Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Semantic Translation of User Intent into Kubernetes Placement Constraints for AI Workflows

Efthymios Chondrogiannis

timchros@gmail.com

Business and IoT Integrated Solutions (Bi2S) Ltd

Evangelos Psomakelis

Business and IoT Integrated Solutions (Bi2S) Ltd

Efstathios Karanastasis

Business and IoT Integrated Solutions (Bi2S) Ltd

Research Article

Keywords: Intent-based orchestration, Ontology-driven deployment, ML-driven orchestration, Kubernetes, Helm chart, Cloud Continuum

Posted Date: November 16th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11541809/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Semantic Translation of User Intent into Kubernetes Placement Constraints for AI Workflows

Efthymios Chondrogiannis[1][0000-0003-3865-2277], Evangelos Psomakelis[1][0000-0002-2342-3626] and Efstathios Karanastasis[1][0000-0003-0460-4648]

[1] Business and IoT Integrated Solutions (Bi2S) Ltd, Kypranoros 13, 1061, Nicosia, Cyprus
{timchros,vpsomak,karanastasis}@gmail.com

Abstract. Modern AI applications are increasingly deployed across heterogeneous cloud–edge infrastructures, where each microservice in a workflow may require different resources and has different performance characteristics. Traditional container orchestration systems (e.g., Kubernetes) do not natively support either expressing high-level user goals or automatically translating them into concrete resource-aware deployment decisions. This paper presents CorMOS, a lightweight, intent-driven orchestration mechanism that enriches container deployment descriptors using ontology-based semantic knowledge. CorMOS introduces an OWL ontology for modelling workload types and user goals and applies semantic reasoning together with ML techniques to infer missing resource specifications and placement preferences. These inferred constraints are automatically injected into Helm charts and Argo Workflows, enabling Kubernetes to schedule microservices on the most appropriate cluster nodes. CorMOS was developed and evaluated both on a heterogeneous k3s testbed and within the NEMO Open Call 1 environment, where it was integrated into the NEMO orchestration pipeline. Experimental results show that CorMOS can autonomously complete incomplete workload descriptors, infer values for up to 50% of missing resource parameters while achieving 80% estimation accuracy, and incur low processing overhead while sustaining a throughput of approximately 20 charts/min. These findings demonstrate the effectiveness of semantic enrichment in reducing manual configuration effort while enabling intent-driven deployment with low computational overhead.

Keywords: Intent-based orchestration, Ontology-driven deployment, ML-driven orchestration, Kubernetes, Helm chart, Cloud Continuum.

1 Introduction

Modern Artificial Intelligence (AI)-based applications are increasingly being deployed across distributed, heterogeneous infrastructures spanning cloud, edge and far-edge devices (collectively referred to as the cloud continuum). This shift is driven by the need to reduce latency, improve energy efficiency, comply with data locality constraints, and support continuous model refinement close to the source of data [1]. Consequently, workflows are increasingly decomposed into multiple microservices

(e.g., data preparation, model training, and inference), each of which may have distinct resource requirements [2]. Traditional container orchestration platforms (e.g., Kubernetes, Docker Swarm) provide scheduling mechanisms based on explicitly defined resource requirements and placement constraints. However, they do not natively support translating high-level user intents (e.g., prioritising energy efficiency or favouring edge execution) into the low-level resource specifications and scheduling policies required by the orchestrator.

To address this gap, there is growing interest in semantic-aware and intent-based orchestration [3,4], where user goals can be specified in a platform-independent manner and then automatically translated into concrete deployment actions. In this context, we present CorMOS, a lightweight orchestration mechanism designed to bridge intent expressions and concrete deployment artifacts for containerised AI workloads. CorMOS introduces an ontology-based abstraction layer to formally describe workload characteristics and user preferences, combining semantic reasoning with machine learning (ML) to infer resource specifications and placement constraints and uses this knowledge to enrich Helm charts. Furthermore, runtime monitoring data collected during microservice execution are used to iteratively refine the inferred resource constraints. In this way, the engine can automatically derive executable deployment descriptors without requiring users to manually specify low-level scheduling details. CorMOS was developed as part of the NEMO[1] Open Call 1 (OC1) programme, which allowed us to test the system not only on our own heterogeneous k3s cluster but also within the wider NEMO experimental environment in combination with other orchestration components and external data-management services. This dual evaluation context enabled us to assess both the functionality of CorMOS in isolation and its behaviour when integrated into a larger orchestration workflow. The results indicate that lightweight semantic reasoning can remove a significant portion of manual configuration effort, and that intent-driven enrichment can be integrated into an existing orchestration pipeline with minimal overhead.

The main contributions of this work are threefold. First, we introduce an OWL-based semantic representation that formally captures the relationships between application components, workload characteristics, resource requirements, and high-level user deployment objectives, providing a structured foundation for intent-driven orchestration. Second, we develop a semantic deployment enrichment mechanism that combines ontology-based reasoning, predefined mapping rules, and machine learning techniques to infer missing resource specifications and derive placement constraints. These constraints are subsequently translated into Kubernetes-compatible deployment configurations through the automatic enrichment of Helm charts. Third, we implement and experimentally evaluate the proposed approach on a heterogeneous k3s cluster and demonstrate its interoperability with the NEMO orchestration environment, assessing its ability to complete partially specified workload descriptors, estimate resource requirements, and introduce limited processing overhead into the deployment pipeline.

[1] Next Generation Meta Operating System (NEMO), https://meta-os.eu/

The remainder of this paper is structured as follows. Section 2 summarises related work on container orchestration frameworks and AI-based approaches for resource orchestration. Section 3 describes the CorMOS methodology, including the layered architecture, the abstraction models, the ontology and the orchestration components developed. Section 4 presents the evaluation scenarios and experimental results, both on our local testbed and within the NEMO environment. Finally, Section 5 concludes the paper and discusses the main lessons learned.

2 Related Work

2.1 Kubernetes and Related Orchestration Frameworks

Container orchestration systems have become essential for managing modern software applications [5]. This is due to their capability to automatically or semi-automatically deploy, scale and manage containerised applications across diverse infrastructures. Among these, Kubernetes[2] stands out as the most widely adopted framework [6], offering a comprehensive orchestration platform. Its widespread adoption is largely due to features such as automated rollouts and rollbacks, service discovery, load balancing, self-healing, and its design for extensibility that allows for customisations and additional features. Kubernetes deployments commonly rely on package management systems for application lifecycle management. The most common one is Helm[3], which enables the definition, installation and upgrade of applications, using reusable templates, known as Helm charts [7].

Several other frameworks exist that complement or extend Kubernetes in various ways. Docker Swarm[4], for example, is a popular alternative that provides a container orchestration solution. Simpler than Kubernetes, it is part of the Docker ecosystem and integrates seamlessly with Docker containers [8]. Similarly, OpenShift[5], which is built on Kubernetes, enhances the process of orchestration through additional enterprise-grade features, such as integrated CI/CD pipelines, advanced security policies and certified databases with related services [9]. For specialised workloads, particularly Directed Acyclic Graph (DAG) tasks, that require executing sequences of computational steps on a cluster, frameworks that focus on machine learning (ML) and extract, transform, load (ETL) workflows exist. Kubeflow[6] is built on top of Kubernetes as a containerised ML platform, providing an abstraction for developing, deploying and managing every step of the ML lifecycle on a Kubernetes cluster [10]. A Python-based platform, Airflow[7] provides a web-based user interface (UI) which simplifies the development and monitoring of workflows. MLflow[8] is an open-source

[2] Kubernetes(K8s), https://kubernetes.io/
[3] Helm, The Package Manager for Kubernetes, https://helm.sh/
[4] Docker Swarm Mode, https://docs.docker.com/engine/swarm/
[5] Red Hat OpenShift, https://www.redhat.com/en/technologies/cloud-computing/openshift
[6] Kubeflow, https://www.kubeflow.org/
[7] Apache Airflow, https://airflow.apache.org/
[8] MLflow, https://mlflow.org/

framework-agnostic platform compatible with a wide range of ML libraries. It can be integrated for tracking ML experiments and deploying models in Kubernetes environments. Lastly, Argo Workflows[9] manages workflows and pipelines in a declarative manner using YAML files. It combines a workflow engine with artifact management to achieve communication between the different steps of a flow.

The practical application and evaluation of these frameworks have been discussed in many studies, demonstrating their effectiveness across a range of application domains. Professionals and researchers utilise these tools to address the rising challenges of ML development on distributed systems, including scaling and complex workflow management in containerised environments. The authors of [11] for example, explored in detail the process of deploying ML models on Kubernetes using Kubeflow. Their study showcased how Kubeflow overcomes established MLOps problems, improves the training and serving of ML models and standardises the containerisation of ML workflows. George and Saha [10] further argue that the entire ML lifecycle should be integrated into a software development lifecycle (SDLC), supported by continuous integration and continuous deployment (CI/CD) practices. Frameworks such as Kubeflow facilitate this integration, allowing practitioners to focus on model development while automating operational aspects of large-scale ML deployments.

Recent studies demonstrate the growing use of Kubernetes-based workflow orchestration across scientific, industrial, and data-intensive applications. In scientific computing, Kubeflow has been applied to bioinformatics workflows that combine Kubernetes with high-performance computing, with extensions such as the Dataset Lifecycle Framework enabling native access to shared storage [12]. Similarly, in high-energy physics, Kubeflow has supported scalable deep-learning pipelines for particle-jet energy calibration, improving hyperparameter optimisation and model performance [13]. Argo has also been used for scientific and industrial workflows, including cost-aware execution through combinations of on-demand and spot cloud instances [14], as well as integration with the Common Workflow Language to support portable workflows and flexible data-transfer mechanisms [15,16]. Beyond large-scale scientific workloads, Kubernetes has also been employed in lightweight and edge-oriented data-processing environments, supporting distributed storage, containerised services, and data-driven applications [17,18].

Despite their maturity, these platforms still require developers to manually specify deployment descriptors, resource requests, and scheduling constraints to a considerable extent. Moreover, they lack native mechanisms for translating high-level user intents into executable Kubernetes deployment artifacts through semantic reasoning and resource inference. This limitation motivates the CorMOS approach presented in this paper.

2.2 Machine Learning Techniques for Resource Orchestration

Artificial intelligence (AI) and especially machine learning (ML) has become an important enabler of intelligent resource orchestration across the cloud continuum. By

[9] Argo Workflows, https://argoproj.github.io/workflows/

learning from historical execution data, ML models can predict workload characteristics, estimate resource requirements, and support adaptive scheduling decisions that improve resource utilization and application performance. For instance, in containerised cloud settings [19], supervised learning techniques have been applied to forecast workload patterns, allowing for proactive resource allocation and scaling decisions. This predictive approach ensures that resources are allocated efficiently, reducing latency and preventing bottlenecks in data processing. Additionally, in the orchestration of ML workflows [20], supervised learning aids in automating the deployment and scaling of models across distributed cloud environments, ensuring that computational resources are utilised effectively to meet varying workload demands. Clustering algorithms, such as K-means and hierarchical clustering [21], were used by Sharma and Saini (2019) [22] to group similar nodes, enabling more efficient resource scheduling and distribution. Also, anomaly detection techniques, such as Isolation Forest [23], were employed to identify outliers and unusual patterns in resource utilisation, enabling potential performance degradation or resource overuse to be detected early and supporting proactive resource management and fault prevention, as demonstrated by Mitropoulou et al. (2024) [24].

Reinforcement learning (RL) has become a pivotal technique for resource orchestration in the cloud continuum, effectively managing the dynamic allocation of tasks across cloud, edge, and IoT devices. In this integrated environment, RL algorithms, such as Deep Q-Learning, are employed to optimise task offloading decisions, thereby balancing workloads and minimizing latency. For instance, the study [25] proposed an RL-based task-offloading solution that dynamically balances workloads and reduces overloads in the computing continuum. Additionally, Multi-Agent Reinforcement Learning (MARL) approaches have been utilised to guide the autoscaling of distributed applications, enhancing resource utilisation and application performance [26]. These RL-driven strategies can help systems to adapt to fluctuating network conditions and resource constraints across the continuum. On the other hand, RL requires substantial amounts of data, which must be continuously processed using computationally intensive methods.

Generative AI (GenAI) and large language models (LLMs) [27] have recently been explored as assistants for cloud management tasks, including infrastructure-as-code (IaC) generation, Kubernetes manifest creation, workflow specification, and operational support. For example, Angi et al. (2025) [28] propose KGen, an LLM-based approach that translates natural-language user intents into Kubernetes manifests, demonstrating the potential of LLMs to automate part of the configuration and deployment process. Also, Palavalli and Santolucito (2024) [29] investigated an LLM-based approach for generating infrastructure configurations, incorporating iterative feedback from deployment errors to improve the generated IaC artifacts. While these approaches can automate the creation of deployment descriptors, they typically generate syntactic artifacts rather than performing semantic reasoning over workload characteristics and they often rely on active user involvement to accurately capture and interpret user intent; consequently, they fall outside the scope of this work.

Overall, existing ML approaches (excluding GenAI) primarily focus on predicting resource demand, workload scheduling, autoscaling, or task offloading. However, to

the best of our knowledge, relatively few studies have investigated the combination of ML techniques and semantic knowledge representation to automatically infer Kubernetes deployment constraints from high-level user intents. CorMOS addresses this gap by integrating ontology-based reasoning with supervised learning for resource inference and deployment enrichment.

3 Methodology

3.1 CorMOS Layered Architecture

Modern applications, particularly those leveraging AI and ML techniques, are increasingly built as distributed systems composed of multiple containerised microservices. Each microservice encapsulates a specific functionality (e.g., data preprocessing, model training, inference, visualisation, etc.) and communicates with others through lightweight APIs. To ensure efficient execution, each microservice must be deployed on infrastructure that satisfies its resource requirements while also respecting the high-level deployment intents specified by the system administrator (hereafter referred to as the user). In this work, an intent denotes a high-level formal specification of user's deployment objectives at a given point in time. Since intents represents user goals, the two terms are occasionally used interchangeable throughout the paper.

The CorMOS framework provides a structured approach to orchestrating containerised applications, enabling the translation of high-level user-defined intents into precise deployment actions that generate resource-aware placement constraints across heterogeneous infrastructures. Specifically, user intents are first represented semantically using the ontology, after which semantic reasoning and ML techniques are applied to infer missing resource specifications and placement constraints. These inferred constraints are finally injected into the Helm deployment descriptors before being executed by Kubernetes.

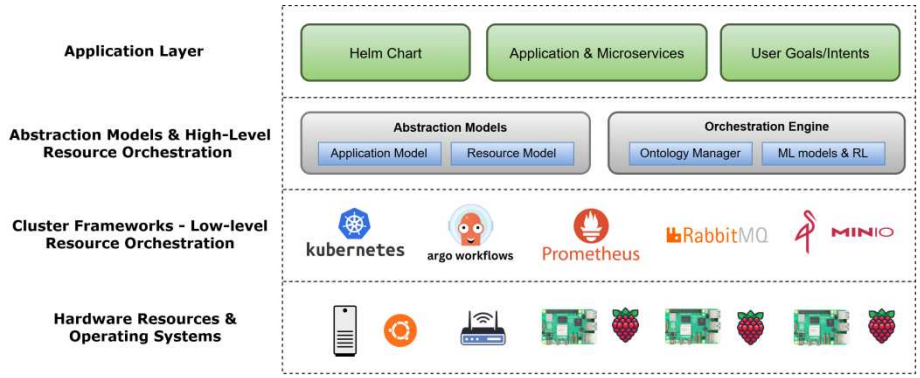


Fig. 1. CorMOS Layered Architecture.

For the efficient deployment of microservices across cluster nodes a layered architecture was followed, as shown in Fig. 1. The proposed CorMOS system architecture consists of four layers, each playing a distinct role in the orchestration process. At the topmost layer, resides the application layer. Each application is composed of one or more microservices that communicate through well-defined interfaces to form complete workflows. These applications are described using Helm charts, which provide a standardised, platform-agnostic way to define service configurations, dependencies, and resource requirements. The second layer comprises the Abstraction Models and Software Components developed within CorMOS. This layer is responsible for interpreting user-defined intents and transforming them into executable orchestration logic. It hosts the Application Model (AM) and Resource Model (RM) along with the OWL 2[10] ontology developed in this work, which jointly capture semantic relationships between high-level objectives (such as performance or energy efficiency) and low-level infrastructure requirements (such as CPU, memory, or GPU allocation). These models enable formal reasoning about workload placement, while ML techniques, including a monitoring-driven adaptation component incorporating RL principles, help infer or refine resource specifications that are subsequently transformed into executable deployment constraints by the Orchestration Engine. The third layer comprises the orchestration frameworks responsible for deploying and managing containerised applications across the cluster, including workflow-based applications composed of multiple interconnected microservices. The lowest layer comprises the physical infrastructure, such as compute nodes, network devices, and storage, which are used for the execution of workloads and the communication among the cluster nodes.

In this section, we focus on the software components and abstraction models themselves, explaining their structure, internal roles, and interactions within the CorMOS layered architecture. The specific cluster infrastructure used for evaluation, as well as the experimental scenarios executed on top of it are presented in Section 4, where the behaviour of the system is validated in realistic conditions.

3.2 Semantic Representation of Application Requirements and User Intents

The Application Model (AM) provides a platform-independent framework to define the parameters that capture the application's high-level requirements, encapsulating both holistic application-level and component-level metrics. The model captures application-level characteristics such as energy consumption, reflecting sustainability objectives, and execution time, indicating responsiveness under representative workloads. By maintaining a high level of abstraction, the AM ensures that the application can be deployed across diverse platforms without being tied to a specific underlying infrastructure. Beyond energy and execution time, the model can also capture other performance indicators like throughput, latency, and reliability metrics, allowing for a comprehensive evaluation of application behaviour.

[10] W3C OWL 2 Web Ontology Language, https://www.w3.org/TR/owl2-overview/

The Resource Model (RM) focuses on specifying the computational and infrastructural resources essential for the efficient execution of application components. It defines requirements such as CPU capacity, memory allocation, persistent storage, and network bandwidth, which are vital for ensuring that the application operates efficiently under varying workloads. The design of the RM was inspired by the Resource Model developed in the SERRANO project [30] as well as the analysis of parameters used for the description of Kubernetes nodes. For modern containerised environments, the RM can also include advanced attributes such as GPU usage for computationally intensive tasks, storage Input/Output Operations per Second (IOPS) for data-intensive applications, and network latency thresholds to support real-time operations.

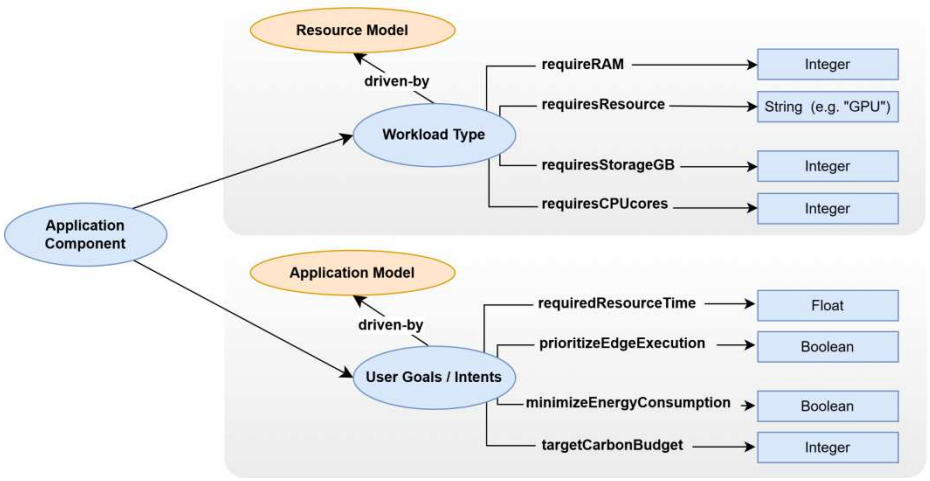


Fig. 2. A subset of the CorMOS OWL 2 ontology.

Together, the AM and RM form the semantic foundation of CorMOS and have been used in the design of the CorMOS ontology. The CorMOS ontology is an OWL-based knowledge model that formally specifies how an application component can be described along two complementary dimensions: (i) its workload type, which captures the resource-related requirements of the component, and (ii) the associated user intent, which reflects the optimisation criteria and execution constraints defined at the application level (Fig. 2). The OWL class “Workload Type” is used to specify the concrete resource parameters that are necessary for the efficient execution of an application, such as the required CPU cores, RAM, and storage capacity. In parallel, the OWL class “User Goals/Intents” is used to express objective-driven constraints and optimisation targets, such as the maximum response latency, the prioritisation of edge execution, the minimisation of energy consumption and the existence of a target carbon budget. This separation of resource requirements and user objectives enables CorMOS to reason independently about what the workload needs and what the user wants to achieve. Furthermore, the ontology remains extensible: new workload types, new resource parameters, and new goal dimensions can be added as additional individuals

or datatype properties, allowing the semantic model to evolve as new application patterns and new profiles emerge.

Based on this structure, different workloads can be described in the ontology by instantiating the classes and datatype properties with concrete values. For example, a specific application component (e.g., a data preprocessing step or an inference microservice) can be expressed as an OWL individual of type Application Component, and then annotated with a specific Workload Type along with its corresponding resource requirements (e.g., requiresCPUcores = 2, requiresRAM = 2048 MB, requiresStorageGB = 1). In parallel, the User Intents branch can also be instantiated with explicit values (e.g., prioritizeEdgeExecution = true, requiredResponseTime = 0.5s or targetCarbonBudget = 200 gCO_2). These OWL individuals encode semantic associations between user intents and workload characteristics. As a result, when the ontology reasoner is queried during orchestration time, CorMOS can infer the resource configurations that should be enforced based on the intents provided by the user. Together, the ontology, Helm charts, and Argo Workflows form an integrated orchestration pipeline in which semantic application descriptions are translated into executable Kubernetes deployments while preserving workflow dependencies and execution order.

3.3 Software Components and Background Mechanisms

The system developed consists of several software components that are responsible for managing the CorMOS ontology, application deployment and infrastructure data collection and processing. In the core of this system lies the orchestration engine that is responsible for the deployment of the application's microservices to the appropriate nodes of the cluster taking into consideration the resource requirements and user-set intents.

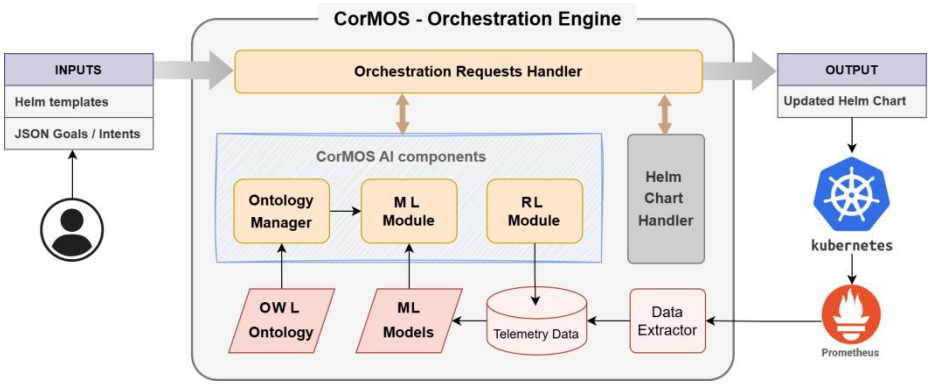


Fig. 3. CorMOS Orchestration Engine Components.

The CorMOS Orchestration Engine (Fig. 3) receives two main inputs: (i) a machine-readable description of the application (Helm chart templates including microservice

definitions) and (ii) a formalised description of the user goals, provided in JSON format, which captures the high-level intents. Based on these inputs, the Orchestration Engine, and especially the Orchestration Requests Handler component, derives enriched deployment descriptors with the support of the remaining components of the system architecture (Fig. 3) that incorporate the necessary resource constraints and placement preferences. The final output of this process is an updated Helm chart (and corresponding workflow YAML entries) that can be directly used for deployment and orchestration of the application microservices to the appropriate nodes of the cluster.

Initially, CorMOS leverages its ontology and mapping rule set as prior knowledge, represented as OWL ontology instances of an Application Component. These rules define deterministic relationships between Application Model parameters (e.g., energy consumption, throughput) and Resource Model features (e.g., CPU frequency, GPU presence, network latency). Accordingly, the Pellet reasoner [31] is employed by the Ontology Manager to infer additional statements about a particular Application Component by leveraging the hierarchy of Application Component classes defined in the OWL ontology. For instance, an application component that responds within a stringent response-time requirement is a real time application component, thereby identifying the mapping rules associated with the inferred classifications and finally selecting the one that complies with user's intent. At this stage, deployment decisions are guided by expert-defined heuristics. For example, when the user specifies an intent to minimise response latency (i.e., below a given threshold), then the orchestrator prioritises nodes with low Round Trip Time (RTT) and high CPU clock rate.

Once applications are deployed, the system continuously collects runtime metrics, such as CPU utilisation, memory consumption, queue latency, power draw, and task completion time, and stores them in the Prometheus database. These measurements are aggregated and normalised into structured datasets, which serve as training inputs for the AI models. Supervised ML techniques (including traditional classification and regression methods, such as Random Forest and Linear Regression respectively) are used for the development of ML models that can automatically learn the relationship between deployment configurations and KPIs such as latency and energy consumption, so that they can be taken into consideration by the translation mechanism. More precisely, the ML module employs suitable ML models, when available, to predict the impact of different resource configurations on the satisfaction of user intents, as expressed through the Application Model parameters, and to select the most appropriate configurations accordingly. Therefore, the ML module may derive new resource constraints or refines existing ones accordingly (i.e., those previously specified by the Ontology Manager).

Reinforcement learning operates at a higher level, treating the orchestration process as a sequential decision-making problem. The system regularly observes the current workload state, available resources, and telemetry (environment), makes the appropriate decisions regarding the placement or scheduling of microservices (action selection), and then computes a reward based on the collected data that indicate how well the resulting deployment satisfied user intents. The reward is a numerical score that is computed based on the satisfaction KPI (e.g., inverse of latency, normalised energy efficiency). To prevent the orchestration logic from stagnating on suboptimal

configurations, CorMOS maintains a dynamic balance between exploration (testing new mappings) and exploitation (using known optimal ones). For this purpose, the system occasionally (i.e., with a small probability ε, particularly after several trials have been conducted) selects a different deployment so that it can accordingly collect more information about the KPIs. Since collecting data across multiple resource configurations and developing ML models are time-consuming processes, in CorMOS these activities are repeated only when significant deviations from the specified resource constraints are observed. In such cases, the ML models are updated through a fully automated process [32] that encompasses both the selection of appropriate hyperparameters and the training of the ML models using the collected data.

The Helm Chart Handler component is responsible for updating the given Helm charts based on the new requirements identified through the previous process. Node selectors provide a strict matching mechanism based on key–value node labels (e.g., location=edge, accelerator=GPU), enabling hard binding to specific categories of nodes. In more complex cases, node affinity expresses required or preferred conditions, enabling softer placement constraints (e.g., favouring a GPU-enabled node without absolutely requiring it). These annotations are critical, as they allow Kubernetes to place each microservice on nodes whose characteristics match both the resource needs of the workload and the user's declared goals. Once deployment is triggered, the resulting workflow can be inspected via the Argo UI, while runtime performance metrics are observable through Grafana[11].

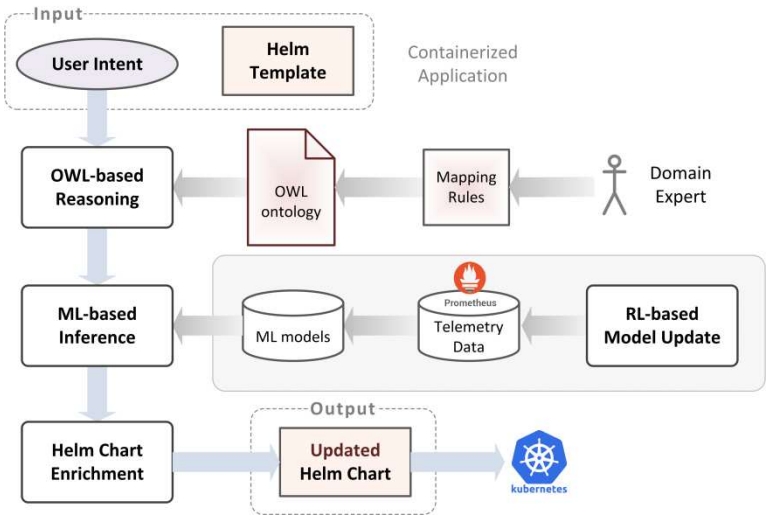


Fig. 4. Application Helm Chart Update Pipeline.

The approach followed (Fig. 4) supports continuous adaptation to variations in workload demands, network conditions, and resource availability, which is particularly

[11] Grafana, https://grafana.com/

important in dynamic IoT–Edge–Cloud environments. Through this iterative process, CorMOS continuously improves its ability to align user intents with real infrastructure performance requirements, supporting adaptive orchestration across the edge–cloud continuum. The ontology acts as a semantic scaffold, constraining the search space for AI models by filtering infeasible or irrelevant configurations. For instance, if the ontology indicates that a workload type "requires GPU", the ML module will exclude CPU-only nodes from its action space. Conversely, insights learned through ML (e.g., empirical performance distributions) can enrich the ontology by adding new mappings or updating confidence levels of existing relationships. For example, the collected data and the trained ML models may reveal specific relationships between Application Model and Resource Model parameters that remain stable under the observed operating conditions and can subsequently be incorporated into the OWL ontology. This forms a bidirectional knowledge flow between symbolic reasoning (in our case, the OWL ontology) and statistical learning (in our case, traditional ML techniques). Nevertheless, in several cases, the relationships between Application Model and Resource Model parameters may be too complex to be effectively represented in OWL. In such cases, the system can leverage the developed or updated ML models to derive appropriate resource allocation and deployment constraints at runtime.

4 Results and Discussion

4.1 CorMOS Experimental Testbed and Example Workflow

To evaluate CorMOS, we developed a heterogeneous Kubernetes cluster for deploying distributed ML workflows. The cluster comprises three physical nodes: one high-performance server (master node) and two Raspberry Pi 5 devices (worker nodes), all connected through the local-area network (Fig. 5). K3s[12] and Argo Workflows were selected as the cluster management and workflow orchestration platforms because of their lightweight footprint, ease of deployment, and full Kubernetes compatibility. K3s serves as the Kubernetes distribution, installed as a single binary on the master node. Worker nodes were registered with the K3s server using the standard K3s agent registration mechanism. Argo Workflows (with the Argo CLI) manages the execution of containerised applications and workflow dependencies. Prometheus and Grafana were integrated for observability and performance visualisation, while MinIO[13] provides S3-compatible object storage for datasets and workflow artifacts, ensuring compatibility with the NEMO Meta-OS data-management model. This integrated environment enables end-to-end evaluation of the CorMOS orchestration pipeline, from intent specification and semantic enrichment to deployment, monitoring, and feedback analysis, within a realistic, resource-constrained infrastructure. It thus provides the basis for assessing the effectiveness and robustness of CorMOS in managing distributed ML workloads aligned with NEMO's architectural and operational goals.

[12] K3s, Lightweight Kubernetes, https://k3s.io/
[13] MinIO , https://www.min.io/

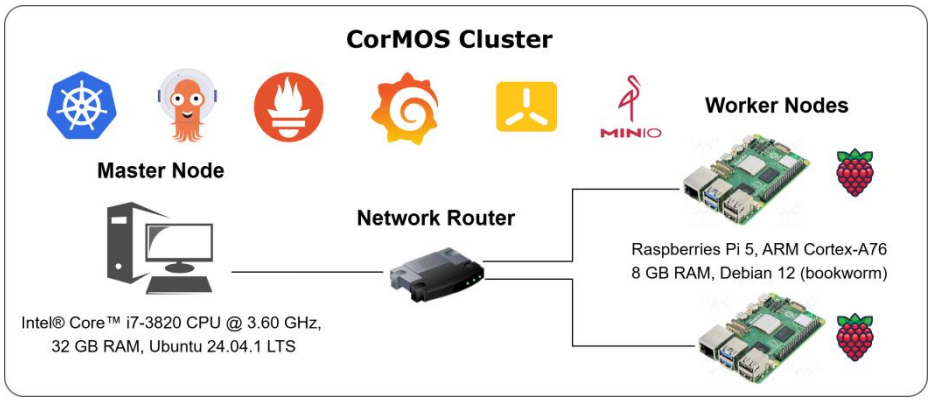


Fig. 5. CorMOS cluster – Main and Worker Nodes including relevant Frameworks.

To demonstrate the practical functionality of the CorMOS orchestration framework, a representative ML workflow was developed. The application was implemented as two distinct containerised microservices. The first microservice performs dataset preprocessing and model training, producing a serialised scikit-learn MLPClassifier model artifact (stored as a .pkl file). The second microservice consumes this trained model and executes inference on new environmental sensor values (e.g., soil nutrients, atmospheric measurements, meteorological parameters) to generate a crop recommendation output. Both microservices were implemented in Python (v3.10), using common data science libraries including pandas, numpy and scikit-learn. The output of the training microservice (model artifact) is consumed by the input to the inference microservice.

Upon receiving the Helm chart (Fig. 6), the CorMOS Orchestration Engine first parsed the descriptor and retrieved the metadata of the involved microservices. In this experiment, the user expressed a specific goal (e.g., minimise response latency) which was declared using a JSON structure that followed the NEMO Intent schema. This JSON object was consumed by CorMOS as the formal specification of the user's intent. Before orchestration, the ontology instances relevant to this category of workload (e.g., ML Training Component, ML Inference Component, Low-Latency Edge Inference Scenario) had previously been instantiated in the OWL repository, therefore the engine could perform semantic queries to determine which resource profile corresponds to this goal. In this step, the ontology acts as a semantic filter that determines which resource parameters form the valid search space. The inferred resource requirements (CPU cores, RAM, storage capacity, etc.) are treated as mandatory constraints, while placement preferences are encoded through node affinity where appropriate. Using this knowledge base, the Orchestration Engine updates the Helm deployment template by injecting the appropriate node selectors and node affinity expressions directly into the Helm chart. For example, the inference component is annotated with location=edge and given lower CPU and RAM requirements, while the training component is annotated with location=cloud (i.e., a label that corresponds to the high-performance local node) and allocated a higher CPU budget. This allocation

reflects the different computational profiles of the two components: model training is computationally intensive, whereas inference prioritises low latency. Node selector and affinity rules are then automatically embedded into the deployment YAML, so that when Argo submits the workflow the Kubernetes scheduler is guided to place each microservice on the correct node class. In this way, high-level user intents are translated into concrete resource-aware deployment decisions.

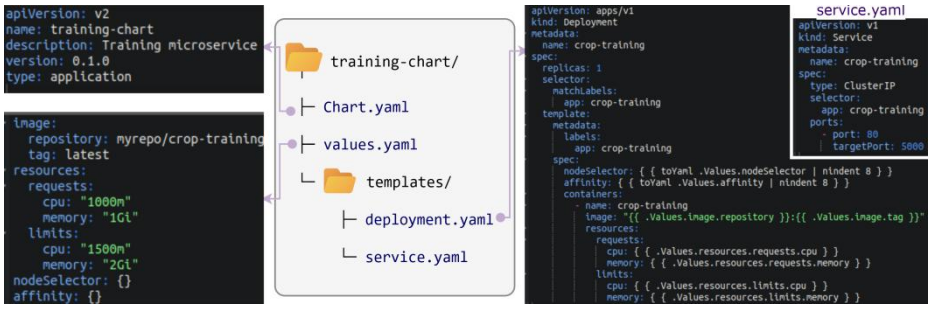


Fig. 6. Containerised application developed for testing purposes.

To further evaluate the ML-based resource optimisation capabilities of CorMOS, an experimental procedure was conducted using the two microservices deployed in the developed Kubernetes cluster. Initially, execution-time and resource-utilisation data were collected by executing each microservice under different workload sizes (i.e., numbers of dataset records) and resource configurations (i.e., CPU and memory requests and limits). These measurements were subsequently used to train Linear Regression models that predicted the expected execution time of each microservice for a given workload and resource configuration. Following the approach presented in our previous work [30], these predictions were employed to identify resource configurations that were expected to satisfy the application-level execution-time constraints specified through the user's intent.

To assess the adaptability of CorMOS under changing resource availability, an additional experimental scenario was conducted in which background workloads were deployed concurrently on the cluster nodes, introducing different levels of CPU contention. Under these conditions, resource availability and execution conditions differed from those represented in the initial training dataset, reducing the prediction accuracy of the existing ML models. Moreover, resource configurations derived from the initial predictions were not always sufficient to satisfy the specified execution-time constraints. To address this limitation, the RL module exploited resource-utilisation metrics collected through Prometheus, together with the observed execution times, to identify significant deviations between predicted and actual application performance. When such deviations exceeded a predefined threshold (i.e., at least one model achieved a prediction accuracy of less than 80% on the test dataset), the module initiated an automated data collection and model update procedure, incorporating measurements obtained under the new operating conditions. Following these updates, the RL module reassessed candidate resource configurations and derived revised re-

source constraints, which were subsequently incorporated into the Kubernetes deployment descriptors through the CorMOS Orchestration Engine.

The experiments described in the last two paragraphs demonstrated the use of ML models for intent-based resource orchestration and the ability of monitoring-driven model updates to adapt resource predictions in response to deviations between expected and observed execution times under resource contention. Nevertheless, further experiments involving diverse workloads and heterogeneous cluster configurations are necessary to assess the generalizability and robustness of the proposed approach, as well as to compare its effectiveness with that of conventional RL methods.

In addition, the implemented software components were evaluated against several criteria, including overall execution time, the identification of missing resource constraints through ontology-based reasoning, and the contribution of the ML/RL modules to the refinement of user-defined intents, the improvement of prediction accuracy, and the overall effectiveness of the orchestration process. The evaluation criteria and results are summarised in Tables 1 and 2, respectively. Table 1 defines the Key Performance Indicators (KPIs) considered, while Table 2 presents the corresponding results across different experimental scenarios conducted on the developed cluster and within the NEMO platform, as described in the following section.

4.2 NEMO Testbed, Use Cases, and Experimental Validation

The NEMO project defines a meta-operating system (meta-OS) for the IoT–Edge–Cloud continuum, enabling heterogeneous clusters to be managed as a unified platform for deploying AI-enabled workloads. In NEMO, workloads are submitted as intents which are validated, translated and enforced by a Meta-Orchestrator (MO), while cluster resources are abstracted as managed Kubernetes clusters grouped into cluster sets forming a global cluster shell. This functionality is exposed through the Intent-Based API, which supports workload registration, validation and automated service exposure. Within this framework, multiple NEMO clusters may be registered through the Life-Cycle Management (LCM) UI or the Intent-Based API, each providing compute, storage, energy and cost characteristics to the global resource pool. The high-level architecture can be observed schematically in Fig. 7. More details can be found in the NEMO deliverables[14] D1.2, D1.3, D4.1 and D4.2.

[14] NEMO deliverables, https://meta-os.eu/index.php/deliverable/

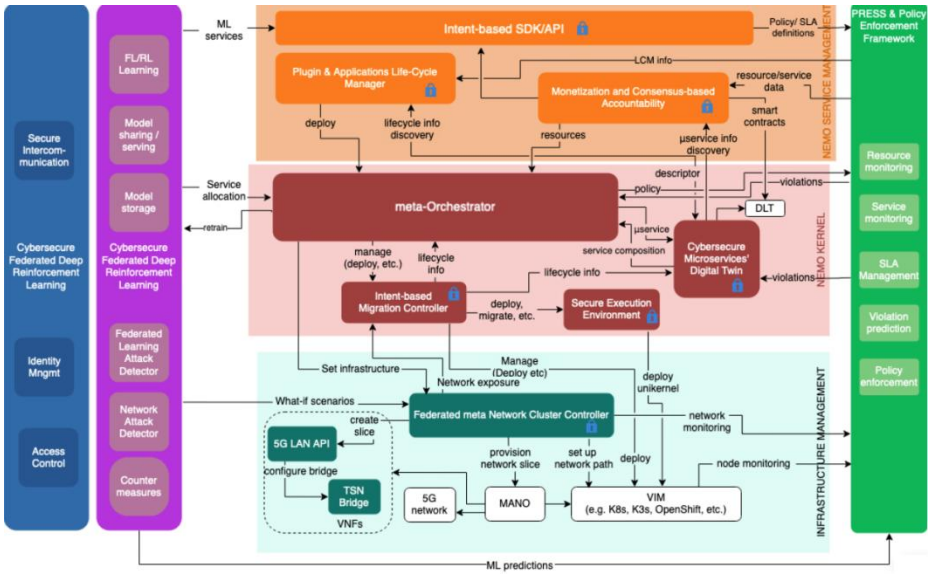


Fig. 7. High level architecture of NEMO (Source: NEMO Deliverable D4.1).

NEMO defines a number of Use Cases (UCs) that implement real deployments across different domains (e.g., Smart Agriculture, Smart Mobility, Smart Manufacturing, Energy). In the context of this work, we focus on UCs related to Smart Farming. Both UCs share a similar high-level objective, to support sustainable agricultural decision-making through AI/ML models that analyse agronomic conditions. The main difference between the two UCs lies in the monitoring modality: one UC generates observations primarily through in-field terrestrial sensors, while the other acquires environmental state estimates via aerial acquisition (drones). However, from an orchestration perspective, the computational structure is similar across both UCs, as they rely on multi-step ML pipelines that include data acquisition, preprocessing, model training, and inference. Therefore, both UCs can be represented using the same classes of containerised microservices and dependency graphs. This similarity makes them appropriate reference patterns for validating CorMOS, because the orchestrator must handle the same workload structure but under different infrastructure, energy and data acquisition characteristics.

CorMOS was designed to integrate with NEMO as the component responsible for the deployment of workloads within the cluster where it runs. Due to architectural decisions made during the development of the NEMO platform, a slightly different integration approach was adopted. In the originally envisioned architecture, NEMO defines the high-level intent and selects which cluster will execute it, while CorMOS takes this validated intent and translates it into concrete Kubernetes deployment constraints and descriptors. In practice, the Meta-Orchestrator in NEMO produces the final intent representation, and CorMOS then enriches the corresponding workload description (Helm chart) with resource constraints, node selectors and affinities before triggering deployment on the target cluster. For this purpose, CorMOS includes its

own ontology-based semantic model, which supports the interpretation of intents in a structured way and enables automatic completion of missing resource parameters. In this sense, CorMOS does not replace NEMO orchestration functions; instead it extends NEMO by providing the specialised execution logic needed for precise node-level placement and optimisation inside the selected cluster.

Unlike the scenario, which focused on executing the classification workflow on our own heterogeneous k3s cluster, the experiments performed inside the NEMO testbed targeted two different aspects of CorMOS validation. The first experiment evaluated end-to-end interoperability within the NEMO orchestration pipeline. In this setup, Helm charts together with their configuration values were uploaded to MinIO/S3, retrieved via the RabbitMQ[15] message bus, transformed by CorMOS into well-formed intents, and then submitted to the NEMO Meta-Orchestrator for execution. The purpose of this scenario was not to test the accuracy of the workload itself, but to verify that CorMOS could act as a semantic front-end within NEMO, correctly consuming remote artefacts, enriching incomplete descriptors, and forwarding fully structured intent objects for orchestration. The second experiment focused on robustness under heterogeneous workload inputs. A set of diverse Helm charts were submitted with intentionally omitted or partially specified resource parameters. CorMOS was responsible for inferring these missing fields, normalizing the descriptors, and generating complete deployment intents. This scenario stressed semantic inference and enrichment rather than execution logic, and assessed whether CorMOS could handle unpredictable descriptor variability without manual corrections. Together, the two experiments evaluate complementary aspects of CorMOS: the first validates interoperability within the complete NEMO orchestration pipeline, whereas the second evaluates the robustness of the semantic enrichment mechanism when confronted with incomplete workload specifications. The results are presented in Table 2.

4.3 Key Performance Indicators (KPIs) and Experiment Results

To ensure consistent evaluation across all experiments, a common set of Key Performance Indicators was defined. These KPIs capture both the functional efficiency of the CorMOS orchestration process and its semantic enrichment accuracy. The KPIs used are shown in Table 1.

Table 1. Definition of Key Performance Indicators (KPIs).

KPI Name	Short Name	Brief Description
Processing time per Helm chart	Processing time	Measures the time required for CorMOS to parse, semantically enrich, and transforms workload descriptors into executable deployment descriptors.
Semantic inference rate	Inference rate	Quantifies the proportion of missing resource parameters successfully inferred through semantic reasoning.
Intent completeness	Completeness	Indicates the percentage of generated intents that

[15] RabbitMQ, https://www.rabbitmq.com/

		passed validation without manual correction.
Resource estimation accuracy	Accuracy	Compares inferred resource parameters against reference (ground-truth) configurations.
Throughput	Throughput	Measures sustained workload processing capability.
Orchestration overhead / latency	Latency	Additional time introduced by message exchanges, artifact retrieval, or scheduling.

Table 2 summarises the KPI results obtained across all three experimental scenarios. All reported values correspond to the average of 3 independent executions.

Table 2. Summary of KPIs across all experimental scenarios.

KPI Short Name	Experiment 1: ML Workflow in CorMOS cluster	Experiment 2: Integrated with NEMO Meta-OS	Experiment 3: ML Workflow in NEMO cluster
Processing time	≈ 1.2 s – 4.5 s	≈ 1.5 s – 6.0 s	≈ 3 s
Inference rate	≈ 40 %	≈ 40 %	≈ 50 %
Accuracy	≈ 78 %	≈ 79 %	≈ 80 %
Completeness	≈ 92 %	≈ 91 %	≈ 100 %
Throughput	≈ 20 charts/min	≈ 17 – 18 charts/min	≈ 17 charts/min
Latency	< 1 s added latency	≈ 300 ms extra (due to RabbitMQ and S3)	≈ 1.3 s total overhead
Kubernetes scheduling delay	≈ 0.5 s	≈ 0.5 s	≈ 0.5 s

These results demonstrate the effectiveness of CorMOS in the orchestration pipeline by delivering fast and semantically accurate workload preparation. Across all scenarios, intent translation remained within only a few seconds, demonstrating that the enrichment logic introduces limited additional latency. On average, CorMOS automatically inferred approximately 40% of missing parameters in generic Helm charts, and up to 50% in more realistic multi-container ML workflows, effectively turning incomplete descriptors into deployable ones without manual adjustment. The inferred resource parameters were compared against reference (ground-truth) configurations, yielding an estimation accuracy of approximately 80%, while more than 90% of the generated intents passed schema validation without manual correction. Even when exposed to external middleware interactions, throughput remained near 20 charts/min and the additional overhead introduced by RabbitMQ, S3 and Kubernetes scheduling remained below 1.5s end-to-end.

Although direct quantitative comparison is difficult because existing intent-based orchestration systems report different evaluation metrics, the observed processing overhead (1–6 s) and inference accuracy (up to 80%) indicate that CorMOS is suitable for practical deployment within Kubernetes-based orchestration pipelines. On the other hand, some limitations remain. In the current implementation, most resource inference still depends on static descriptors and predefined workload classes, which

means that dynamic or rapidly changing workloads (e.g., bursty inference services) are not yet fully addressed. Also, the ontology currently covers only a limited set of application categories, so new domains still require some manual enrichment before automatic inference becomes reliable. Nevertheless, the experiments showed that the CorMOS components (ontology, mapping rules, and the chart enrichment logic) are not structurally coupled to our specific cluster or to NEMO. They can be reused in other orchestration systems that rely on Helm charts or similar declarative formats, as long as those systems support applying placement hints (selectors/affinities) before deployment. In this sense, CorMOS can act as a portable semantic pre-processor that sits before any Kubernetes-based scheduler.

5 Conclusion and Future Work

In this paper, we presented CorMOS, a semantic deployment descriptor enrichment mechanism that translates high-level user intents and partially specified AI workflows into Kubernetes-compatible resource requirements and placement constraints. CorMOS combines OWL-based semantic reasoning, predefined mapping rules, and ML techniques to automate resource inference and Helm chart enrichment. The system was evaluated on a heterogeneous k3s cluster and integrated into the NEMO orchestration environment. Experimental results demonstrated resource inference rates of up to 50%, estimation accuracy of approximately 80%, and throughput of 17–20 Helm charts per minute, highlighting the feasibility of lightweight, intent-driven deployment automation across heterogeneous cloud–edge infrastructures. Despite these encouraging results, the current approach relies partly on predefined workload classifications and expert-defined rules, limiting its applicability to previously unseen workloads. Furthermore, the evaluation covers a limited range of workflows and infrastructure configurations. Future work will focus on extending the ontology, conducting systematic comparisons with alternative inference approaches, and evaluating adaptive resource prediction and deployment scalability across larger, geographically distributed cloud–edge environments.

Acknowledgments. The work presented in this paper was carried out as part of the Cross-Domain Orchestration of Meta-OS Edge Resources (CorMOS) project, funded through the Next Generation Meta Operating System (NEMO) Open Call 1 (OC1). The authors gratefully acknowledge the support and fruitful collaboration of all members of the NEMO consortium throughout the project.

References

1. Zhou, Z., Chen, X., Li, E., et al.: Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing. Proceedings of the IEEE, 107(8), 1738-1762 (2019)
2. Zaharia, M., Chen, A., Davidson, A., et al.: Accelerating the Machine Learning Lifecycle with MLflow. IEEE Data Engineering Bulletin, 41(4), 39-45 (2018)

3. Wu, C., Horiuchi, S., Tayama, K.: A Resource Design Framework to Realize Intent-Based Cloud Management. In: Proc. IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 37-44, IEEE (2019)
4. Kang, J.M., Lee, J., Nagendra, V., Banerjee, S.: LMS: Label Management Service for Intent-Driven Cloud Management. In: Proc. IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 177-185, IEEE (2017)
5. Pahl, C.: Containerization and the PaaS Cloud. IEEE Cloud Computing, 2(3), 24-31 (2015)
6. Burns, B., Grant, B., Oppenheimer, D., et al.: Borg, Omega, and Kubernetes. Communications of the ACM, 59(5), 50-57 (2016)
7. Block, A., Dewey, A.: Managing Kubernetes Resources Using Helm. Packt Publishing (2022)
8. Merkel, D.: Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal, 239(2), 2 (2014)
9. Alavilo, A., Puutonen, E.: Containerization and Deployment of a Web Platform via Red Hat OpenShift (2025)
10. George, J., Saha, A.: End-to-End Machine Learning Using Kubeflow. In: Proc. 9th ACM IKDD CODS and 27th COMAD, pp. 336-338 (2022)
11. Pandey, A., Sonawane, M., Mamtani, S.: Deployment of ML Models Using Kubeflow on Different Cloud Providers. arXiv preprint arXiv:2206.13655 (2022)
12. Gkoufas, Y., Yuan, D.Y., Pinto, C., et al.: Datashim and Its Applications in Bioinformatics. In: Proc. International Conference on High Performance Computing, pp. 416-427, Springer (2021)
13. Holmberg, D., Golubovic, D., Kirschenmann, H.: Jet Energy Calibration with Deep Learning as a Kubeflow Pipeline. Computing and Software for Big Science, 7(1), 9 (2023)
14. Jayanetti, A., Halgamuge, S., Buyya, R.: A Deep Reinforcement Learning Approach for Cost Optimized Workflow Scheduling in Cloud Computing Environments. In: Proc. Asia Pacific Conference on Computing Technologies, Communications and Networking, pp. 74-82 (2024)
15. Wang, Y., Janse, N., Bianchi, R., et al: Towards a Service-Based Adaptable Data Layer for Cloud Workflows. In: Proc. IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 904-911 IEEE (2023)
16. Crusoe, M.R., Abeln, S., Iosup, A., et al.: Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language. Communications of the ACM, 65(6), 54-63 (2022)
17. Psomakelis, E., Nikolakopoulos, A., Marinakis, A., et al.: A Scalable and Semantic Data as a Service Marketplace for Enhancing Cloud-Based Applications. Future Internet, 12(5), 77 (2020)
18. Psomakelis, E., Makris, A., Tserpes, K., et al.: A Lightweight Storage Framework for Edge Computing Infrastructures / EdgePersist. Software Impacts, 17, 100549 (2023)
19. Buchaca, D., Berral, J.L., Wang, C., Youssef, A.: Proactive Container Auto-Scaling for Cloud Native Machine Learning Services. In: Proc. IEEE 13th international conference on cloud computing (CLOUD), pp. 475-479 IEEE (2020)
20. Rubak, A., Taheri, J.: Machine Learning for Predictive Resource Scaling of Microservices on Kubernetes Platforms. In: Proc. IEEE/ACM 16th international conference on utility and cloud computing (UCC), pp. 1-8 (2023)
21. Xu, R., Wunsch, D.: Survey of Clustering Algorithms. IEEE Transactions on Neural Networks, 16(3), 645-678 (2005)

22. Sharma, S., Saini, H.: A novel four-tier architecture for delay aware scheduling and load balancing in fog environment. Sustainable Computing: Informatics and Systems, 24, 100355 (2019)
23. Chandola, V., Banerjee, A., Kumar, V.: Anomaly Detection: A Survey"; ACM Computing Surveys, 41(3), 1-58 (2009)
24. Mitropoulou, K., Kokkinos, P., Soumplis, P., E Varvarigos: Anomaly Detection in Cloud Computing using Knowledge Graph Embedding and Machine Learning Mechanisms, Journal of Grid Computing, 22(1), 6 (2024)
25. Martin, I.R., Arcas, G.I., Cioara, T.: Reinforcement-Learning-Based Edge Offloading Orchestration in Computing Continuum. Computers, 13(11) 295 (2024)
26. Hady, M.A., Hu, S., Pratama, M., et al.: Multi-Agent Reinforcement Learning for Resources Allocation Optimization: A Survey. Artificial Intelligence Review, 58(11), 354 (2025)
27. Zhao, W.X., Zhou, K., Li, J., et al.: A Survey of Large Language Models. Frontiers of Computer Science, 20(11), 2012627 (2026)
28. Angi, A., Nedoshivina, L., Sacco, A.: A perspective on llm data generation with few-shot examples: from intent to kubernetes manifest. In: Proc. 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), pp. 345–354 (2025)
29. Palavalli, M.A., Santolucito, M.: Using a feedback loop for LLM-based infrastructure as code generation. arXiv preprint arXiv:2411.19043 (2024)
30. Chondrogiannis, E., Karanastasis, E., Andronikou, V., et al.: Intent-Based AI-Enhanced Service Orchestration for Application Deployment and Execution in the Cloud Continuum. In: Proc. European Conference on Service-Oriented and Cloud Computing, pp. 251-262, Springer (2023)
31. Sirin, E., Parsia, B., Grau, B.C., et al.: Pellet: A practical owl-dl reasoner. Journal of Web Semantics, 5(2), 51-53 (2007)
32. Hutter, F., Kotthoff, L., Vanschoren, J.: Automated machine learning: methods, systems, challenges (p. 219). Springer Nature (2019)