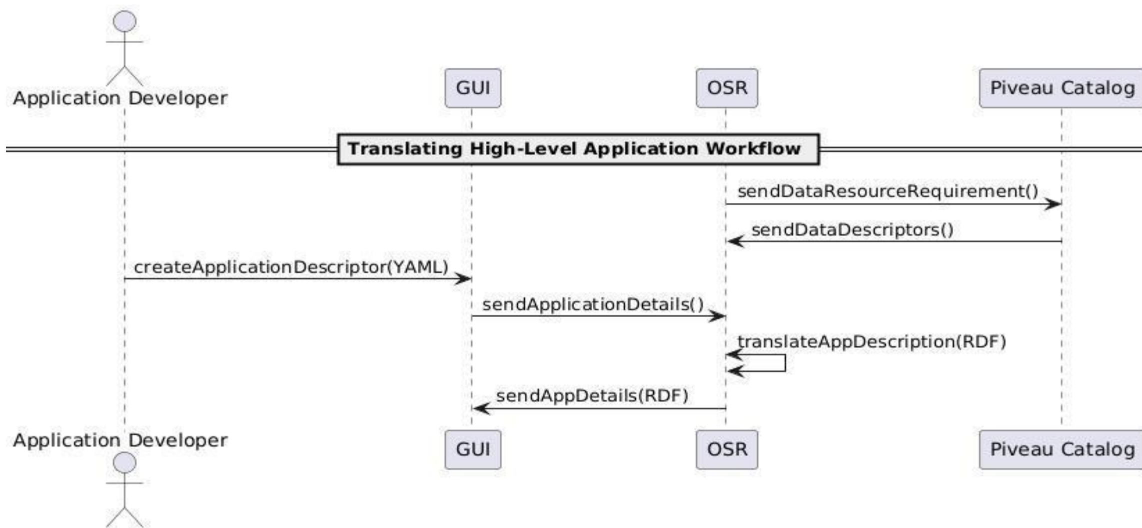


Figure 5: Translating High-Level Applications Specifications Workflow

The first section of the form, "Application General Information", captures the identity of the application. Users are prompted to provide a name, version, and a descriptive summary that contextualizes the purpose of the application. This metadata becomes the foundation of the application descriptor and anchors the relationships between components, services, and policies that will be defined in subsequent steps.

Next, the "Application Consumers" section allows users to describe the intended user groups and their interaction patterns with the deployed application. Here, users define different consumer profiles, specifying interface technologies, communication endpoints, and usage characteristics. Additional details include authentication methods, authorization scopes, and security mechanisms employed at the user interface layer. This information is critical for the OSR to assess connectivity requirements, enforce access control, and inform SLA-aware deployment decisions that reflect real-world usage intensity and interface expectations.

The "Microservices" section is central to the application definition. Users enumerate all the microservices that make up their application, providing service names, images, versioning details, and the purpose of each component. Each microservice entry contains specific API endpoints, dependencies on other services, and detailed resource requirements such as CPU, memory, and storage allocations. Users may also define operational settings like environment variables, scaling strategies, and load balancing rules. This section supports the specification of deployment affinities, initialization containers, and persistent volume configurations, ensuring each microservice is well-described and ready to be placed intelligently by downstream orchestration logic.

Building on that, the "Data Sources" section enables users to indicate the datasets their application requires access to, distinguishing between hot (real-time) and cold (archival) data. Users provide descriptive metadata about each dataset, including its geographic location, access protocols, broker addresses, and associated data models. They also define security measures such as encryption methods and access control techniques. These descriptions allow the OSR to resolve the appropriate data connectors and pre/post-processing services, which are later injected into the final application descriptor through integration with the Piveau catalogue.

Following the data definition, the user configures the "Deployment Context". This section captures information about infrastructure preferences and regulatory constraints. Users define the geographical affinity of microservices, indicating whether specific services should be deployed in cloud, edge, or far-edge environments.

This ensures that latency-sensitive or data-local applications are placed optimally. These contextual constraints guide the OSR in selecting appropriate deployment targets and configuring the runtime environment accordingly.

In the “Network Traffic and Load” section, users describe how different microservices are expected to interact. The form supports the definition of inter-service connections, including protocols, ports, traffic types, and quality of service expectations. Users specify latency targets, availability thresholds, and bandwidth requirements.

Together, these sections enable the AC³ platform to extract a complete, semantically consistent view of the intended application. Once the user submits the application information, either via the form or as a structured JSON template, the Ontology and Semantic Reasoner (OSR) begins constructing the Application Descriptor by mapping each section of the user input to the corresponding part of the descriptor model.

The Microservices Configuration section of the descriptor is populated using the information provided in the microservices composition step. This includes the microservice name, version, container image, resource requirements, environment variables, and scaling policies. Deployment-specific constraints, such as geographical affinity or compute optimization preferences, are extracted from the Deployment Context section and used to inform placement decisions within the Cloud-Edge Continuum.

The Networking Graph, which defines inter-service communication, protocols, and connection-level SLAs, is constructed using inputs from the Network Traffic and Load section. It captures details like source-destination pairs, ports, and communication strategies. Meanwhile, SLA and QoS-related parameters at the infrastructure and service levels are drawn from a combination of microservices, networking, and deployment context inputs.

The Data Sources section triggers more complex reasoning. Here, rather than simply embedding references to user-specified datasets, the OSR interacts with the Piveau Catalogue to identify and retrieve the service connectors, access protocols, and any data transformation modules necessary to access those datasets. For each dataset defined, whether classified as hot or cold, the OSR appends additional microservices to the application descriptor that represents the required data connectors and auxiliary services. This ensures that, upon deployment, the application has the infrastructure and runtime logic needed to access and process the required data seamlessly. While the internal mechanisms of this dynamic interaction are presented in detail in Deliverable D3.4, the key outcome in this deliverable is that user-defined data needs are semantically resolved and converted into fully integrated components of the final descriptor.

The result is the structured, YAML-based application descriptor that unifies all aspects of user intent—logical configuration, data access, networking behavior, deployment context, and security—into a deployable artifact. This descriptor serves as the definitive blueprint used by the AC³ LCM. Its standardized, enriched structure supports automation, correctness, and adaptability, and it enables reproducibility and team collaboration through the availability of templated JSON inputs, all included in the annex of this deliverable.

4.5 KPIs collection and presentation

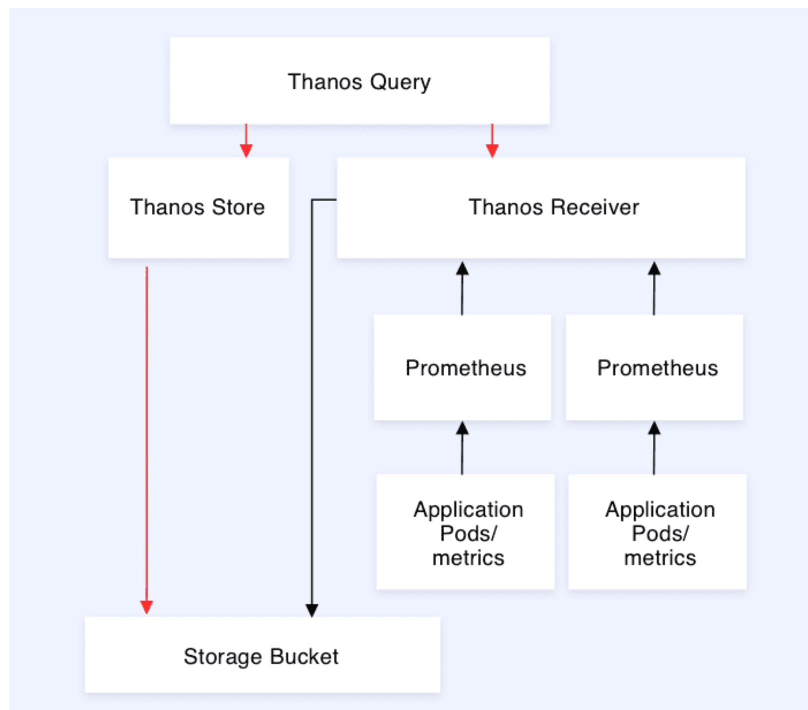
Modern application development heavily relies on Key Performance Indicators (KPIs). They serve as measures of performance, efficiency, and reliability that convert decision-making processes from being based on intuition to objective, data-driven evaluations. Appropriate use of KPIs makes sure that development efforts resonate with more general business objectives, as well as user expectations. Specifically, the AC³ framework guarantees this alignment by means of a structured procedure that translates user requirements into concrete system-level, measurable outcomes, rendering KPIs pertinent rather than just theoretical.

In essence, the whole system implements a mechanism for monitoring. This work involves capturing KPIs, as determined by the user, gathering relevant low-level metrics, performing real-time analysis upon these metrics,

and presenting the processed outcomes in user-friendly and actionable ways. The dissemination is done through two complementary channels: an interactive GUI dashboard (Grafana) and an API that serves programmatic-level access. The dual exposure mode allows the users to either literally see the data or tap into direct engagement on the relevant applications or their analysis pipelines.

This layered model remains extremely accommodating of a variety of end users. Desktop scientists and system and application engineers take advantage of visual dashboards in the living mindset of intuitions and for rapid diagnosis, whereas such highly technical users consume the actual metrics through the exposure of the API, thus affecting automated decision-making processes or deeper integration in CI/CD workflows. This setup enables what one might refer to as "AI-time" responsiveness, where intelligent agents continuously monitor performance regressions, SLA violations, or infrastructure anomalies.

Accordingly, metrics and alerts are exposed by this architecture in a well-defined and standardized format, richly adhering to the OpenMetrics specification[5], and fully compatible with the Thanos monitoring ecosystem[6], thus ensuring consistency and interoperability, particularly when interfacing with any outside platform or infrastructure provider. Thanos is an open-source, highly available Prometheus extension designed to enable long-term storage, global querying, and horizontal scalability for time-series monitoring data in distributed environments. Within the AC3 architecture, Thanos Ruler and Thanos Querier process and expose aggregated monitoring data collected from microservices Figure 6: Architecture Design workflow. These components enable application developers to access enriched metrics and derive actionable insights into application behavior, system performance, and SLA compliance across the Cloud-Edge Computing Continuum. At the core of exposing this information lie two key components:



[5] https://prometheus.io/docs/specs/om/open_metrics_spec/

[6] https://thanos.io/

Figure 6: Architecture Design workflow

The alerting system in Thanos supervises foul play issues before they become aggregate-grade metrics, offering a bird's eye view of system health. These output lines are meant to be read only by programs and carry an extra semantic label. Some descriptors given to alerts or recorded metrics, like `job`, `component`, `region`, are used downstream by other systems to classify and act on these objects. Going beyond the technical bits, these descriptors would include a summary and a detailed description of the communication portal alert. In some cases, links to human-readable documentation or a runbook are also provided. This ponder is the context that translates alerts into immediate and informed action.

The Querier presents a unified interface that extends querying for live and historical metrics from various distributed data sources. It exposes a Prometheus-compatible API so that developers and analytics systems can query data in the usual way. Output is formatted in line with the OpenMetrics standard, conceivably allowing visualization tools like Grafana or Kibana to consume it. Each metric further carries resolution tag(s) (e.g., `resolution="1m"`), helping the users to assess and put the metrics to use based on how coarse or fine they are.

On top of that, for enhancing integration, the components exposing the metrics expose some metadata endpoints for describing the metrics themselves with units, descriptions, and contextual information. The existence of this metadata layer results in a self-describing system, allowing tools and users to interpret a metric without having to refer to internal documentation. In extended scenarios, these metrics may be semantically enriched in case an extra module regarding that aspect is implemented with annotation or ontological mappings (e.g., JSON-LD), thus making them fully discoverable and reusable within and across systems. It will therefore be possible to perform automated reasoning on categories of metrics, like "latency," "availability," or "throughput," which very well may come in handy for AI-enhanced systems or advanced decision-support platforms.

The complete KPI lifecycle in AC^3 is quite well thought out, consisting of five major phases.

1. It starts with KPI Selection, during which the users define what performance aspects really matter in the context of their application. Users may either pick from a preselected catalogue of conventionally accepted metrics based on domain-specific benchmarks and historical performance or make up their own KPIs wherever project-specific needs render a generic approach less applicable. Selecting from the catalogue means that the KPI is considered to be the best-known approach, while opting for the custom will provide fine-grained control in a novel situation.
2. Once KPIs are selected, users must go on to Threshold Definition, where they must specify acceptable performance ranges for each metric in terms of, say, maximum latency or minimum throughput. Thresholds may be static if they are set based on certain business rules or SLA requirements, while some may need to be adaptive because they vary with system conditions, such as user load or traffic volume.
3. Thirdly, during Semantic Mapping, KPIs and their thresholds are passed to the Ontology and Semantic Reasoner (OSR), which maps them into formal semantic entities using an ontology in a structured, machine-readable format. The mapping itself ensures full compatibility with the overall data model of AC^3 and thus ensures its interoperability across systems. The final output, mostly a YAML descriptor, acts as a formal contract between user intent and system behavior.
4. The fourth phase is Monitoring Activation. When the application gets provisioned, the Lifecycle Management System (LMS) starts the monitoring subsystem by passing the application ID and KPI definitions through an API. It starts with collecting metrics and generating alerts in real-time to guarantee that the monitoring works precisely according to what the users have set forth.
5. Lastly, in the Presentation and Exposure stage, KPI data is available to stakeholders (Application Developer/Application Designer, etc). After observing the dashboards in Grafana (if it is deployed or

used), the users themselves can gauge application performance in real time and in direct feedback through the UI. More advanced users can subscribe to the same data stream through the API implementation. Broadly, this provides an asynchronous channel to consume metric events in near real-time and forge insights into larger systems for purposes of compliance reporting, anomaly detection, or automated scaling.

An offering list of the KPIs is presented in Table 2: List of the KPIs provided.

Table 2: List of the KPIs provided

KPI Category	KPI Name	Description	Example PromQL Query (Thanos-Compatible)	Alert/SLO Reference
Resource Utilization	CPU Usage	Percentage of CPU cores used by a pod/deployment/node.	sum(rate(container_cpu_usage_seconds_total{namespace=""<ns>""}[5m])) by (pod) / <CPU_LIMITS>	Alert if > 80% for 5+ mins
Resource Utilization	Memory Usage	Percentage of memory used by a pod/deployment.	sum(container_memory_working_set_bytes{pod=~""<pod>.*""}) by (pod) / <MEMORY_LIMITS>	Alert if > 90% for 5+ mins
Resource Utilization	Disk I/O Pressure	Disk read/write latency or throughput saturation.	rate(node_disk_io_time_seconds_total[5m])	Alert if > 50ms latency
Availability	Pod Restarts	Count of pod restarts (indicates crashes/OOMKills).	sum(kube_pod_container_status_restarts_total{namespace=""<ns>""}) by (pod)	Alert if > 3 restarts/hr
Availability	Deployment Replicas Available	Ratio of desired vs. ready replicas.	kube_deployment_status_replicas_available / kube_deployment_status_replicas_desired	Alert if < 90% available
Availability	API Server Latency	99th percentile latency of K8s API server requests.	sum(rate(apiserver_request_duration_seconds_bucket[5m])))	histogram_quantile(0.99)
Networking	Network Errors	TCP/UDP packet drops or errors.	sum(rate(node_network_receive_errs_total[5m])) by (instance)	Alert if > 10 errors/sec

© AC3 2023

Networking	Request Latency (HTTP/gRPC)	Endpoint latency (e.g., 99th percentile).	sum(rate(http_request_duration_seconds_bucket[5m])) by (path))	histogram_quantile(0.99)
Networking	5xx Errors	Rate of HTTP 5xx errors from ingress controllers.	rate(nginx_ingress_controller_requests{status=~""5..""}[5m])	Alert if error rate > 1%
Storage	Persistent Volume (PV) Usage	Percentage of PV storage used.	kubelet_volume_stats_used_bytes / kubelet_volume_stats_capacity_bytes * 100	Alert if > 85%
Storage	Volume Latency	Storage latency	rate(aws_ebs_volume_total_read_time_seconds[5m])	Alert if > 100ms (P99)
Thanos-Specific	Thanos Store API Latency	Latency of Thanos StoreAPI queries.	sum(rate(thanos_store_api_query_duration_seconds_bucket[5m])))	

For example, if the end user wants CPU utilization under 70% on all the applications running on their project, the monitoring application would automatically produce the following query on Thanos querier:

```
(
 sum(rate(container_cpu_usage_seconds_total{namespace="<your-namespace>"}[5m])) by
(pod, deployment)
 /
 sum(kube_pod_container_resource_limits{namespace="<your-namespace>",
resource="cpu"}) by (pod, deployment)
 ) * 100
```

which in turn, would produce the following alert on Thanos Ruler (alertManager):

```
(
 (sum(rate(container_cpu_usage_seconds_total{namespace="<your-namespace>"}[5m])) by
(pod, deployment)
 /
 sum(kube_pod_container_resource_limits{namespace="<your-namespace>",
resource="cpu"}) by (pod, deployment)
 ) * 100 > 70
```

In order to achieve that, we update the rules configuration file (Figure 7: Rule configuration example) , whenever we receive a set of new KPIs:

```
groups:
- name: cpu-utilization-alerts
  rules:
  - alert: HighCPUUtilization
    expr: |
      (
        sum(rate(container_cpu_usage_seconds_total{namespace="<your-namespace>"}[5m])) by (pod,
deployment)
        /
        sum(kube_pod_container_resource_limits{namespace="<your-namespace>", resource="cpu"}) by
(pod, deployment)
      ) * 100 > 70
    for: 5m  # Trigger only if sustained for 5 minutes
    labels:
      severity: warning
      team: devops
    annotations:
      summary: "High CPU usage in {{ $labels.pod }} ({{ $labels.deployment }})"
      description: "CPU utilization is {{ $value }}% (limit: 70%). Check scaling or resource lim
its."
```

Figure 7: Rule configuration example

4.6 Adaptation layer for LCM

The translation layer serves as the final step in the intent-driven application deployment pipeline within the AC³ framework. Its main function is to adapt the semantically enriched application descriptor, produced through the coordinated efforts of the GUI, Ontology, and Semantic Reasoner (OSR), and Piveau Catalog, into formats suitable for consumption by the downstream lifecycle management (LCM) systems.

In AC³, two LCM solutions are currently used: the NFVO-based LiSO orchestrator and the MAESTRO orchestrator, both introduced in Deliverable D2.3. To interface with these systems, the translation layer transforms the application descriptor into two distinct target formats: a Network Service Descriptor (NSD) for NFVO/LiSO, and a service order for MAESTRO. This ensures that the original high-level deployment intent, encoded in the descriptor, can be operationalized by different infrastructure management tools without requiring manual rewriting or intervention.

Rather than focusing on low-level implementation specifics, the translation layer is designed as a conceptual module that maps the logical components of the descriptor, such as microservices composition, networking graph, SLA constraints, and deployment context, into the structural and semantic models expected by each LCM target. For example, it aligns microservice definitions and connectivity information to the corresponding Appd structures within the NSD, while generating region-specific deployment details and metadata required by service orders in MAESTRO.

This translation step plays a key role in ensuring interoperability between the AC³ application modeling environment and the execution backends, maintaining the separation of concerns between design-time intent expression and runtime orchestration. It also enables seamless integration with heterogeneous infrastructure platforms, allowing the same application descriptor to be executed across different environments depending on the deployment target and operational context. By abstracting and generalizing the transformation logic, the translation layer guarantees that AC³ remains extensible and adaptable to future orchestrators or LCM

mechanisms, further reinforcing the modularity and reusability of the platform's application modeling and deployment workflow.

It is noted that the specific adaptation layer solutions for the LiSO and MAESTRO-based orchestrators have been studied under WP5 activities (i.e., as part of use case adaptation). Figure 8: mapping between one of the use case 2 microservices configuration in the App Descriptor to its equivalent in the NSD format and Figure 9: Example of the application translator mapping for use case 2. Adaptation between AC3 application descriptor and the MAESTRO service order as a custom manifest work provide an example of the implemented adaptations for LiSO and MAESTRO, respectively. Further details are presented in D5.2 and use cases 2 and 3, respectively.

Figure 8: mapping between one of the use case 2 microservices configuration in the App Descriptor to its equivalent in the NSD format

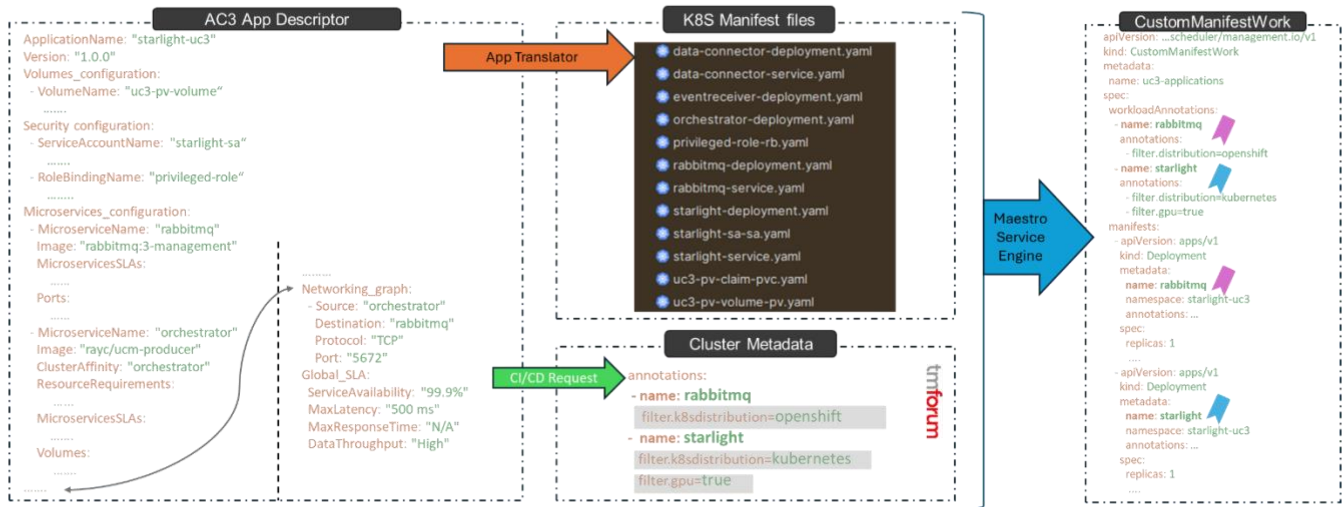


Figure 9: Example of the application translator mapping for use case 2. Adaptation between AC3 application descriptor and the MAESTRO service order as a custom manifest work

5 Application profile model mechanism

This section provides an overview of the design, implementation plan, and current development activities related to the AI-based Application Profiling Mechanism (APM) in the AC^3 project. It outlines the objectives, conceptual architecture, and technological approach used to build a unified profiling system that enables intelligent lifecycle management of microservices-based applications deployed across the Cloud Edge Computing Continuum (CECC).

5.2 Introduction

The AI-based Application Profiling Mechanism (APM) within the AC^3 framework is designed to enable efficient, autonomous, and adaptive management of microservice-based applications deployed across heterogeneous CECC infrastructures. Leveraging Artificial Intelligence (AI) and Machine Learning (ML) methodologies, the APM dynamically captures, analyses, and predicts application behavior, operational context, and resource utilization patterns. The primary objectives include optimizing resource allocation, ensuring compliance with Service Level Agreements (SLAs), improving quality of service (QoS), and enabling real-time adaptive decisions to maintain application performance and reliability.

The approach is grounded in the development of a unified, ontology-based Application Profile Model (APM), which enables formalized, machine-readable descriptions of applications and their runtime requirements. This model supports real-time monitoring, data collection, AI-driven behavioral analysis, and proactive adaptation to changing workloads or environmental conditions. The expected results include enhanced lifecycle management (LCM), improved resilience to workload fluctuations, reduced operational overheads, and more sustainable resource usage throughout the continuum.

5.3 Overview of the design and implementation plan

The rise of the Cloud Edge Computing Continuum (CECC) reflects the increasing demand for scalable, low-latency, and resource-efficient computing environments capable of supporting real-time and data-intensive applications. This paradigm integrates centralized cloud infrastructure with distributed edge nodes, enabling computational tasks to be executed closer to data sources and end-users. This proximity reduces network congestion, minimizes latency, and enhances application responsiveness, particularly in scenarios involving IoT, real-time analytics, and intelligent surveillance.

To effectively operate within this environment, AC^3 adopts microservice-based architecture. Microservices decompose applications into modular, loosely coupled components that can be independently deployed and managed. This modularity offers high adaptability and scalability, which are essential in CECC deployments where application components must be distributed based on latency sensitivity, processing needs, and resource availability. For instance, lightweight services can run at the far edge on UAVs or constrained IoT devices, while compute-intensive modules can be offloaded to more powerful cloud nodes.

However, orchestrating microservices across such a fragmented infrastructure introduces significant challenges in dependency management, runtime optimization, and SLA enforcement. Conventional application profiling methods are inadequate in these contexts, as they often focus on static configurations and ignore the dynamic interplay between microservices, infrastructure constraints, and consumer-driven demand variability.

To address these limitations, the AC^3 project is presenting a Unified Application Profile Model (APM) Mechanism. This mechanism formalizes the structure and semantics of microservices-based applications in CECC through ontology-based representations. Using technologies such as the Web Ontology Language (OWL), the APM ensures semantic consistency, extensibility, and interoperability across the diverse elements of the CECC architecture.

The APM supports runtime characterization of microservices, enabling real-time adaptation based on workload metrics, resource availability, and policy constraints. This semantic model serves as a foundational component for the AI-based profiling mechanism, facilitating:

- Real-time monitoring of application behavior and infrastructure status,
- Predictive resource allocation based on historical and contextual data,
- SLA-aware optimization of deployment and migration strategies,
- Feedback-based adaptation of the application descriptor to reflect changes in operational context.

In practice, this implementation involves the collection of operational metrics such as CPU usage, memory load, latency, and network metrics, which are continuously ingested by the monitoring module. These metrics are analyzed using ML algorithms trained to detect performance anomalies, forecast future loads, and recommend proactive adjustments. For example, in video analytics applications, the system can identify high traffic periods or workload surges and automatically scale the backend services or trigger service migration to edge/cloud nodes accordingly. Furthermore, the profiling mechanism is tightly coupled with the LCM (Lifecycle Management) subsystem, allowing it to influence key decisions such as service placement, scaling, healing, and termination.

The Application Profiling Mechanism represents a cornerstone of the AC^3 framework's intelligence layer. It brings together formal semantic modelling, AI-based runtime analysis, and zero-touch management principles to deliver an adaptive, resilient, and performance-aware platform for orchestrating CECC applications.

5.4 Related works

The exploration of application profiling within the Cloud-Edge Computing Continuum (CECC) and microservices architecture has garnered significant attention in recent research. This area addresses the challenges of managing applications in dynamic, distributed, and heterogeneous environments, where the interplay between user behavior, system performance, and data management is critical. Over the years, profiling methodologies have evolved from narrowly focused user-centric approaches to more comprehensive models that integrate user, system, and data-centric perspectives. This section synthesizes prior work, highlighting key advancements, limitations, and research gaps while proposing the need for a unified application profiling model tailored for CECC environments.

Early profiling methods primarily focused on understanding user behavior and preferences, often for personalization or marketing purposes. For instance, Shaman et al. [1] proposed a model for user profiling based on network metadata at the application level, leveraging a gradient-boosting machine learning algorithm to improve user identification accuracy. As the complexity of applications grew, profiling models began to incorporate system-centric and data-centric elements. Ferrari et al. [2] developed the Linked Data JSON SPARQL Application Profile (JSAP), a resilient model designed for applications exploiting the SPARQL Event Processing Architecture (SEPA). In the context of scientific databases, López-Acosta et al. [3] proposed a Metadata Application Profile (MAP) for structuring large-scale data in Social Network Analysis (SNA).

Despite these advancements, existing profiling models remain fragmented, current methodologies either focus on isolated profiling dimensions (user, system, or data) or rely on static profiling techniques that fail to capture real-time changes in CECC environments. Furthermore, many models lack formal semantic representations, such as ontologies, which ensure interoperability across heterogeneous infrastructures.

To address these limitations, this section introduces the Application Profile Model (APM) — a unified profiling approach that integrates user, system, and data-centric profiling through an OWL-based semantic representation. The APM ensures a holistic, scalable, and interoperable approach for managing microservices-based CECC applications.

5.5 Application Profile Model

The Application Profile Model (APM) developed under the AC³ framework builds upon the initial conceptual structure introduced in Deliverable D3.1 but introduces several new enhancements in D3.2. Specifically, this updated version:

- Expands the model's granularity by adding explicit subcomponents under each of the four layers (Metadata, Application Consumers, Microservices and Data Sources, and SLAs), as depicted in Figure 10.
- Integrates runtime performance metrics (e.g., latency, throughput, error rates) directly into the profiling framework, enabling a tighter coupling between monitoring data and lifecycle management decisions.
- Introduces geographical and interaction pattern attributes for Application Consumers to better support latency-aware and demand-driven service placement in the CECC.
- Adds data access characteristics and regulatory compliance attributes within the Microservices–Data Sources layer to ensure adherence to privacy regulations such as GDPR during deployment and migration.
- Incorporates semantic enrichment via OWL to enhance interoperability across heterogeneous infrastructure and facilitate reasoning-based automation in service orchestration.

These enhancements make the APM not just a descriptive model, but an actionable profiling mechanism capable of feeding AI/ML components with both static and dynamic context for predictive, SLA-aware application management.

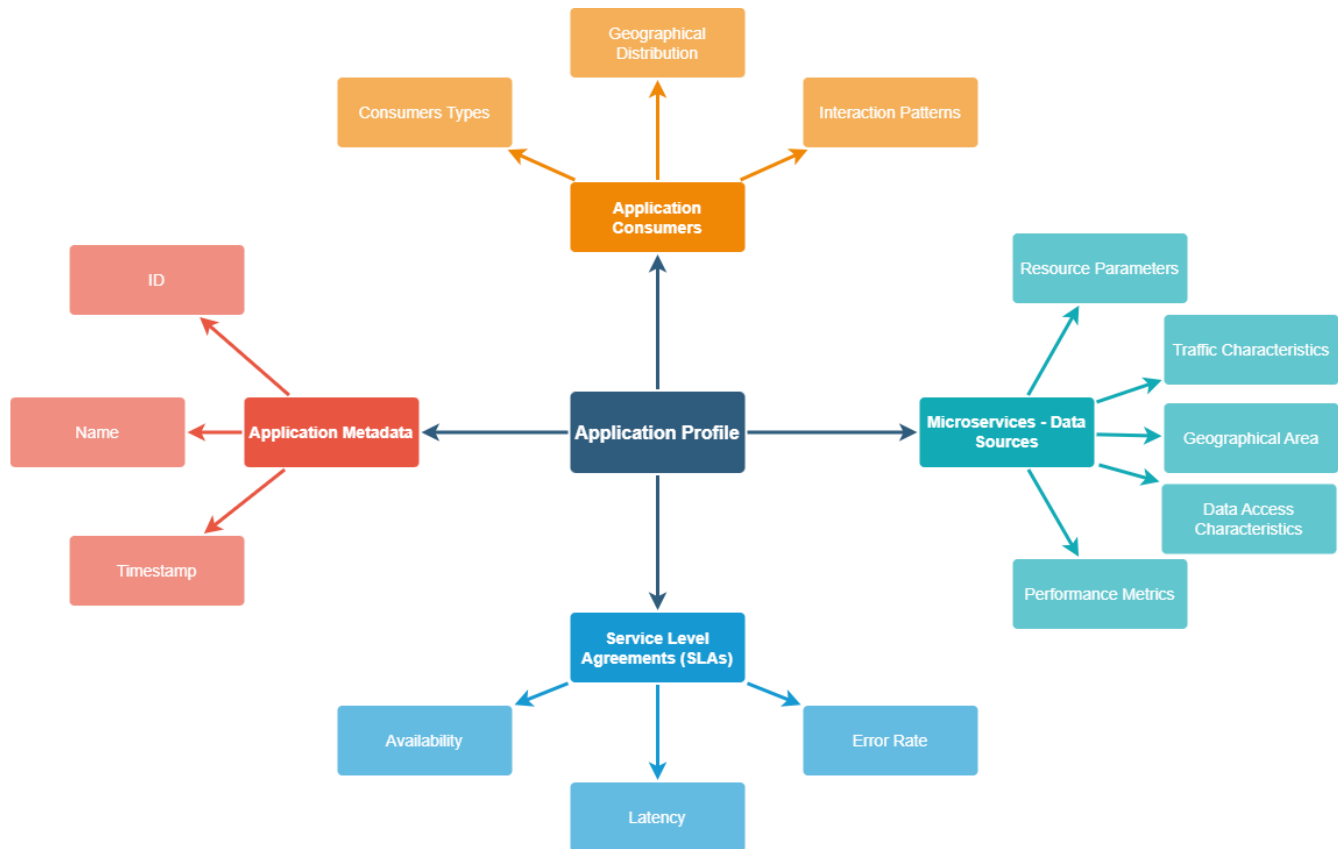


Figure 10: Holistic Representation of the Application Profile Model

5.5.1 Application Metadata

The Metadata Layer forms the foundation of the APM by encapsulating essential identification and lifecycle attributes. It ensures applications are uniquely identifiable across distributed infrastructures, facilitating version control, traceability, and change tracking. This layer includes attributes such as a globally unique identifier, a human-readable name, a versioning system to maintain consistency across updates, and timestamps that log creation and modification times. By integrating these elements, the Metadata Layer ensures seamless application monitoring and management, preventing inconsistencies in distributed environments.

5.5.2 Application Consumers

The Application Consumers Layer captures the interaction dynamics between users and the application, providing insights for performance optimization. It models consumer roles, distinguishing between administrators, developers, and end-users while incorporating geographical distribution data to support latency-sensitive deployments. Additionally, it profiles interaction patterns, analyzing user access frequency, preferred access methods (e.g., mobile app, API), and session durations. This layer enables applications to dynamically adjust resources based on user demand, facilitating intelligent workload distribution and improving overall Quality of Service (QoS). The ability to anticipate high-traffic periods allows proactive resource scaling, minimizing latency and ensuring an optimized user experience.

5.5.3 Microservices and Data Sources

The Microservices-Data Layer defines the structural and functional aspects of microservices-based applications, capturing microservices interdependencies and their associated data sources. Each microservice is profiled based on resource parameters, including CPU, memory, bandwidth, and storage requirements, ensuring optimized allocation across cloud and edge nodes. The APM also tracks traffic characteristics, such as data flow types and communication rates, identifying potential bottlenecks in inter-service interactions. Additionally, geographical deployment considerations allow microservices to be optimally placed within cloud, edge, or far-edge environments based on cost, availability, and latency constraints. Performance monitoring is embedded within this layer, capturing latency, throughput, and error rate metrics to support real-time decision-making. Complementing the microservice component, data sources are analyzed to determine data access patterns, classification (hot vs. cold data), and regulatory constraints such as GDPR compliance. These attributes enable organizations to design intelligent data storage and retrieval mechanisms, ensuring data locality optimization while maintaining regulatory adherence.

5.5.4 Service Level Agreements (SLAs)

The SLAs Profiling Layer formalizes performance guarantees and compliance parameters, ensuring that applications operate within predefined Service Level Agreements (SLAs). This layer establishes core attributes such as availability requirements, latency thresholds, and acceptable error rates, defining the operational constraints of the application. These parameters automate SLA monitoring, allowing applications to dynamically reallocate resources in response to fluctuations in workload conditions. By incorporating predictive analytics, this layer enables proactive resource provisioning, ensuring that latency-sensitive applications maintain uninterrupted service delivery. Additionally, SLA compliance tracking helps mitigate risks associated with service degradation, improving overall system resilience and fault tolerance.

This unified model serves as a semantic basis for dynamic profiling, ensuring interoperability, effective resource allocation, and informed decision-making within CECC.

© AC3 2023

5.6 Data Collection

A fundamental aspect of effective application profiling within CECC is the ability to systematically collect, structure, and refine information over time. To achieve this, the APM classifies profiling data into two primary categories: **Static Data and Dynamic Data.**

Static Data, provided by the end user, refers to deployment-time attributes that remain constant throughout the application lifecycle. These attributes are predefined at deployment and serve as the foundation for system configuration, monitoring, and management. As part of the APM representation, static data includes core application metadata, such as the Application ID, Name, and Version, which uniquely identify the application within the system. It also encompasses Service Level Agreements (SLAs) that define essential operational constraints, including availability guarantees, latency thresholds, and error rate limits. Additionally, static user roles are specified, detailing access privileges and expected interactions within the system. In the context of microservices, this category includes microservice identifiers, dependencies, and initial resource allocation parameters, ensuring that each microservice is properly configured with its CPU, memory, and networking requirements from the outset. Since these attributes are established at deployment, they ensure an accurate baseline profile that serves as the foundation for system monitoring and management.

- Dynamic Data is collected at runtime and continuously evolves as the application operates. Unlike static data, which remains unchanged unless explicitly modified, dynamic profiling enables the system to adjust to real-time conditions and predict future workloads. This category is further divided into predictable and unpredictable dynamic data.
    - Predictable Dynamic Data consists of metrics that follow recognizable trends, such as expected traffic loads, user access frequencies, and seasonal workload fluctuations. These attributes can be forecasted using historical trends and predictive analytics, allowing the system to proactively adjust resources in anticipation of demand surges.
    - Unpredictable Dynamic Data includes high-variance metrics subject to sudden fluctuations, such as unexpected traffic spikes, anomalies in system behavior, and variations in user engagement due to external factors. These unpredictable changes necessitate real-time monitoring and adaptive decision-making to ensure system stability and prevent performance degradation.

The APM's OWL-based structure incorporates both static and dynamic profiling information, as illustrated in Figure 11: APM Data collection, enabling a holistic view of application performance. By structuring static attributes such as metadata and predefined SLAs alongside dynamically collected metrics like real-time latency and throughput, the model facilitates adaptive and predictive resource management. This structured approach ensures that microservices dynamically scale based on changing workloads, user access patterns are continuously refined, and SLA compliance is actively monitored.

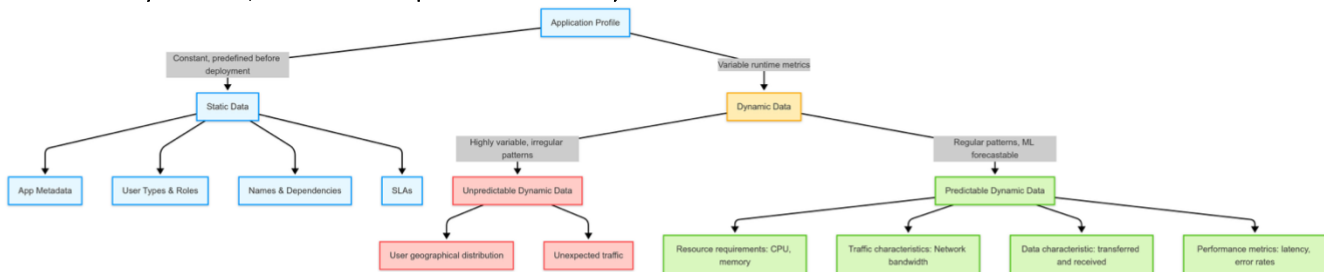


Figure 11: APM Data collection

Before feeding dynamic data collected into machine learning models for intelligent decision-making, a series of comprehensive preprocessing steps is necessary to ensure data reliability, consistency, and usefulness. This preprocessing pipeline maintains the integrity of the application profile model and enables accurate predictions.

- Normalization is applied to unify disparate data formats and scales originating from diverse monitoring sources such as edge nodes, cloud servers, or IoT sensors. Dynamic data metrics—such as CPU utilization, memory consumption, latency, and bandwidth—often come in varying units and value ranges. Normalization scales these values into a consistent format, enabling equitable treatment across features and reducing model bias caused by large value discrepancies.
- Missing Value Handling addresses the frequent issue of incomplete or lost data points in real-time monitoring environments. The causes may include temporary network disconnections, sensor failures, or resource contention on edge nodes. Unaddressed missing data can distort learning outcomes or cause models to fail.
- Noise Reduction techniques are essential for filtering anomalies and outliers that may arise due to transient hardware faults, environmental interference, or data transmission errors. These irregularities can significantly skew performance metrics and degrade the accuracy of predictive models.
- Data Transformation converts raw monitoring inputs into structured formats compatible with downstream ML pipelines. Transformation tasks include aggregating metrics over defined time windows, calculating derived statistics (e.g., rolling averages), and reshaping time-series data into feature-rich matrices.

Through these systematic preprocessing steps, dynamic monitoring data is converted into a semantically rich and structured input that supports robust learning. This enhances the predictive power of machine learning models and ensures they are responsive to evolving operational conditions. The result is a profiling framework that can deliver fine-grained, real-time resource management decisions in Cloud-Edge Continuum Computing (CECC) environments.

5.7 Prediction Algorithm

In developing the AI-based application profiling mechanism—a core component of the AC³ Application Profile Model (APM)—we introduce a new predictive analytics workflow specifically designed for the Cloud–Edge Continuum (CECC) context. Unlike conventional workload prediction approaches, which typically operate in static or homogeneous environments, this mechanism integrates semantic application profiling (as defined in Section 5.5) with fine-grained, CECC-aware performance monitoring. The novelty of this mechanism lies in its ability to:

- Correlate static semantic descriptors (e.g., SLA constraints, geographical affinity, microservice interdependencies) with dynamic runtime metrics to improve forecasting accuracy
- Support cross-environment prediction in heterogeneous CECC infrastructures (cloud, edge, far-edge) where workload distribution, resource constraints, and latency sensitivities vary significantly.
- Enable proactive lifecycle management decisions—such as scaling or migration—based not only on historical performance but also on CECC-specific operational constraints.

This mechanism leverages historical monitoring data to capture recurring daily patterns in workloads, allowing accurate forecasting of load fluctuations and system performance metrics. The predictive component focuses on the following operational parameters

- Disk Throughput (KB/s): Read and write operations.
- CPU Usage (%): Percentage of CPU utilization.

- Memory Usage (KB): Consumption of memory resources.
- Network Throughput (Mbps): Uplink and downlink bandwidth usage.
- Data Transferred and Received (bytes): Data volume metrics.
- Request Rate (requests/sec): Incoming service requests per unit time.
- Latency (ms): Response times of the system.
- Throughput (requests/sec): Overall processing capacity.
- Error Rates (%): Frequency of erroneous responses.

To facilitate the initial modeling without relying solely on live production data, synthetic datasets representing typical operational patterns were generated. These datasets reflect the daily cycle of an application's load, capturing minute-by-minute fluctuations, resulting in high-resolution time-series data. Each data point within the dataset corresponds to a one-minute interval, totaling 1440 data points per day. The modeling methodology adopted for the initial predictive profiling employed the following steps:

1. Feature Selection and Engineering: Initially, a single engineered feature representing each minute as a sequential daily index was used. This decision assumed that workloads repeat cyclically each day. The rolling average method was applied to smooth short-term fluctuations, providing a stable foundation for predictive models.
2. Training and Evaluation Process: The dataset was partitioned into training, validation, and testing subsets to ensure robust evaluation. A rigorous cross-validation procedure was employed, enhancing model reliability and generalization.
   - K-Fold cross-validation with k=3
   - Random shuffle with seed=42 for reproducibility
   - 80-20 train-test split within each fold
3. Algorithm Exploration: Compares multiple regression models, identifying those most effective for CECC-specific workloads, as shown in Table 3: Regressions algorithms Pros and Cons

Table 3: Regressions algorithms Pros and Cons

Model	Pros	Cons
Random Forest Regressor	Handles non-linear relationships and feature interactions well. Offers good performance with minimal parameter tuning. Resistant to overfitting in many practical scenarios. Provides feature importance, supporting explainability.	It can be memory-intensive with large datasets. Slower during prediction if many trees are used.
K-Nearest Neighbors (KNN) Regressor	Simple and intuitive; no training phase needed. Effective at capturing local variations in data.	Poor scalability with large datasets due to high inference time. Sensitive to noisy data and irrelevant features.

	Performs well with datasets exhibiting natural clustering.	Requires careful selection of distance metrics and number of neighbors.
Support Vector Regressor (SVR)	Robust against outliers and high-dimensional data. Good theoretical generalization performance. Flexible with different kernel functions.	Computationally expensive with large datasets. Requires careful tuning of kernel parameters. Limited interpretability.
Gradient Boosting Regressor	High accuracy and flexibility. Good at capturing non-linear relationships. Supports custom loss functions and regularization.	Computationally intensive training process. Susceptible to overfitting if not properly regularized. Slower to train compared to Random Forests.

5.8 Testing and results

The performance of the predictive models was evaluated using the coefficient of determination (R^2), which measures how effectively each model predicts variations in system performance metrics over time. The dataset used for training and evaluation represents the behavior of a specific application instance over a 24-hour period, with measurements collected at one-minute intervals. This high-resolution time-series captures realistic fluctuations in system load across various operational metrics, including CPU usage, memory usage, disk throughput, network throughput, data transfer rates, request rates, latency, throughput, and error rates.

The dataset was synthetically generated to emulate the daily operational cycle of a typical web application, assuming workload patterns repeat on a 24-hour basis. This emulation captures both peak and idle periods, simulating realistic application usage. No live production environment was used; instead, a controlled dataset was constructed to allow consistent benchmarking of predictive models.

Each record in the dataset includes timestamped measurements of all target metrics. Rolling averages with a 10-minute centered window were applied to smooth out noise while preserving short-term fluctuations.

The experiment used the sequence index (minute of the day) as the sole input feature (X), under the assumption that the daily load profile is cyclical and predictable. Each performance metric was modeled independently (Y contains 12 targets). Data was standardized using StandardScaler for both input and output features.

Model performance was assessed using 3-fold cross-validation, with the mean R^2 score computed for each target metric. Negative R^2 values were clipped to zero for visualization clarity.

The evaluation considered multiple regression algorithms: Linear Regression, Support Vector Regressor (SVR), Gradient Boosting Regressor, Random Forest Regressor, and K-Nearest Neighbors (KNN) Regressor. Figure 12: R2 Scores for each target for each tested algorithm demonstrates that both **Random Forest** and **KNN Regressors** consistently achieved the highest performance, with R^2 scores frequently close to or surpassing **0.90** across

nearly all metrics. This suggests exceptional effectiveness in predicting performance metrics such as CPU usage, memory usage, disk read/write, and network throughput, as these parameters exhibit strong cyclical patterns.

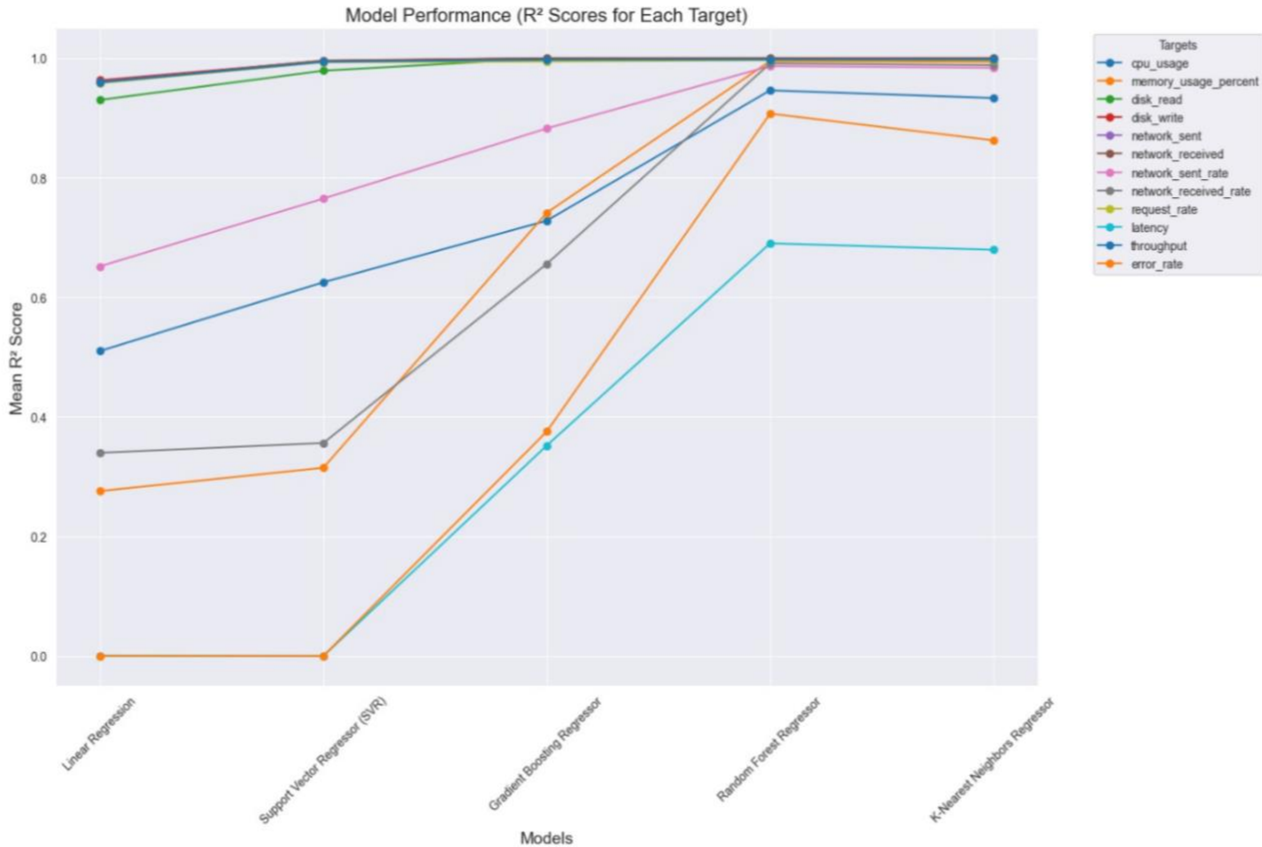


Figure 12: R2 Scores for each target for each tested algorithm

Conversely, **Linear Regression** and **SVR** algorithms displayed significantly lower predictive accuracy, particularly for complex metrics such as latency, error rates, and request rates. For example, Linear Regression yielded near-zero scores for latency and error rates, highlighting its inability to capture the non-linear and complex temporal relationships within the data.

The **Gradient Boosting Regressor** showed intermediate performance, significantly better than Linear Regression and SVR but still somewhat below Random Forest and KNN, especially for predicting latency and error rates. However, it excelled in predicting network throughput and request rate metrics, indicating its capability to handle certain performance parameters well, especially those with moderate complexity and fewer abrupt variations.

The overall analysis of these results strongly supports the initial hypothesis that daily cyclic patterns significantly impact application performance metrics. The superiority of Random Forest and KNN regressors underscores their ability to leverage the temporal continuity and recurrent patterns within the data effectively. These results validate the approach of using a simple sequential index combined with rolling averages, establishing a solid baseline that captures the primary temporal dynamics of application workloads. Future steps will enhance

predictive accuracy further by integrating additional features, temporal contexts, and more advanced modeling architectures.

5.9 Future Perspectives: Enhancing APM with Generative AI and Log Analytics

Looking forward, the Application Profile Model (APM) Mechanism within the AC^3 framework has significant potential for enhancement through the incorporation of advanced generative Artificial Intelligence (AI) techniques and the extended capability of analyzing application logs alongside traditional operational metrics. These enhancements promise to substantially improve adaptability, responsiveness, and predictive accuracy of the lifecycle management of microservice-based applications within the Cloud Edge Computing Continuum (CECC).

Generative AI technologies, particularly transformer-based language models and large-scale generative frameworks, offer transformative possibilities for the APM. These models are well-suited to synthesizing complex and multidimensional patterns across extensive historical datasets, providing deeper insight into application behaviors that traditional regression or classification models might overlook. By employing generative AI, the APM could dynamically generate comprehensive profiles and scenarios based on historical and real-time metrics, proactively identifying potential performance bottlenecks, resource shortages, or impending SLA breaches. This predictive foresight would enable the CECCM to make highly informed and preemptive adjustments to resource allocation and service deployment strategies.

Furthermore, generative AI opens possibilities for interactive intent-based resource management, allowing users to specify requirements in natural language or high-level abstractions. The model would interpret these specifications and automatically translate them into actionable deployment descriptors and resource provisioning strategies. Consequently, this would significantly simplify user interactions, promote intuitive, user-friendly management interfaces and lowering the barrier to entry for managing complex CECC deployments.

In addition to leveraging advanced generative AI, future iterations of the APM should integrate log analysis capabilities alongside metrics-based monitoring. Application and system logs contain rich contextual information that traditional performance metrics alone cannot fully capture. Logs provide qualitative insights into system events, errors, configuration changes, and user interactions, offering essential context for understanding metric anomalies and performance fluctuations. By incorporating sophisticated log analysis, including natural language processing (NLP) and anomaly detection algorithms, the APM would achieve a more holistic understanding of the application's operational environment.

For example, analyzing logs in real-time can help detect subtle issues, such as recurrent error messages or warnings preceding system degradation, which metrics might initially overlook. Integrating log data with performance metrics can enable the identification of root causes behind performance anomalies, facilitating faster troubleshooting and remediation. This combined analysis approach also supports improved anomaly detection by distinguishing between genuine performance anomalies and benign metric fluctuations.

Furthermore, predictive models enriched with log data can achieve enhanced accuracy, particularly for unpredictable dynamic data scenarios. Logs provide valuable contextual hints regarding external events, software configuration changes, and user actions that metrics alone cannot capture. Utilizing this combined data approach would significantly improve the predictive power of the APM's ML algorithms, enabling them to better anticipate and adapt to both expected and unexpected workload fluctuations and operational events.

Finally, the integration of generative AI technologies and advanced log analytics into the Application Profile Model Mechanism offers exciting opportunities for the future development of the AC^3 project. These

enhancements promise more robust predictive capabilities, improved resource optimization, streamlined user interaction, and comprehensive operational visibility, further strengthening the CECCM's ability to manage microservices-based applications autonomously and intelligently within federated cloud-edge environments.

6 Service migration mechanisms

6.1 Resiliency Focused Proactive migration for Stateful Microservices in Multi-Cluster Containerized Environments

6.1.1 Related work

Existing research covers application migration and stateful microservices management, but to our knowledge, no work addresses both simultaneously. [4] employs reinforcement learning for microservices migration, considering user mobility and latency. While innovative, it focuses only on policy formulation without practical migration implementation. [5] proposes using OverlayFS for container relocation in Kubernetes, transferring disk state via snapshots. However, it does not address the process of state migration, a key requirement in production environments. [6] (MyceDrive) enables Kubernetes pod migration with full state preservation using DMTCP instead of CRIU. However, it requires a custom execution agent per container, adding overhead and limiting scalability. Different works have been done in this area: [7] combines Bi-LSTM fault prediction with stateful migration but focuses only on CPU overload, whereas our solution also considers memory and supports multiple Kubernetes distributions. [8] provides fault tolerance via periodic checkpointing but is reactive, risks state loss between checkpoints, and targets monolithic applications rather than microservices. [9] improves stateful microservices HA using standby pods but does not handle node failures, which is a core focus of our work. [10] introduces a Persistent Volume Autoscaler for storage scaling, addressing challenges orthogonal to ours. [11] optimizes migration for ML workloads in Kubernetes by parallelizing data transfer but is domain-specific and focuses on disk state, not process state. [12] models microservice migration via reinforcement learning, but only support stateless services, neglecting state persistence.

6.1.2 Solution

Since the checkpoint feature was introduced as an alpha feature in the CRI-O container runtime, research on container checkpointing has gained significant attention lately. In AC^3, we leverage this capability to enable stateful microservice migration. Our focus is on resiliency-driven, proactive lifecycle management for stateful microservices in multi-cluster containerized environments. The solution provides a proactive, zero-touch management approach that ensures seamless application lifecycle management. It enables the migration of containers from resource-constrained nodes to available ones while preserving their state, preventing data loss. Our solution integrates seamlessly with container platforms like Kubernetes and supports multi-cluster environments, enhancing fault tolerance and data persistence in stateful applications. We have extensively tested it on various hardware configurations in public cloud environments and our on-premises servers.

To ensure the seamless operation of stateful microservices applications throughout their lifecycle, we propose a zero-touch management solution. It autonomously handles application placement, migration, and termination while preventing data loss during migration. Migration decisions are made proactively using machine learning models trained on real-world datasets to forecast memory and CPU consumption based on historical data provided by the resource exposer component. A key feature of our solution is its ability to migrate services across multiple clusters, expanding re-scheduling options when a cluster lacks sufficient resources. Our approach employs a two-tier scheduling algorithm: the resource orchestrator selects the target cluster in the first tier, while the cluster manager assigns the computing node in the second tier. This strategy optimizes scheduling time, ensures near-optimal microservice placement, and minimizes latency between application components.

Figure 13 presents a high-level conceptual architecture of our solution. The figure demonstrates how the architecture integrates with an infrastructure containing two Kubernetes clusters, each with two worker nodes.

Additionally, it shows two microservices applications: their components are highlighted in yellow (Application 1) and green (Application 2). While all microservices of Application 2 are deployed within one cluster, some microservices of Application 1 are either scheduled or migrated to the second cluster due to a lack of resources in the first cluster at certain points in the application's lifecycle. For simplicity, not all Kubernetes components are depicted in the schema; instead, we focus on showcasing the components specifically added to the architecture. Our contributions to this architecture are highlighted in red. In the next subsections, we will detail each component and provide the end-to-end workflow of our solution to offer a better understanding.

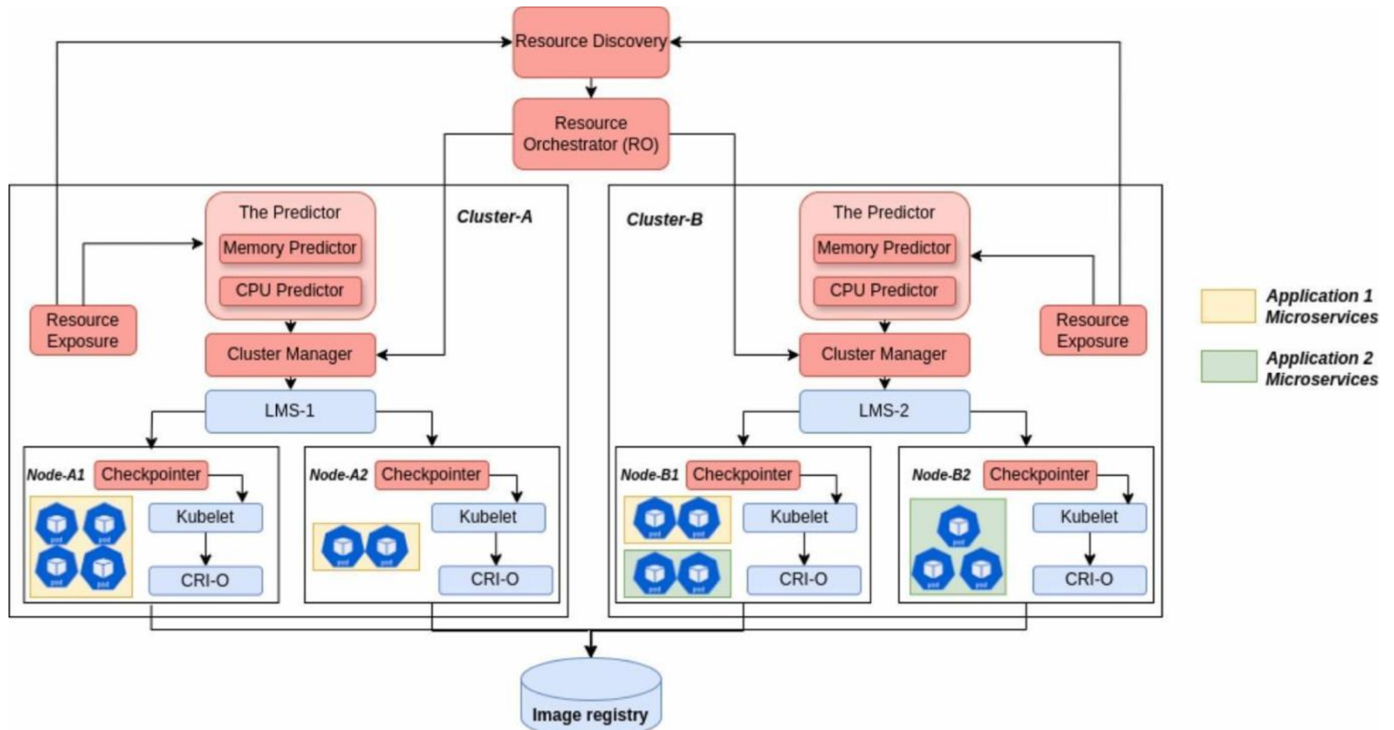


Figure 13: Stateful applications LCM Architecture

6.1.3 Resource Discovery and Resource Exposer

The resource discovery module provides the Resource Orchestrator (RO) with real-time information on network and compute resources across clusters. It collects data from resource exposer agents deployed on each cluster, ensuring up-to-date resource availability for scheduling decisions.

6.1.4 Resource Orchestrator (RO) (AI-Based LCM)

The RO centrally manages applications in a cloud-native system with clusters running Kubernetes, K3s, or OpenShift. It acts as a first-tier scheduler, selecting the most suitable cluster based on memory and CPU availability. The selection is guided by a configurable parameter that balances CPU and memory considerations. To minimize latency, batch scheduling ensures microservices from the same application are placed together whenever possible. Scheduling priority is based on a microservice's availability KPI, favoring those with higher uptime requirements.

6.1.5 Cluster Manager (LMS)

The cluster manager bridges the RO and local management systems, handling node-level scheduling. It selects the optimal node for deployment using a scoring formula similar to the RO's cluster selection. To maintain high availability, a Safety Threshold prevents overloading nodes, while a Migration Threshold triggers pod migration only when resource usage becomes critical.

6.1.6 Predictor

To prevent performance degradation, the predictor component uses LSTM-based machine learning models trained on the GWA-T-13 Materna dataset to forecast CPU and memory usage. Predictions exceeding predefined thresholds trigger proactive stateful pod migration. The predictor operates on each cluster, continuously monitoring resource usage and aiding in preemptive scaling.

6.1.7 Checkpointer Operator

Kubernetes' checkpointing [15] feature is exposed via a custom operator, simplifying stateful container migration. Checkpoints are stored in a shared image registry, allowing seamless restoration on a different node without direct node-to-node connections. The checkpointer operator uses Buildah [16] for efficient image creation and transfer, ensuring lightweight, parallelized operations.

6.1.8 Workflow

Figure 14 illustrates a simplified workflow of the system components. To clarify the application lifecycle management handled by our solution, we outline eight key steps, from deployment to migration. For brevity, interactions between LMSs and lower-level components (e.g., kubelet, container runtime) are omitted, as they fall outside our contributions. The following sections detail each step.

6.1.8.1 Step 1; first-tier scheduling

Before the deployment process begins, the application developer (or the entity deploying the application) must provide the RO with an application descriptor file, formatted in YAML or JSON. This file contains the detailed specifications of the application's microservices, including global parameters such as open ports, computing resource requests and limits, environment variables, and, most importantly, the availability KPI for each microservice. For the first lscheduling operation, the content of this file will be similar to the output of the osr. Upon receiving the application deployment request, the RO communicates with the resource discovery to obtain an overview of the infrastructure. Based on this information, the RO selects the most suitable cluster for deployment by considering resource availability. The microservices are then sorted by their availability KPI, ensuring that those with higher availability constraints are prioritized. Finally, the RO sends a request to the cluster manager instance deployed within the selected cluster to handle the microservices' deployment.

6.1.8.2 Step 2; second-tier scheduling

Upon receiving the deployment request from the RO, the cluster manager gathers information from the resource exposer component to select the most available node within the chosen cluster. After selecting the computing node for a given microservice, the cluster manager contacts the local LMS to enforce the decision and deploy the corresponding pod. This step, along with the next one, is repeated for each microservice in the application. However, in cases where the resource consumption of all nodes exceeds the predefined threshold, some microservices may remain unscheduled.

6.1.8.3 Step 3; pod deployment

The Local Management System (LMS) pulls the base images for the microservices from the image registry and starts the pods. After the loops in Steps 2 and 3 are completed for all microservices, it may happen that some microservices remain unscheduled. This can occur if the local LMS determines that the selected node lacks sufficient resources to meet a microservice's requirements, even if the node's overall resource consumption has not exceeded the threshold. In such cases, at the end of the loops of steps 2 and 3, the cluster manager reports back to the RO, indicating the non-deployed microservices. The RO then restarts the scheduling process from step 1 by searching for another suitable cluster. This iterative process continues until all microservices are successfully deployed.

6.1.8.4 Step 4; resource predictions

After scheduling all the application' microservices, the predictor continuously gathers information about resource consumption to predict future values. These predictions are sent to the cluster manager to make proactive decisions.

6.1.8.5 Step 5; saving the container state

If the cluster manager detects that the predicted resource usage exceeds the set thresholds on a specific computing node, it triggers a migration process for the microservices deployed on that node. This process involves several steps, starting with sorting the microservices. However, unlike the initial deployment, migration prioritizes microservices with lower availability requirements. This approach is chosen because migration can be time-consuming, depending on various factors highlighted in Section 5. By migrating microservices with lower availability constraints first, it is possible to reduce resource consumption to a level where migrating higher priority microservices becomes unnecessary. Once the microservices are sorted by availability, the LMS proceeds by checkpointing the container states, deleting them, and repeating this process until the node's resource usage returns to acceptable levels.

6.1.8.6 Step 6 and 7; build images, looking for nodes

In parallel with Step 5, during Step 6, the checkpointer builds Docker images from the checkpoint files saved earlier and pushes them to the image registry to facilitate the migration process. Simultaneously, in Step 7, the cluster manager begins searching for other available nodes within the same cluster. Our solution prioritizes migration operations within the same cluster to maintain low latency between microservices. This approach is generally faster than inter-cluster migration due to factors such as the reduced latency between the infrastructure and the image registry. By focusing on local migrations, we aim to minimize downtime and ensure efficient rescheduling of evacuated microservices.

6.1.8.7 Steps 8, 9 ; rescheduling and restoring the pod

Depending on the results from step 7, if an appropriate node is found, Step 8 involves the cluster manager requesting the LMS to deploy the pod on this new node. Unlike Step 3, where the LMS uses the base microservice image, here, the LMS pulls the updated image that includes the container's state. This allows the pod to resume operation from its previous state. However, in situations where no suitable nodes are available, such as during peak times or in single-node clusters, if all nodes exceed certain resource thresholds, Step 8 comes into play. In this step, the issue is flagged to the RO, who then seeks out a new cluster for the pod. The RO instructs the cluster manager of the new cluster to allocate a node for the pod, akin to the initial placement process. Once a node is assigned, the new LMS retrieves the checkpointed image and restores the pod on the selected node in the new cluster. Steps 7, 8, or 9 will be repeated until all evacuated microservices are re-scheduled again.

© AC3 2023

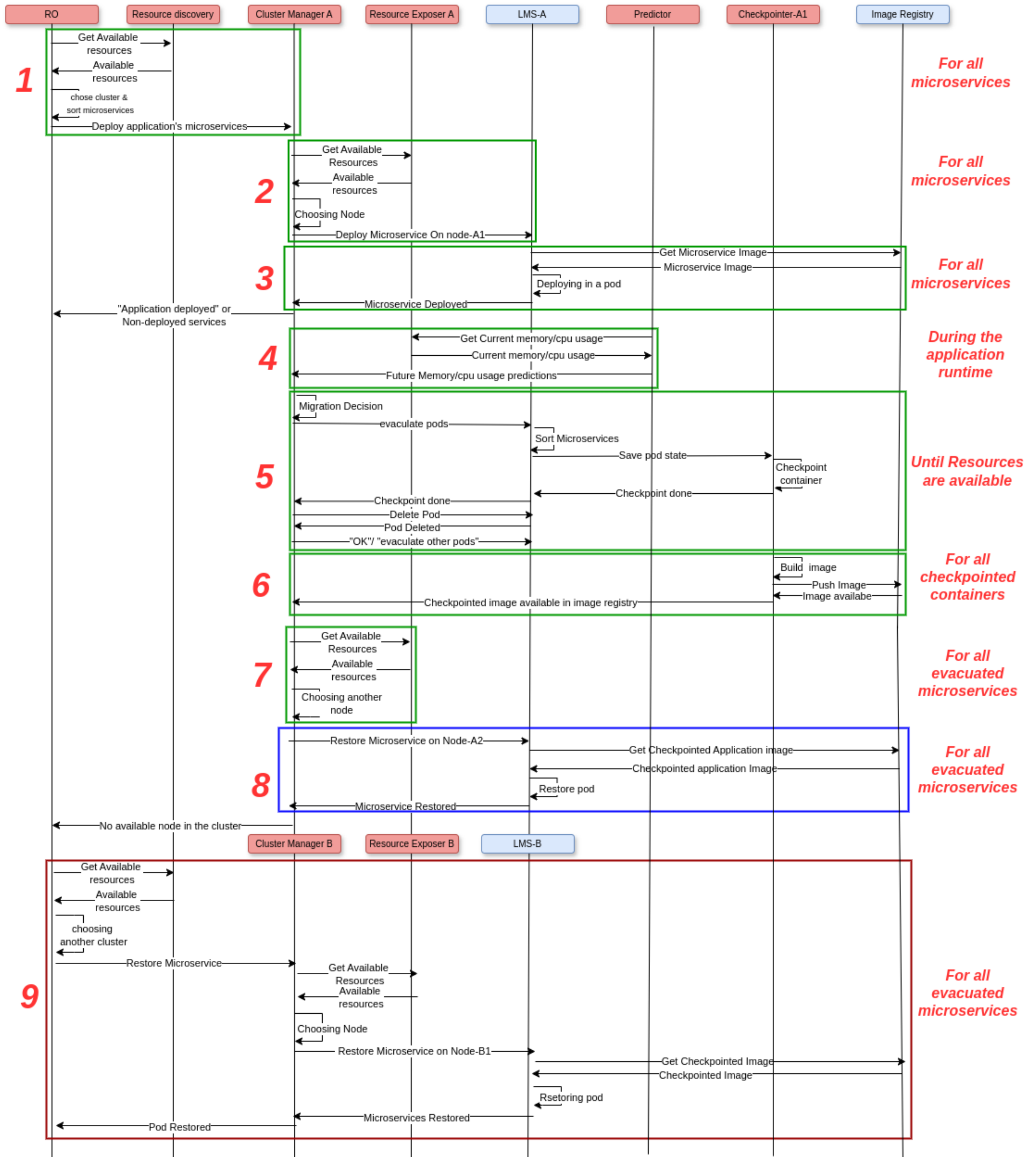


Figure 14: System components workflow

6.1.9 Testing and results

In this section, we will showcase the results obtained from our solution across different phases of the application lifecycle in different environments and with various hardware resources and use cases. This section is divided into two main subsections. The first subsection will discuss the results obtained from our solution before migration, focusing primarily on the predictor component's results. The second subsection will examine the solution's performance during migration, focusing on the container's state-saving time, rescheduling time, and pod restart time under different scenarios, such as single cluster and multi-cluster migration, in on-premises and public cloud servers and to validate our results, we repeated our tests multiple times under the two different scenarios using different hardware configurations.

In the multiple cluster migration scenario, we used resources from the IONOS public cloud provider. We provisioned two single-node clusters in separate geographical data centers: the original cluster, which hosted the pods, was located in a data center in France, while the destination cluster was situated in Germany. Both nodes were equipped with 4 Intel Ice Lake CPU cores, 4 GB of RAM, and an SSD hard disk. The IONOS cloud platform offers various (but limited) disk performance options, so we tested different configurations on the source node to investigate how changes in disk performance impact migration time, particularly the container state saving time, the disk configurations that we used in our tests for the multi-cluster migration scenario are summarized in Table 4. In this scenario, we used the public QUAY image registry [17] as the image registry. This choice was made to ensure that the setup is cluster independent. In this scenario, we will refer to the source node as France-node and the destination node as Germany-node.

In the single cluster migration scenario, we implemented a Kubernetes cluster configuration consisting of one virtual machine serving as the master node and two additional virtual machines designated as worker nodes. The image registry, which hosts the checkpointed images, was deployed on the cluster master node. Each machine was configured with 4 CPU cores and 4096 MB of RAM. The physical machine hosting these virtual machines features a 13th Gen Intel Core i7-1365U processor, with a maximum frequency of up to 3.70 GHz for E cores and up to 5.00 GHz for P cores, accompanied by 32 GB of LPDDR5-6400 MHz memory. The physical machine provisioning the cluster is equipped with an SSD hard disk, which has the following Input/Output Operations Per Second (IOPS) and sequential read and write performance metrics: Sequential Read: Up to 7000 MB/s; Sequential Write: Up to 5100 MB/s; Random Read IOPS: Up to 1,000,000 IOPS; Random Write IOPS: Up to 1,000,000 IOPS. In this scenario, worker-node01 will serve as the source node for migration, while worker-node02 will act as the destination node.

For both scenarios, we used Kubernetes v1.26.0 as the local management system for the clusters and CRI-O v1.26.0 as the container runtime to run containers in the nodes of the clusters.

Table 4: Disk performance metrics

Config	R IOPS	W IOPS	Sequential read	Sequential write
Config 1	30,000	20,000	400 MB/s	400 MB/s
Config 2	37,500	25,000	500 MB/s	500 MB/s
Config 3	45,000	30,000	600 MB/s	600 MB/s

6.1.10 Before the migration

As we mentioned earlier, the predictor component of our solution consists of two LSTM models designed to forecast CPU and memory time series data. In the training phase, both models take the same input: a vector containing past memory and CPU usage values. This approach was chosen because our experiments demonstrated that using both metrics as input yielded better results than training each model solely on its respective output (i.e., training the CPU model only on CPU values and the memory model only on memory values). This improvement is due to the correlation between the two metrics. The models are trained on two subsets of the GWA-T-13 Materna dataset, each subset contains around 10K time series values, obtained by merging data from different types of machines to ensure compatibility with a wide range of input values. The LSTM algorithm was selected for its ability to effectively capture long-term dependencies and patterns in sequential data, making it well-suited for our large subsets. We trained the models using different time window sizes, representing the length of past data sequences used as input for the model.

To evaluate and compare the results of our models, we use commonly employed time-series evaluation metrics: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE).

Figure 15 and Figure 16 present the changes in evaluation metrics for the memory and CPU models as the time window values vary from 2 to 20. Examining both figures, we see that the models demonstrate similar behavior in response to changes in the time window parameter. Initially, increasing the time window positively affects model performance, enhancing accuracy up to a certain point—12 time periods for the memory model and 10 time periods for the CPU model. Beyond these points, further increases in the time window do not improve model performance and may lead to unpredictable results. This pattern can be explained by the hypothesis that current memory and CPU consumption are more strongly related to recent values rather than older data. Given that the GWA-T-13 Materna dataset is sampled at 5-minute intervals and since we are predicting only a single near feature value, the model does not need to detect from distant past periods, so it is reasonable to assume that current memory and CPU values are mainly influenced by the most recent 10 to 12 time periods. Therefore, incorporating older values with a larger time window does not provide significant benefits for predicting current memory and CPU usage.

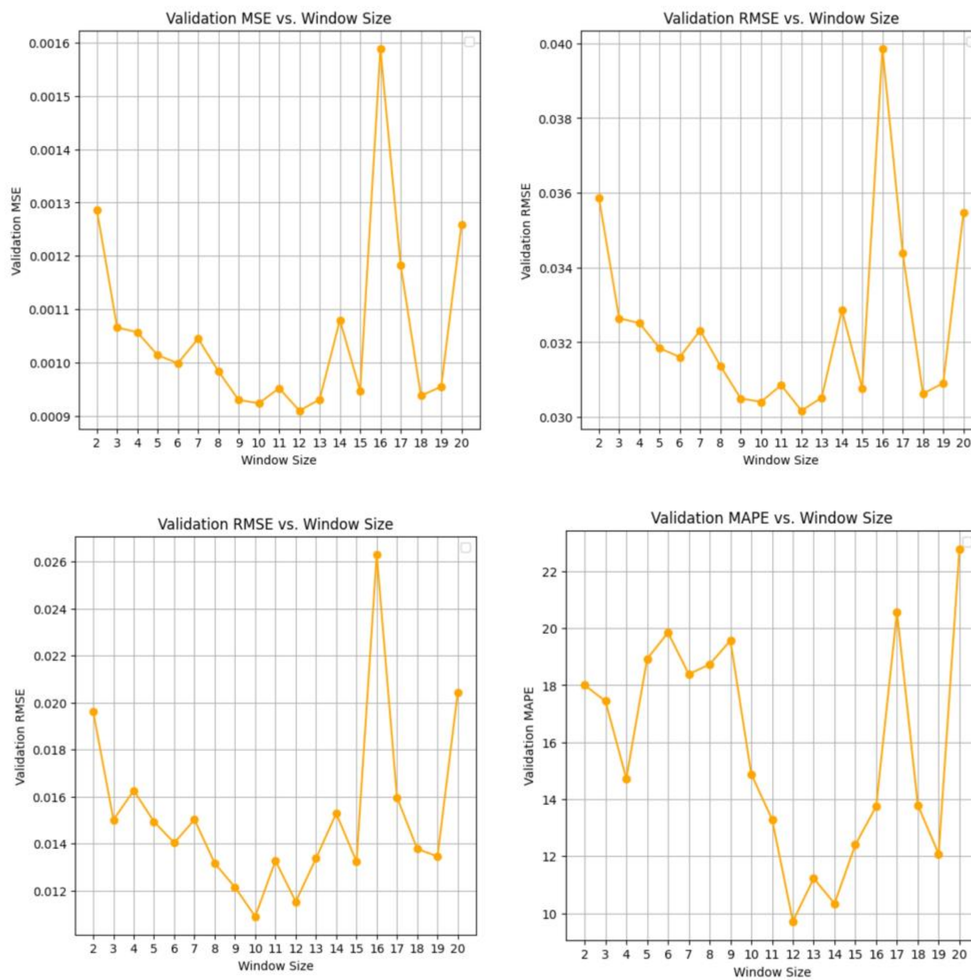


Figure 15a: Model's MSE, RMSE, MAE, and MAPE across Different Window Sizes for the memory predictors models

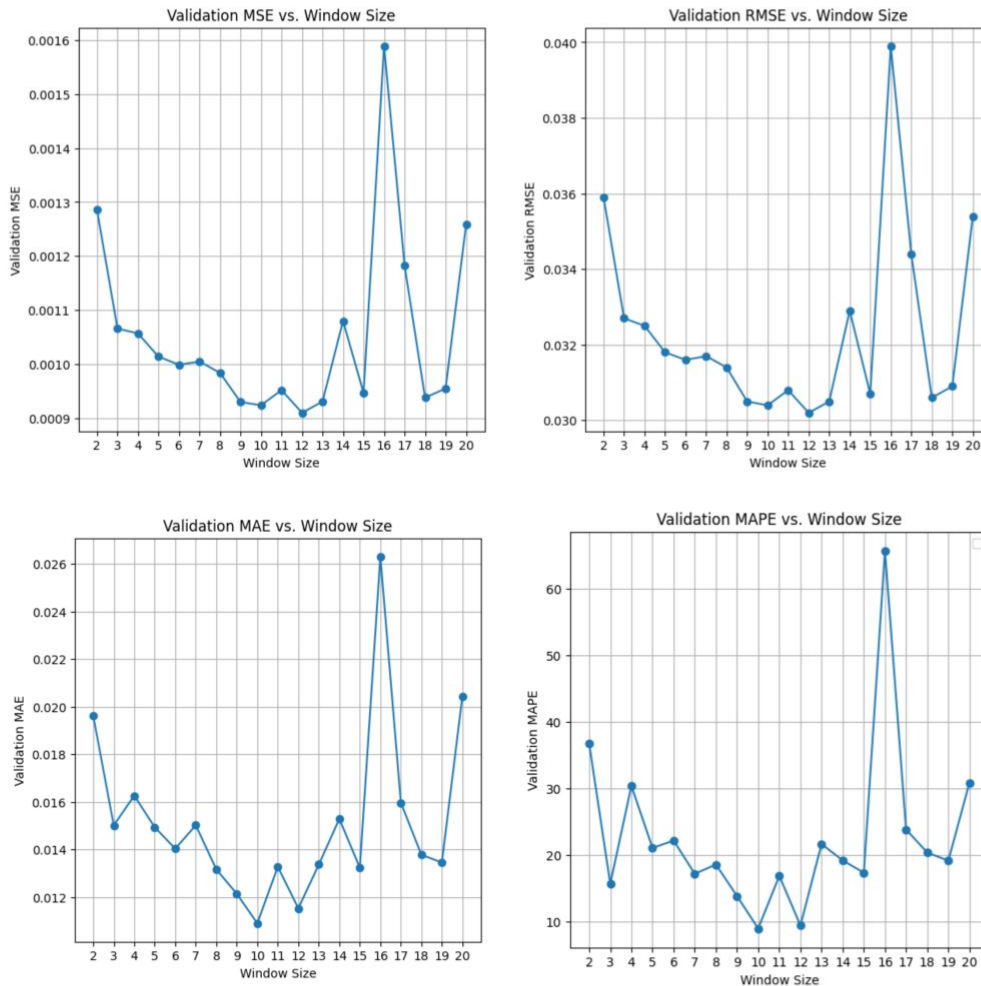


Figure 16b: Model's MSE, RMSE, MAE, and MAPE across Different Window Sizes for the cpu predictors models

Considering that the predictor needs to function in real-time, relying solely on time series validation metrics is insufficient. The predictor must operate smoothly so the entire system remains unaffected by model delays. Therefore, we also present the average inference time of both types of models, measured after running the models on an Intel i7 1355 U processor, in Figure 17.

Looking at these values, we observe that increasing the time window generally results in a longer inference time. However, this increase is negligible (on the scale of milliseconds), allowing us to select the best-performing pair of models for deployment in our solution. The metrics for the selected models are detailed in Table 5. As observed from the table, the values for the MSE, MAE and RMSE are notably low, indicating strong model performance. MSE, in particular, is designed to penalize larger errors more heavily due to its quadratic nature, meaning that it gives greater emphasis to outliers. The low MSE values suggest that our models effectively minimize significant errors. This observation is further corroborated by the Mean Absolute Percentage Error (MAPE) values, which show that the average prediction errors are 9.71% for the memory model and 8.96% for the CPU model, which we consider good values in our context.

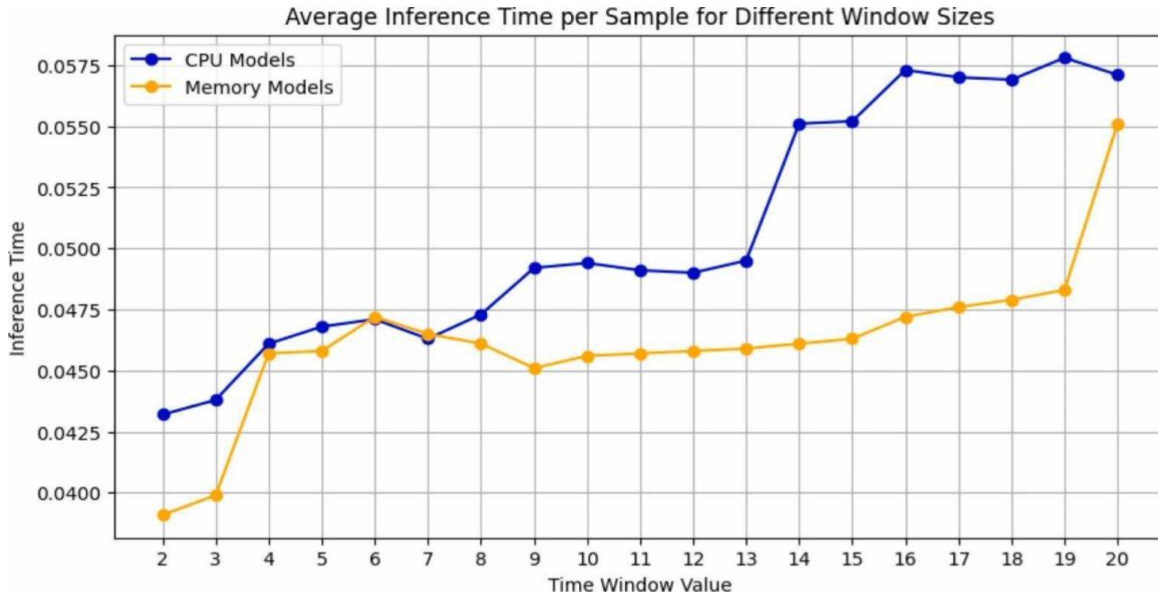


Figure 17: Average Inference Time (Seconds) for both resource models

Table 5: Model's performance metrics

Model	MSE	RMSE	MAE	MAPE (%)	Inference time (S)
Memory	0.002148	0.046349	0.017732	9.716583	0.0458
CPU	0.000924	0.030405	0.010894	8.962406	0.0494

6.1.11 During the Migration

This section examines our solution's performance during migration, from the migration trigger to the application's restart at the destination. Migration time consists of saving the pod containers, re-scheduling, and restarting the application. Optimizing migration requires minimizing these three components. We evaluate our approach using Redis, KeyDB, and SQLite, injecting datasets of 10 MB to 500 MB to simulate varying container states.

6.1.11.1 Container State Saving Time

As previously mentioned, saving the container state involves checkpointing the container, creating a new image from this checkpoint, and pushing this image to an image registry. Once the image is successfully pushed, the container state is considered saved. To provide a deeper insight into how each of these processes contributes to the total migration time, **Figure 18** presents the average time, in seconds, required to complete each of these processes for different container sizes or each type of database on the source computing nodes in both scenarios. Specifically, **Figure 18**a shows the detailed saving time for the Redis database, **Figure 18**b presents the saving time for KeyDB, a Redis alternative, and **Figure 18**c details the saving time for the SQLite database. It is worth noting

that for SQLite, we are using the standard Python library, which results in a larger base image (76.68 MB compressed) compared to the official images used for Redis (16.04 MB) and KeyDB (26.67 MB).

By examining the figure, we can observe two key points. 1) First, as container size increases, the time required for checkpointing, creating a new image from this checkpoint, and pushing the image also increases for all types of applications, resulting in longer saving times. 2) Second, altering the hard disk configuration impacts each of these times, particularly the image-building time. This is because building an image from a checkpoint file involves copying the filesystem from the node to the container, a process that benefits from better disk performance and thus takes less time with higher-performing disks. Diving deeper, we observe that, except for the single-cluster migration scenario, image pushing time constitutes the largest portion of the overall container saving time. This can be explained by the fact that, in multi-cluster migration scenarios, the image registry is typically remote and shared among multiple entities. This setup introduces significant network latency between the remote registry and the source migration nodes, unlike in single-cluster scenarios where the image registry is local and private. Additionally, unlike computing resources, IONOS cloud providers offer limited and non-configurable network resources, which further contributes to the increased time required to push the image in multi-cluster migrations compared to single-cluster scenarios.

Another observation is that the different phases of saving a container's state are generally influenced by hardware resources, the registry location, and the size of the state. However, the type of application does not appear to be a decisive factor, as all three databases exhibit similar patterns across every phase of the saving process. This means that factors like the base image size will not affect the service container saving time, only the newly injected data will do. This finding is valuable because it simplifies the design and optimization of systems handling container state management. Since the application type has minimal impact, developers and operators can focus on optimizing hardware resources, registry placement, and state size without needing to tailor solutions for specific database types.

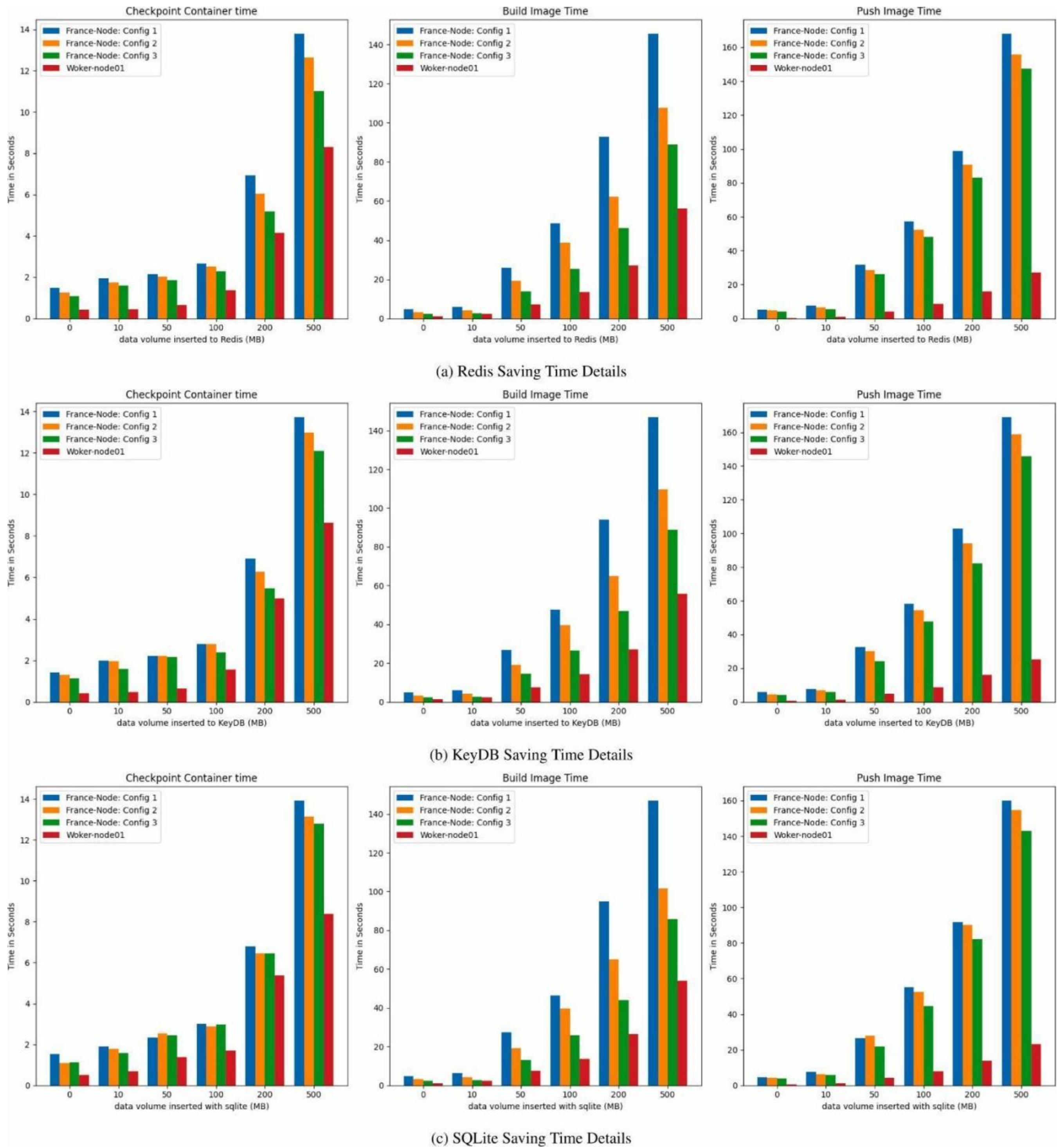


Figure 18: Average time, required to complete checkpointing, image creation, and image pushing processes for different container sizes and container images on the source computing nodes in both migration scenarios

6.1.11.2 Rescheduling Time

After saving the container state, the second phase in the migration process involves assigning the pods hosting the saved containers to new, healthy nodes. This process, known as rescheduling the pods, occurs at two levels: first, by choosing the cluster, then by selecting the computing node based on their resource availability scores. In our tests, the time taken to reschedule the application was negligible, typically in the tens of milliseconds. However, these tests were conducted on a relatively small testbed, so the results may not be representative of larger environments. Testing a multi-cluster scheduling solution across many clusters and nodes would require substantial hardware resources that we do not currently possess. To address this limitation and demonstrate the effectiveness of our solution in larger environments, we used simulation. we modeled an infrastructure consisting of 100 clusters, each containing between 5 and 10 computing nodes. We assumed that the nodes were identical in terms of the resources they provide. Initially, the percentage of available resources (ranging from 20% to 80% for both CPU and memory) was assigned randomly at the start of each experiment. Our goal was to compare our solution with two alternatives: one that selects the best available node across the entire infrastructure and another that performs random scheduling. The comparison focused on three key metrics: scheduling time, latency between microservices, and the percentage of utilized clusters in the total infrastructure. We varied the number of microservices from 50 to 150, with each microservice having specific resource and availability requirements. Each microservice application was modeled as a Directed Acyclic Graph (DAG), where nodes represented microservices, and edges represented the connections required between them.

The infrastructure was represented as a weighted graph, with nodes representing the computing nodes and edges representing latency between clusters. For latency values we used abstract numerical values for measurement, latency between connected clusters were varied between 1 and 5, and for infrastructures without direct links, the latency was calculated as the shortest path between clusters. We assumed that nodes within the same cluster could directly access one another, with an inter-node latency of 0.1. The latency between two microservices was equal to the latency between the nodes that hosted them, meaning that if two microservices were hosted on the same node, the latency between them would be 0. The overall latency of the application was determined by the sum of the latencies between each pair of connected microservices.

The results presented in this section are the averages of 1000 repetitions of our experiments. Figure 19 illustrates the outcomes of these experiments. Figure 19a represents the scheduling time required to schedule microservice applications with different configurations using the three solutions. The figure shows that the optimal solution takes more time to complete the scheduling compared to both our solution and the random solution. This is because, for each microservice, the optimal solution must search for the most available node across the entire federated infrastructure, which increases the search space. In contrast, our approach first selects the cluster and then schedules as many microservices as possible within that cluster, reducing the search space to the set of the cluster nodes. While the optimal solution ensures that microservices are hosted on nodes with the highest available computing resources, this approach can become time-consuming depending on the number of microservices and the size of the infrastructure. The random solution, on the other hand, is the least time-consuming, as it randomly selects nodes from all those that meet the application's resource requirements.

Figure 19b represents the overall application latency after being scheduled using the solution that seeks the optimal node, our solution, and the random algorithm. We observe that the latency is significantly lower with our solution compared to the two other alternatives. This is because our solution prioritizes scheduling as many

microservices as possible within the same cluster, which helps minimize the inter-cluster penalty associated with connecting microservices.

Figure 19c shows the percentage of used clusters after scheduling the microservices using the three solutions. We can observe that our solution limits infrastructure usage for the same reason—maximizing the number of microservices hosted within the same cluster. In contrast, the optimal solution searches for the best node regardless of its location, which leads to a higher percentage of clusters being used.

To summarize, we believe that the two-tier scheduling solution strikes a good balance between selecting nodes with high availability to host the microservices and minimizing scheduling time, which is much closer to the random approach that imposes no additional constraints. Additionally, our solution results in a lower percentage of used clusters and provides the minimum latency for the scheduled application compared to both other alternatives.

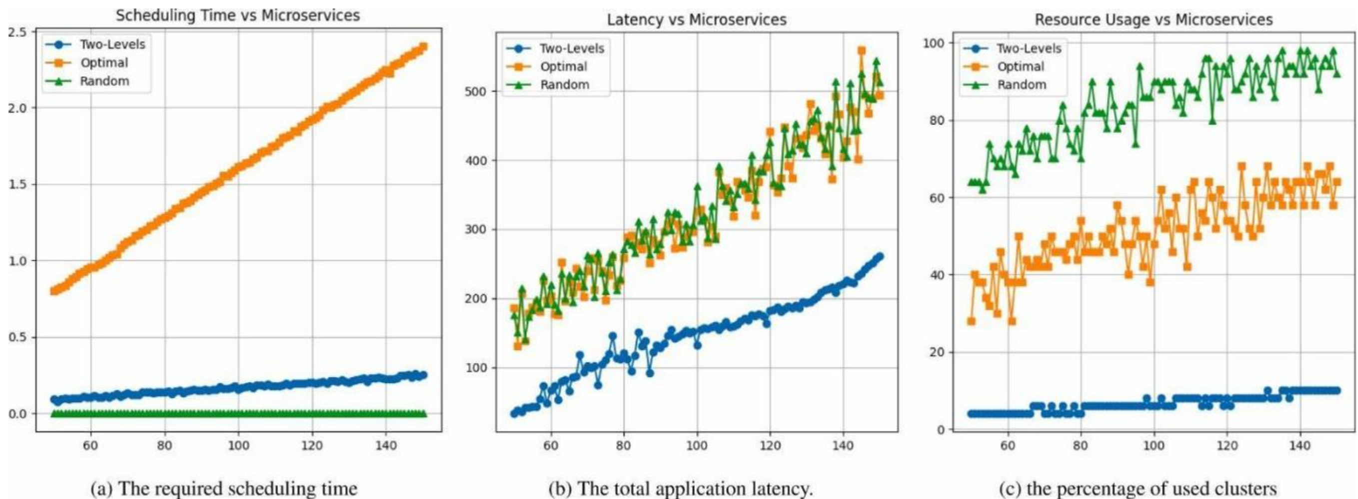


Figure 19: Multi-Clusters Scheduling techniques comparison

6.11.1.3 Restore Time

Figure 20 shows the average restore time for the three in-memory databases used in our experiments, measured from node selection to full container operation. Key observations include:

- Larger container states increase restore time due to additional data transfer,
- Disk performance has a minor impact compared to its role in checkpointing,
- Single-cluster migrations restore faster since local registries reduce latency, whereas multi-cluster scenarios require remote registry access, adding delay,
- SQLite restores slower than Redis and KeyDB due to fetching both the checkpointed and base images, with larger base images increasing pull time, though this is skipped if already present on the node.

These insights highlight the importance of scheduling policies that prioritize infrastructure already hosting the service, minimizing restore delays. For consistency, we cleared images before each test iteration to eliminate residual data effects.

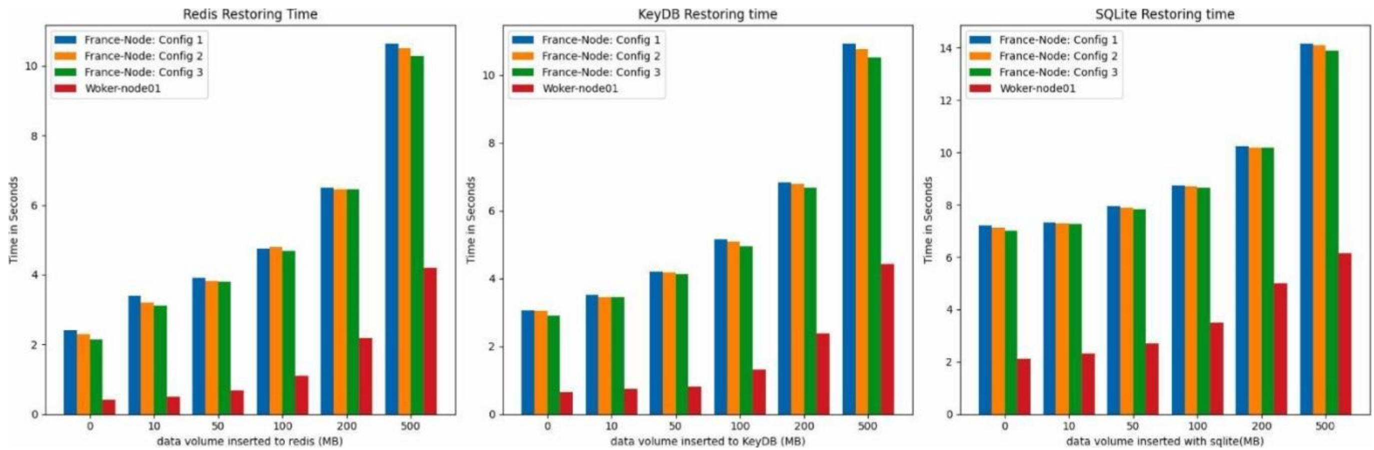


Figure 20: Application restore times with varying disk configuration

6.2 Reinforcement Learning Cache-aided Service Migration Algorithm

6.2.1 Introduction

MEC has emerged as a vital architecture for delivering low-latency, high-quality services by bringing computation and storage closer to mobile users. This proximity reduces round-trip time for data transmission and enhances responsiveness, which is crucial for applications such as augmented reality, real-time gaming, and autonomous driving. However, achieving consistent service quality in MEC environments is challenging due to user mobility, which necessitates effective task migration and resource allocation across multiple edge nodes [18]. To address this, proactive migration strategies have been extensively explored, aiming to preemptively transfer tasks to nearby edge nodes, as users move across coverage areas, thereby minimizing handover delays and service disruptions [19][20]. These studies demonstrate the benefits of proactive migration in maintaining service continuity but often rely on accurate trajectory predictions, which can introduce computational overhead and inaccuracies in dynamic environments [21],[22].

In parallel, the migration process can benefit significantly from effective caching mechanisms, which ensure that frequently accessed content is stored locally on edge nodes. By caching popular content near the user, service relocation can be performed more smoothly, as cached data reduces the need to re-fetch resources from distant servers, thus minimizing migration-induced delays. Furthermore, as MEC shifts towards cloud-native architectures [23], application services are increasingly designed to be stateless, meaning they do not store session information locally. In such architecture, caching plays a critical role by managing data locality independently of the service state, allowing cloud-native, stateless services to scale dynamically across edge nodes while ensuring efficient, latency-sensitive data access. Caching policies based on content popularity models, particularly those using LFU distribution, have shown effectiveness in optimizing cache utilization and reducing core network congestion. Motivation: User mobility in MEC environments inherently induces frequent handovers as users move across service areas, triggering fluctuations in workload on edge nodes and potential service disruptions. Here, rather than predicting the user's path dynamically, the knowledge of the user's future location is used to prepare edge nodes proactively. Such an approach can minimize the delay associated with node handovers, especially for applications that demand uninterrupted service. Existing solutions [18], [23] often rely on either proactive migration or edge caching but lack an integrated approach that jointly optimizes latency and resources by leveraging both methods simultaneously. The main contributions of the proposed solutions are as follows:

- A dual approach combining service placement designed to optimize both latency and resource utilization with an LFU content caching policy at the edge nodes,
- A solution for managing the migration of services in a multi-user environment based on mobility patterns,
- Development of a cache manager module to optimize data fetching for various services,
- Service placement is determined by resource availability and cache memory usage,
- Mobility pattern prediction to initiate the migration process for different services, considering resource availability and latency,
- The solution has been tested in a realistic network simulator implementing 5G protocols.

6.2.2 Migration cache-aided architecture

In the environment, there are five layers that we are considering for the proper analysis and design of the migration cache-aided RL algorithm. Each of the layers involved in the environment are illustrated in Figure 21, and described in the following subsections. The Migration architecture is inside the application & resource management in the AI-Based LCM module. In the Migration cache-aided architecture, the agent's action is transferred to the LCM to start the migration into the cloud edge continuum.

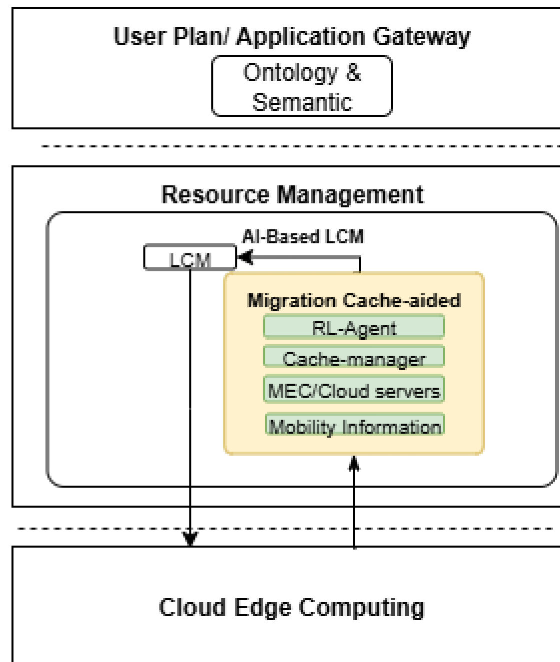


Figure 21: Migration cache-aided architecture

6.2.2.1 Mobility

In this layer, we find the mobility information of the different types of end users. The users are bicycles, pedestrians, vehicles, and IoT devices. Each user will request a specific service that will be deployed in the Cloud Edge Computing Continuum. Those services we defined as Augmented reality, location, message request and response, and smart home data sensor.

6.2.2.2 MEC/Cloud Servers

It consists of gNBs, MEC server, cloud servers, and additional cache memory information. During placement and migration, the services request the CPU, disk, RAM, and cache memory. Later, the data for the agent learning is extracted and shared with the next layer for cache management, and then with the ML/AI layer.

6.2.2.3 Cache Management

In this particular layer, the new Cache manager is responsible for fetching and allocating the data requested by the service into the MEC server cache memory. This is followed by the eviction policy that we have chosen to be Least Frequently Used (LFU).

6.2.2.4 Machine Learning/Artificial Intelligence

Here, the RL agent receives all the state parameters—mobility patterns, resource requirements, and cache status—to learn how to perform migration efficiently. By including cache memory state in the decision-making process, the RL agent can avoid unnecessary data transfers from remote servers and instead use locally cached content. This cache-aided learning approach reduces service interruption time and improves overall system responsiveness compared to conventional RL migration methods.

6.2.3 Workflow

The migration flow can be found in Figure 22. The algorithm functions by starting a migration trigger time to look after the users' needs while they are moving. After the migration timer starts, the migration app requests the agent to inform if a migration is needed for the service. For the agent to respond, it first requests information from the cache manager to understand the memory used. Following this step, the agent decides whether the response is positive or negative. If it is negative, the service stays in the same Edge server. On the contrary, the process is initiated if the migration response is positive.

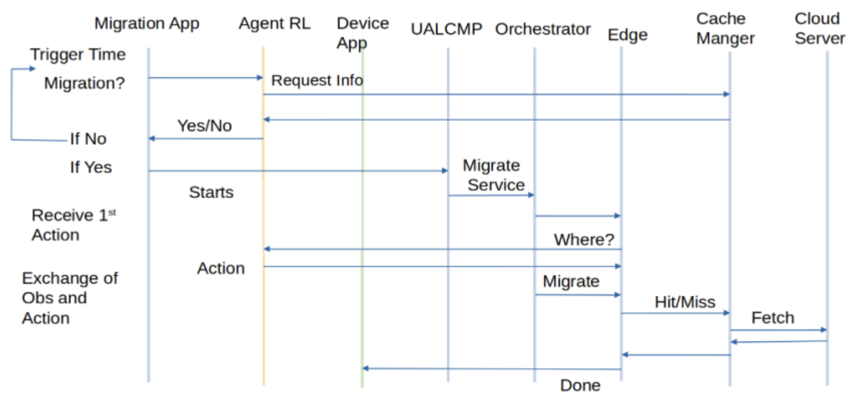


Figure 22: Migration process workflow

Then, the migration is requested to the edge server, the service is migrated, the cache memory is updated if a hit, and then the migration is done. If there is a miss, the migration time is higher, and the data must be fetched from the cloud server. After the data is downloaded, the service starts, and the migration is finished. The idea of using a cache manager is to generate Hits and avoid increases in the fetching steps while migrating a service.

© AC3 2023

6.2.4 Performance evaluation

In this section, the simulation setup is described and the results are analyzed to illustrate the efficiency of the novel RL algorithm proposed in this use case. To demonstrate the enhancement produced by the cache-aided approach, the evaluation of the results was performed in two KPIs: latency and migration time.

6.2.4.1 Simulation Setup

Since this setup is novel and showcases a scenario with multiple MECs, the initial testing of the RL algorithm was carried out in a simulation environment. For this purpose, the OMNeT++ network simulator, together with the Simu5G library, was employed. The setup included several MECs where services were migrated as users moved across the network. A visualization of the environment is shown in Figure 23.

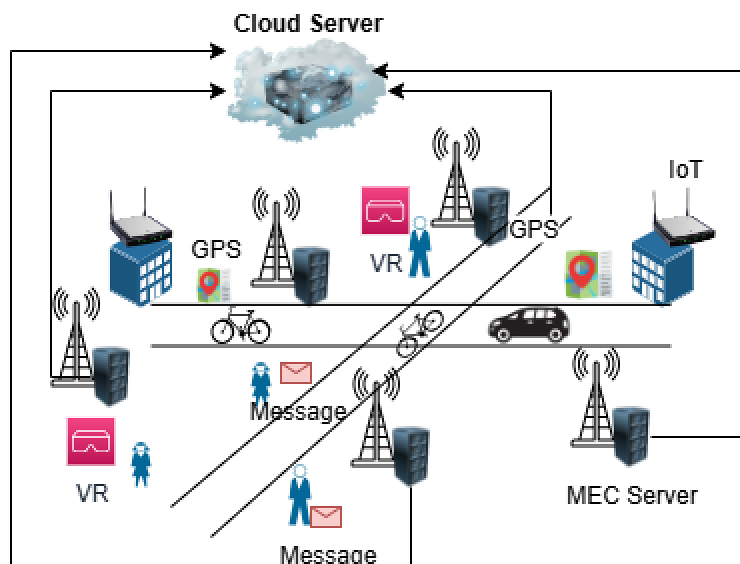


Figure 23: Network Environment

By following this strategy, the simulation environment closely resembles real-world conditions. OMNeT++, implemented in C++, was used for network simulation, while the RL algorithm was developed in Python to enable effective interaction. A REST API was designed to facilitate communication between the network simulator and the RL agent. Specifically, the API allows the RL agent to obtain environment observations from the simulator and respond with actions derived from its policy (see diagram in Figure 24.) The simulation configuration for the network and RL algorithm are listed in.

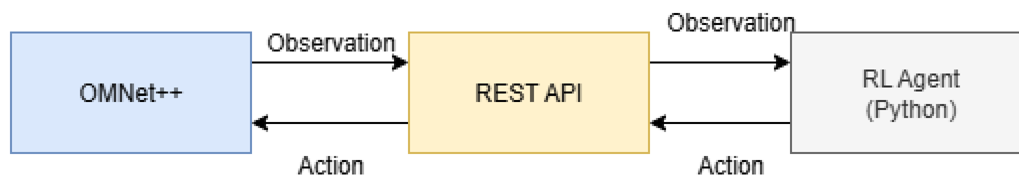


Figure 24: Communication Diagram of the different platforms

Table 6: Simulation Setup for Migration Environment

Parameter	Value
Tile Dimension	1000x700 m^2
Wireless Network	5G
Episodes	1400
e-decay	1 to 0.01
Discount factor γ	0.99
Learning rate α	0.001
NN hidden layers	2
Number of Edge Servers	5
Number of Users	15-50
CPU speed MEC server	1GIPS
RAM MEC server	16GB
Disk MEC server	500GB
Cache Memory	1000MB

6.2.4.2 Result Analysis

In Figure *25*, we present the latency comparison among all migration policies: Random, Available, and RL_with_Cache. Among them, the Random policy demonstrated the most stable latency; it consistently supports a lower and nearly constant number of applications across all MEC servers compared to RL_with_Cache, which is not desirable. Furthermore, Figure 26 shows that Random results in higher migration times on most MEC servers, indicating that its decisions do not optimize for migration efficiency.

These observations suggest that although Random appears stable, it lacks adaptability and does not account for network dynamics. In contrast, the agent Rl_with_cache adapts to the environment and achieves lower latency in 60% of the MEC servers, with an average latency reduction of approximately 20%. When compared with other policies, Available, Location, and RL_without_Cache in approximately 80%, 100%, and 60% of the MEC servers, respectively. of the MEC servers, respectively, confirming its effectiveness in reducing latency through informed, adaptive decision-making.

© AC3 2023

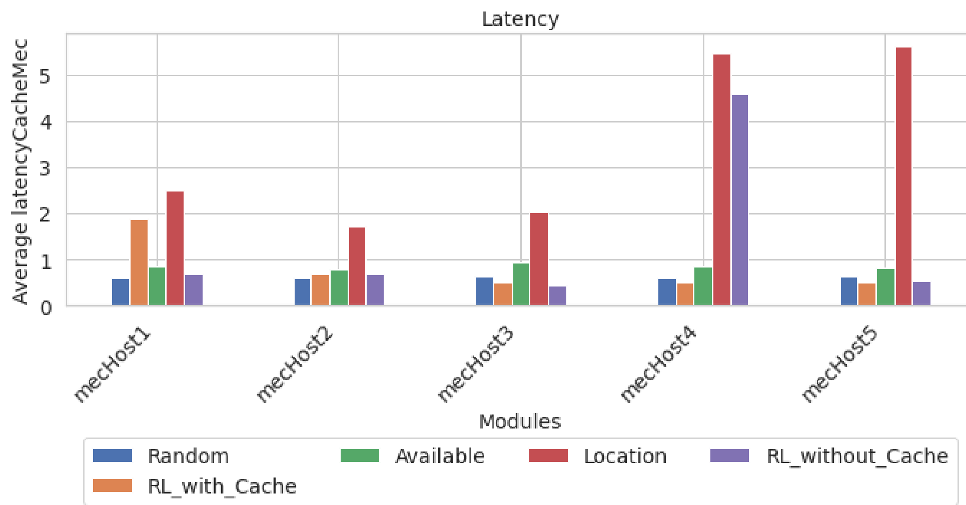


Figure 25 Latency comparison between different migration policies

For the migration time analysis, it displays the time per user type, which is the time experienced by the users as the service must be migrated to another server. As observed, the RL_with_cache demonstrated to perform with a lower value for 80% of the types of services. The other 20% corresponds to the UE type of user. It is a compromise for the agent. Another important fact is that for pedestrian random, available, and location policies, the values of migration are higher than the RL agents. This indicates that the services with lower mobility patterns produce higher migration time, compared to the RL algorithm that enhances the migration time by learning the mobility patterns and cache information.

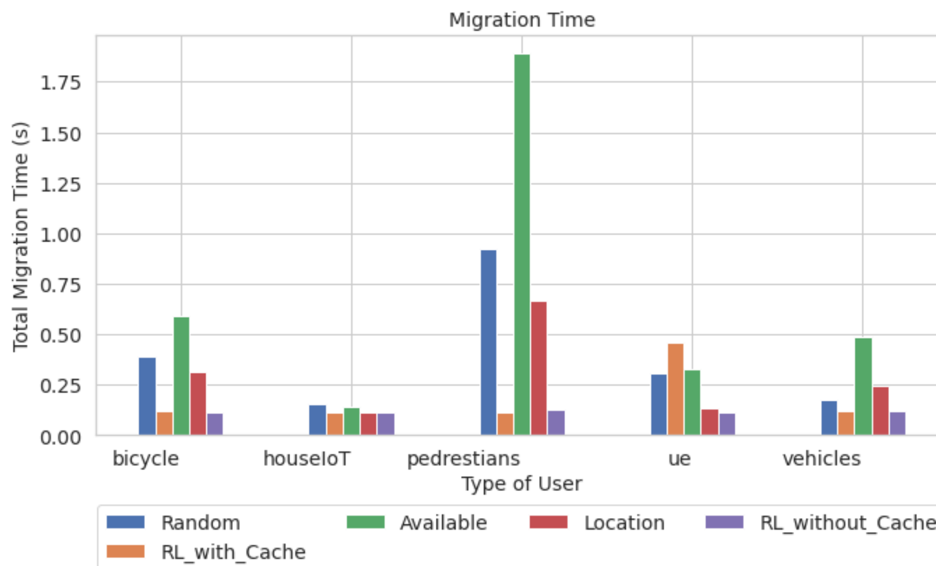


Figure 26: Migration comparison between different migration policies

In terms of migration time (Figure 26), the RL_without_Cache method performs slightly better, showing a reduction of migration time for the bicycle and UE of approximately 8% and 74%, respectively, when compared to RL_with_Cache. For the house, IoT, and vehicles modules, both policies yield nearly identical results, with negligible differences below 1%. However, in the pedestrians module, RL_with_Cache clearly outperforms RL_without_Cache, reducing migration time by approximately 10%. While these results show mixed outcomes in migration time alone, a broader evaluation that includes latency and number of running applications strongly favors RL_with_Cache. Specifically, RL_with_Cache achieves up to 20% lower latency in 60% of MEC servers and maintains a higher number of active applications across 40% of them. This indicates that RL_with_Cache not only maintains competitive migration times but also ensures lower latency and better resource utilization, ultimately reflecting more intelligent and adaptive decision-making under dynamic network conditions.

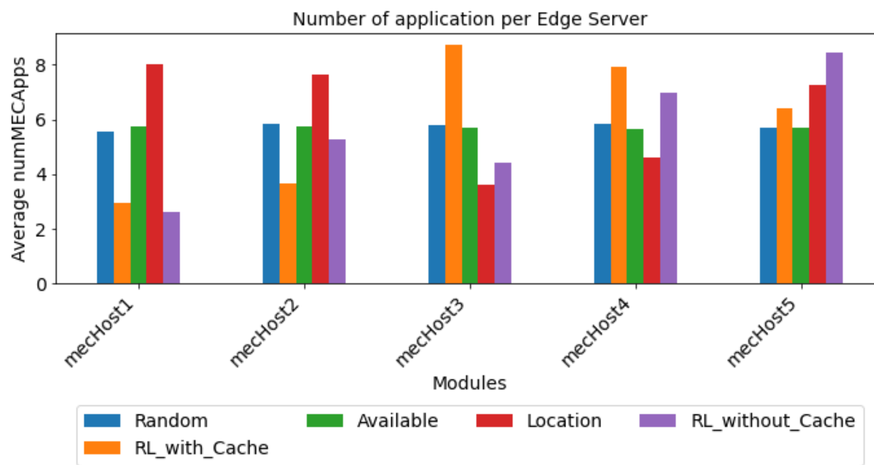


Figure 27: Average number of apps per edge server.

The novel RL solution that considered the cache information has demonstrated efficiency while migrating services within network edge servers. The results showed an improvement in migration time in the network while maintaining the other KPIs within an acceptable performance, such as latency and number of applications per edge server (Figure 27). Furthermore, it demonstrated superiority in comparison with other migration policies, such as Location, availability, and random allocation.

7 Conclusions

This deliverable presents a comprehensive overview of the initial implementations and experimental validations related to Lifecycle Management (LCM) of microservices-based applications within the Cloud-Edge Computing Continuum (CECC). The outcomes demonstrate significant progress towards realizing automated, intent-driven, and SLA-aware deployment mechanisms, supported by robust semantic reasoning and AI-driven profiling methods.

Specifically, the developed SLA and intent-based models successfully translated user-defined performance requirements into actionable deployment strategies. These strategies effectively guided resource provisioning decisions across heterogeneous infrastructure, addressing performance metrics such as latency, throughput, and service availability. Furthermore, extensions to existing application composition models were effectively implemented, integrating critical data management information required by modern microservice architecture.

The introduced application profiling techniques, enhanced by machine learning algorithms, proved effective in predicting application behavior dynamically, enabling proactive adaptation to fluctuating workloads and conditions. This capability significantly improves operational efficiency and enhances the responsiveness of resource management decisions within CECC environments.

Moreover, the algorithms addressing the migration of stateful microservices demonstrated promising results, successfully achieving a balanced trade-off between strict SLA adherence and efficient utilization of infrastructure resources. The innovative use of predictive AI techniques to manage microservice image transfers highlighted potential strategies for reducing downtime and improving resource usage at edge and far-edge nodes.

The findings and methodologies detailed in this report provide a robust foundation for integration activities in Work Package 5 (WP5), which focuses on system-level integration, validation, and orchestration across the AC3 framework. Future work within WP5 will leverage the outcomes of this deliverable to ensure seamless interoperability between lifecycle management components and other management modules, thereby advancing the project’s vision for intelligent, autonomous application lifecycle management in federated cloud–edge infrastructures.

© AC^3 2023

8 References

[1] F. Shaman, B. Ghita, N. Clarke, and A. Alruban, ”User Profiling Based on Application-Level Using Network Metadata,” in 2019 7th International Symposium on Digital Forensics and Security (ISDFS), Barcelos, Portugal, pp. 1-6, Jun. 2019.

[2] A. Ferrari, E. Riforgiato, and L. Roffia, ”JSON SPARQL Application Profile for Linked Data,” in 2022 31st Conference of Open Innovations Association (FRUCT), Helsinki, Finland, pp. 1-10, Apr. 2022.

[3] A. López-Acosta, A. García-Hernández, S. Vázquez-Reyes, and A.Mauricio-González, ”A Metadata Application Profile to Structure a Scientific Database for Social Network Analysis (SNA),” in 2020 8th International Conference in Software Engineering Research and Innovation (CONISOFT), pp. 208-214, 2020.

[4] K. Ray, A. Banerjee, N.C. Narendra, Proactive microservice placement and migration for mobile edge computing, 2020 IEEE/ ACM Symp. Edge Comput.

[5] P.S. Junior, D. Miorandi, G. Pierre, Stateful container migration in geo-distributed environments, 2020 IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom) 4 (2020) 9–56, http://dx.doi.org/10.1109/CloudCom49646.2020.00005.

[6] P.S. Junior, D. Miorandi, G. Pierre, Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes, 2022 IEEE 6th Int. Conf. Fog Edge Comput. (ICFEC) 2 (2022) 6–33, http://dx.doi.org/10.1109/ICFEC54809.2022.

[7] M.-N. Tran, X.T. Vu, Y. Kim, Proactive stateful fault-tolerant system for kubernetes containerized services, IEEE Access 10 (2022) 102181–102194, http://dx.doi.org/10.1109/ACCESS.2022.3209257.

[8] H. Schmidt, Z. Rejiba, R. Eidenbenz, K.-T. Förster, Transparent fault tolerance for stateful applications in kubernetes with checkpoint/restore, 2023 42nd Int. Symp. Reliab. Distrib. Syst. (SRDS) 12 (2023) 9–139, http://dx.doi.org/10.1109/SRDS60354.2023.00022.

[9] L. Abdollahi. Vayghan, M.A. Saied, M. Toeroe, F. Khendek, A Kubernetes controller for managing the availability of elastic microservice-based stateful applications, J. Syst. Softw. 175 (2021) 110924, http://dx.doi.org/10.1016/j.jss. 2021.110924.

[10] J.-H. Na, et al., PVA: The persistent volume autoscaler for stateful applications in Kubernetes, IEEE Access 12 (2024) 179130–179143, http://dx.doi.org/10.1109/ACCESS.2024.3507194.

[11] P. Bellavista, S. Dahdal, L. Foschini, D. Tazzioli, M. Tortonesi, R. Venanzi, Kubernetes enhanced stateful service migration for ML-driven applications in industry 4.0 scenarios, 2024 IEEE Annu. Congr. Artif. Intell. Things (AIoT) 2 (2024) 5–31, http://dx.doi.org/10.1109/AIoT63253.2024.00015.

[12] P. Bellavista, S. Dahdal, L. Foschini, D. Tazzioli, M. Tortonesi, R. Venanzi, Kubernetes enhanced stateful service migration for ML-driven applications in industry 4.0 scenarios, IEEE Annu. Congr. Artif. Intell. Things (AIoT), Melb. Aust. 2024 (2024) 25–31, http://dx.doi.org/10.1109/AIoT63253.2024.00015.

[13] A. Meliani, M. Mekki, A. Ksentini. ”Resiliency focused proactive lifecycle management for stateful microservices in multi-cluster containerized environments.” *Computer Communications*, 2025, p. 108111.

[14] https://www.youtube.com/watch?v=wLnlEQ1wy54

[15] https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/

[16] https://buildah.io/

[17] https://quay.io/

[18] Z. He, L. Li, Z. Lin, Y. Dong, J. Qin, and K. Li, “Joint optimization of service migration and resource allocation in mobile edge–cloud computing,” Algorithms, vol. 17, no. 8, p. 370, 2024.
[19] Q. Yuan, J. Li, H. Zhou, T. Lin, G. Luo, and X. Shen, “A joint service migration and mobility optimization approach for vehicular edge computing,” IEEE Transactions on Vehicular Technology, vol. 69, no. 8, pp. 9041–9052, 2020.
[20] P. Wang, T. Ouyang, G. Liao, J. Gong, S. Yu, and X. Chen, “Edge intelligence in motion: Mobility-aware dynamic dnn inference service migration with downtime in mobile edge computing,” Journal of Systems Architecture, vol. 130, p. 102664, 2022.
[21] D. Sabella, A. Li, H. Lee, L. Cominardi, Q. Huang, E. Pateromichelakis, V. Kashyap, C. Costa, F. Granelli, W. Featherstone et al., “Mec support towards edge native design,” White Paper, no. 55, 2023.
[22] P. Liu, G. Xu, K. Yang, K. Wang, and X. Meng, “Jointly optimized energy-minimal resource allocation in cache-enhanced mobile edge computing systems,” IEEE Access, vol. 7, pp. 3336–3347, 2018.
[23] S. Wang, J. Xu, N. Zhang, and Y. Liu, “A survey on service migration in mobile edge computing,” IEEE Access, vol. 6, pp. 23 511–23 528, 2018.

© AC3 2023