Improving the Cloud Provider Ranking in the INDIGO PaaS Orchestration System Using AI Techniques

Luca Giommi[1(✉)], Giovanni Savarese[2], Gioacchino Vino[2], Domingo Ranieri[3], Alessandro Costantini[1], and Giacinto Donvito[2]

[1] INFN-CNAF, Viale Berti Pichat 6/2, 40127 Bologna, Italy
luca.giommi@cnaf.infn.it
[2] INFNSezione di Bari, Via Giovanni Amendola 173, 70126 Bari, Italy
{giovanni.savarese,gioacchino.vino,giacinto.donvito}@ba.infn.it
[3] ICSC, DAMA - Tecnopolo Data Manifattura Emilia-Romagna, Via Stalingrado 84/3, 40128 Bologna, Italy
domingo.ranieri@supercomputing-icsc.it

Abstract. The INFN Cloud platform's federation middleware is based on the INDIGO PaaS Orchestration system, which allows handling high-level deployment requests from users while orchestrating the deployment process across the IaaS platforms made available by the federated Cloud providers. Currently, the INDIGO PaaS Orchestrator determines which provider use for creating a deployment based on an ordered list of providers selected according to the user's group. This list is provided by the Cloud Provider Ranker (CPR) service, which applies a ranking algorithm using a limited set of metrics related to the user's request and the Service Level Agreements defined for the providers. The INDIGO PaaS Orchestrator submits the deployment to the first provider on the list and, in case of failure, moves to the next provider until the list is exhausted.

This contribution presents the ongoing activity aimed at improving the provider ranking system by leveraging a broader set of metrics and Artificial Intelligence (AI) techniques. In particular, we introduced Apache Kafka as a message broker to collect heterogeneous information, such as resource usage in federated providers, data on past deployments, and the resources requested by users for new deployments. We then developed the AI-ranker service, based on MLFlow, to replace the CPR, for training and inference of Machine Learning models using the messages read from specific Kafka topics. The ranked list of providers is generated by the AI-ranker by combining the prediction of deployment success/failure with the estimated deployment creation time.

Keywords: Artificial Intelligence · Cloud Computing · PaaS Orchestration System · Orchestrator · Cloud Provider Ranking

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026
O. Gervasi et al. (Eds.): ICCSA 2025 Workshops, LNCS 15886, pp. 35–50, 2026.
https://doi.org/10.1007/978-3-031-97576-9_3

1 Introduction

The National Institute for Nuclear Physics (Istituto Nazionale di Fisica Nucleare, INFN) is the leading research institution in Italy specializing in nuclear, particle, theoretical, and astroparticle physics. INFN not only conducts and coordinates scientific research but also drives technological advancements in these fields. Additionally, it oversees the country's most extensive public computing infrastructure for scientific research, supporting more than 40 international collaborations.

In March 2021, INFN started the "INFN Cloud" initiative, designed to establish a federated Cloud infrastructure and provide a customizable portfolio of services tailored to the needs of scientific communities. This offering spans traditional Infrastructure as a Service (IaaS) solutions, alongside more advanced Platform as a Service (PaaS) and Software as a Service (SaaS) options, ensuring flexibility for various research requirements. At the core of INFN Cloud lies a middleware system based on the INDIGO PaaS Orchestration framework, which integrates interconnected open-source microservices. Among these, the INDIGO PaaS Orchestrator is responsible for handling deployment requests and managing resource provisioning across federated Cloud providers. By default, it selects a provider based on a prioritized list derived from user group memberships. This ranking is dynamically generated by the Cloud Provider Ranker (CPR), which applies an algorithm that considers a limited set of deployment metrics and the Service Level Agreements (SLAs) established for each provider. The Orchestrator initially attempts the deployment with the highest-ranked provider, and if unsuccessful, moves sequentially through the list until all options have been exhausted.

This paper presents an ongoing effort to enhance the Cloud provider ranking system by incorporating a more extensive set of metrics and leveraging Artificial Intelligence (AI) techniques. The objective is to improve the efficiency and accuracy of deployment decisions within the federated Cloud infrastructure by utilizing historical data. A key aspect of this approach is the integration of Apache Kafka [4] as a message broker, allowing for the continuous collection and processing of data from multiple sources. Among the collected information are real-time resource utilization metrics from federated Cloud providers and user-defined resource requirements for new deployment requests. To replace the existing CPR, we developed an advanced AI-driven ranking service called AI-Ranker [1]. Built on MLflow [17], AI-Ranker is designed to perform both training and inference of Machine Learning (ML) models by consuming structured messages from dedicated Kafka topics. By adopting this AI-based ranking approach, the system can dynamically adjust to evolving infrastructure conditions, improving decision-making and optimizing resource utilization across the federated Cloud.

The structure of the paper is as follows. Section 2 provides an overview of the INFN Cloud ecosystem, including its federated infrastructure and available services. Section 3 describes the INDIGO PaaS Orchestration system, outlining its components and workflow. Section 4 presents an initial attempt to leverage AI techniques to enhance the Cloud provider ranking mechanism. This process

involved defining data sources, identifying the information to be collected, creating the dataset with data preprocessing, training the models, and evaluating their performance. Section 5 details the implementation of the Kafka-based service for data collection and aggregation, as well as the AI-Ranker service for training and inference of ML models. Finally, Sect. 6 summarizes the work and outlines future directions.

2 The INFN Cloud Ecosystem

To maximize the efficiency of available resources and expertise, INFN developed a national Cloud infrastructure for research, designed with three key principles in mind.

- **Federation of distributed infrastructures**: integrating existing infrastructures while transparently extending them to private Cloud providers when needed.
- **User-centric approach**: offering a dynamic set of tailored services to meet the specific needs of the scientific communities.
- **Building on past initiatives**: leveraging the results of various national and European Cloud projects in which INFN actively participated, such as the INDIGO-DataCloud project [25].

The INFN Cloud infrastructure is built on a core backbone, connecting at high speed the CNAF and Bari data centers, where the core Cloud services are hosted. This backbone connects to a network of loosely coupled federated sites, which currently include Cloud-CNAF, ReCaS-Bari, Cloud-Veneto, Cloud-INFN-Catania, and Cloud-Ibisco-Napoli. Moreover, INFN Cloud offers a broad range of services categorized into IaaS, PaaS and SaaS, allowing users to interact with the infrastructure at different levels.

- **SaaS (e.g., Notebook as a Service)**: ready-to-use solutions requiring no user customization.
- **PaaS (e.g., Virtual Machine (VM) or Kubernetes** [15] **cluster provisioning)**: configurable on-demand services, where users can, for example, select the OS and specify the compute resource flavor.
- **IaaS (e.g., starting and stopping VMs)**: direct infrastructure-level operations that users can execute as needed.

All services are accessible through a user-friendly interface, the INFN Cloud Dashboard [8], see Fig. 1. Advanced users can also interact with the platform programmatically through REST APIs, a command-line interface (CLI), and Python bindings.

The PaaS services available in the INFN Cloud portfolio of services are managed using an Infrastructure as Code (IaC) approach, ensuring automation, flexibility, and portability across different environments through a combination of:

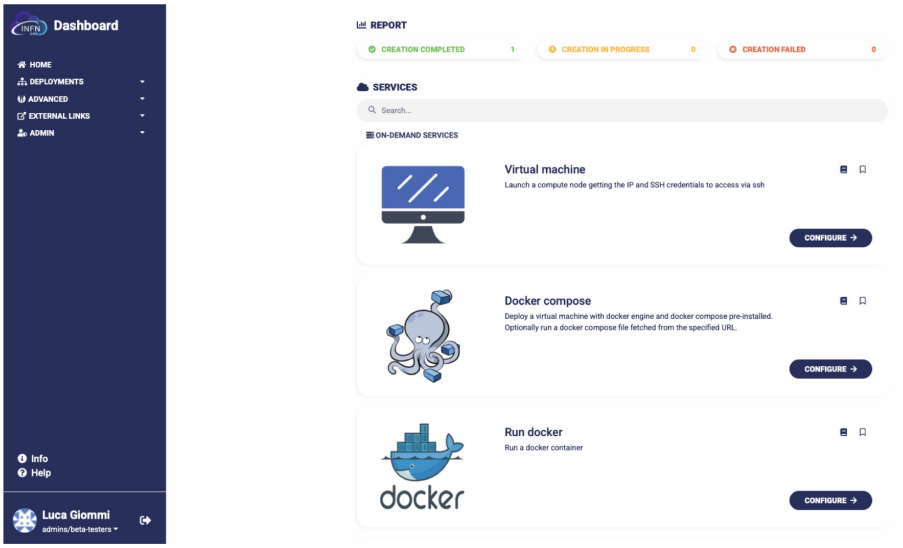

Fig. 1. The INFN Cloud dashboard. It allows users to access centralized services, instantiate PaaS services independently, and access IaaS solutions. The PaaS services available in the dashboard depend on the user group.

- **Topology and Orchestration Specification for Cloud Applications (TOSCA) templates** [29], used to model an application stack by defining its components, relationships, and deployment lifecycle;
- **Ansible roles** [2], which facilitate the automated configuration and management of virtual environments;
- **Docker containers** [5], employed to encapsulate high-level application software along with its dependencies and runtime environment;
- **Helm charts** [6], used to define, package, and manage the deployment of applications in Kubernetes clusters.

In INFN Cloud, each PaaS service is associated with a dedicated TOSCA template [30] that leverages extended INDIGO-DataCloud custom types [31]. A Lego-like approach has been adopted, leveraging modular and reusable components while utilizing the TOSCA service composition pattern.

3 The PaaS Orchestration System

The federation middleware of the INFN Cloud platform is built upon the INDIGO PaaS Orchestration system, which integrates a set of interconnected open-source microservices [3], see Fig. 2. These microservices establish an abstraction layer that facilitates seamless access to compute and storage resources provided by diverse and heterogeneous Cloud infrastructures. Currently in INFN Cloud, all the federated providers make available an instance of

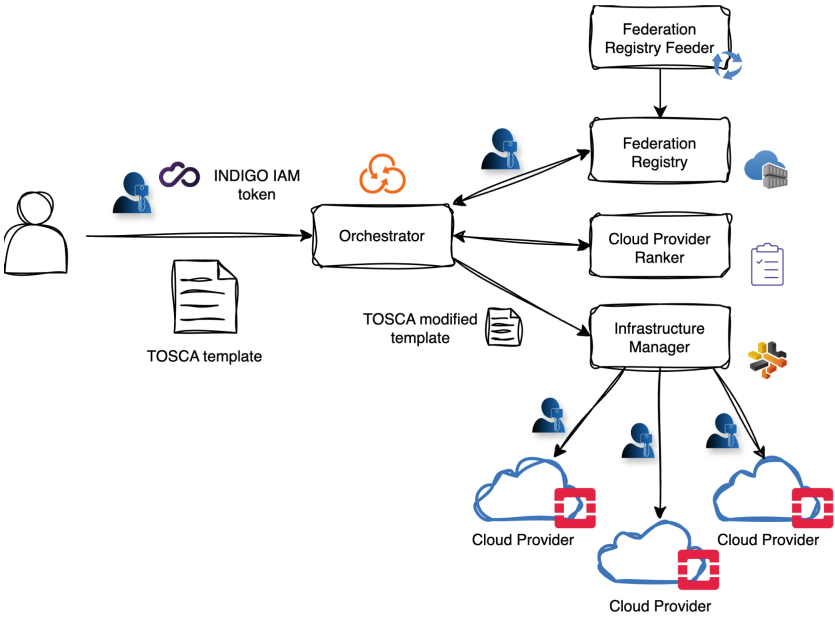

Fig. 2. High-level architecture and workflow of the PaaS Orchestration system. More details are provided in the text.

OpenStack [20] while the backbone provides two different OpenStack implementations, one at CNAF and one at INFN-Bari, which act as a single one with two regions. The core component of such a system is the INDIGO PaaS Orchestrator [21] which takes the information about the user's request in the form of a TOSCA template and interacts with the Federation Registry service to get information about the available providers and their capabilities (e.g. about flavors, images, and networks) [26]. The Federation Registry is populated by the Federation Registry Feeder that periodically contacts the OpenStack sites to collect information about the available resources. Then the CPR, using a limited set of deployment metrics and the SLAs established for each provider, gives to the Orchestrator an ordered list of providers where send the deployment request. The Orchestrator modifies and extends the TOSCA template submitted by the user and sends it to the Infrastructure Manager (IM) service [10] (the low-level orchestrator, at the IaaS level) that contacts the OpenStack site chosen by the Orchestrator for the provisioning of the resources. The Orchestrator tries the submission to the first provider in the list and, in case of failure, moves to the next provider until the list is exhausted.

Another essential component of the platform is the Authentication and Authorization Infrastructure, implemented through the INDIGO Identity and Access Management (INDIGO IAM) [7] service. This system governs authentication and authorization across all layers of INFN Cloud, including IaaS, PaaS, and SaaS services. To access the Orchestrator's APIs, a user must present a

JSON Web Token (JWT) [11] issued by the INFN Cloud INDIGO IAM service. This token contains a *groups* claim that specifies the INDIGO IAM groups the user belongs to. Each group corresponds to a scientific community, collaboration, or experiment and is mapped to dedicated OpenStack projects within the federated Cloud infrastructure. After a token exchange operation, the Orchestrator obtains a token from the INDIGO IAM service, which is then used in the request to the IM to access the IaaS-level services and create the required resources.

4 A First Study on Enhancing the Cloud Provider Ranking System with AI

To enhance the Cloud provider ranking mechanism of the PaaS Orchestration system, we adopted a Data Science approach and worked on integrating AI techniques. As a first step, we conducted an initial attempt to assess the feasibility of this approach and to explore the potential of the available data [16]. A key aspect of this work was to identify relevant information for the use case, determine where to find it, and establish methods for its collection. Below, we report the available data sources and the types of stored information.

- **Monitoring database**: contains information about resource usage over time, categorized by user group and by provider.
- **Rally** [24] **service**: provides data on simple tests periodically run on various providers, e.g. VMs creation.
- **Orchestrator Dashboard database**: stores information about all deployments.
- **Orchestrator logs**: maintain a complete history of all deployments.
- **Accounting**: offers aggregated data on resource usage over time, grouped by user and provider.
- **Orchestrator database**: tracks deployments that have not yet been deleted.

The last two sources were discarded, as their information was either already available in the other sources or considered superfluous. Then we definied the list of information to extract from the sources, that are: deployment ID, deployment creation time, provider name, user group, selected TOSCA template, resources requested by the user for the deployment (in terms of CPU, RAM, number of volumes, disk size of the volumes, number of VMs, floating IPs, request of GPU), quotas and resource usage by group and provider (in terms of CPU, RAM, number of volumes, disk size of the volumes, number of VMs, floating IPs), failure rate of simple tests run periodically on various providers, status of a deployment, creation/failure time of a deployment, average creation time of a deployment in the last month by deployment type and provider, deployment failure rate in the last month by deployment type and provider. Additionally, we included some static information obtained by contacting the administrators of the federated sites, i.e. bandwidth and overbooking for CPU and RAM.

The deployment type is determined by the name of the selected TOSCA template used to submit the deployment request. We grouped the templates

into five categories based on different service types: single VM, which can be deployed with or without an external volume and on a private network or not; single VM with services, where the services can include Sync&Share, INDIGO-IAM, or Elasticsearch and Kibana; Docker, used for the Docker Run and Docker Compose services, with or without an external volume; K8s, which includes both simple Kubernetes clusters and configurations with Kafka, HTCondor, and Spark, deployed on a private network or not; Jupyter, which includes the JupyterHub/Lab service on a single VM, on a private network or not, with or without Matlab installed, and the services used for the Jupyter-based platforms of the Cygno and ML_INFN experiments. See [9] for more details about the services offered by INFN Cloud.

A dataset was created with the aforementioned information collected from the corresponding data sources. In particular, a dump of the Orchestrator Dashboard database and the Monitoring database was made, Rally data was collected using the OpenSearch [19] tool, and Orchestrator logs were gathered by the SysLog server. We considered a time window of 6 months, from the 4th of August, 2023, to the 25th of January, 2024, with a total of 643 deployments. When constructing the dataset, an entry is added for each provider that was attempted for a deployment. For instance, if a deployment is submitted to four different providers, the dataset will include four entries. See Table 1 for the distribution of deployments that ended with success or failure by deployment type during this period.

Table 1. Distribution of deployments by outcome (success or failure) and deployment type within a specific time window. More details are provided in the text.

Deployment type	# of successes	# of failures
Single VM	137	67
Single VM with services	19	23
K8s	124	159
Docker	38	41
Jupyter	25	10
Total	**343**	**300**

The obtained dataset was cleaned by removing outliers, mapping strings to integers (e.g., deployments with the *CREATE_COMPLETE* status were mapped to 0, those with the *CREATE_FAILED* status to 1, while other deployment statuses were excluded), and eliminating redundant features based on the correlation matrix, using a 90% threshold.

Given the limited number of entries in the dataset, we decided to generate new features that aggregate information from multiple existing ones, replacing them. Specifically, we introduced features that measure the difference between the available quota of a resource and its current usage, further adjusted by subtracting the amount requested by the user. For example, we defined ram_diff as:

$ram_diff = (quota_ram - ram_used) - requested_ram$. Additionally, we introduced a new feature to represent deployment complexity, replacing the original deployment type feature. To define this, we analyzed the distribution of deployments based on their creation/failure times across different deployment types, as shown in Fig. 3. This analysis revealed two distinct categories with different behaviors: both peak at low time values but differ in tail length, one with a short tail and the other with a long tail. The first category (Single VM and Docker deployments) was assigned a complexity value of 0, whereas the second category (Single VM with services, K8s, and Jupyter deployments) was assigned a complexity value of 1.

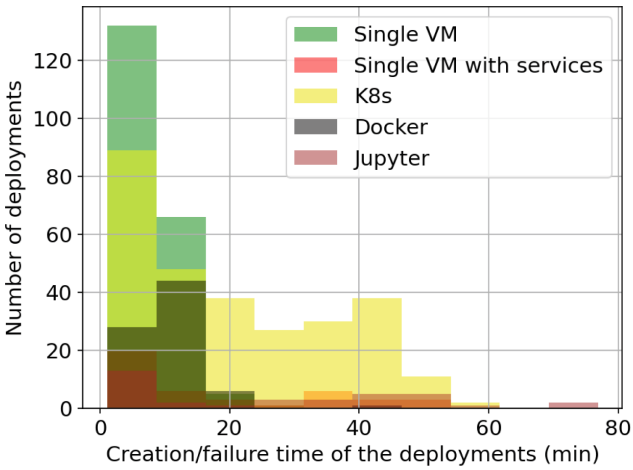

Fig. 3. Number of deployments by creation/failure time. More details are provided in the text.

Finally, after applying the aforementioned preprocessing steps, we defined eleven metrics as input features for the ML models. These features were normalized using the Min-Max normalization technique, which scales the data to the fixed range of $[-1, 1]$.

4.1 Model Design, Training, and Performance Evaluation

Two ML models were defined: one for classifying the deployment status (success/failure) and another for predicting the deployment creation/failure time using regression. For training, a sliding window approach with fixed-size data segments was adopted, where predictions were made on the subsequent time window. This process was iterated by shifting the windows along the time axis. A study was conducted to determine the optimal window size, resulting in the selection of 75 entries for training and 10 for performance evaluation. This configuration yielded the highest average accuracy of 0.692 for the Random Forest

classification model. For the classification task, multiple ML models were evaluated using their default hyperparameters, specifically: Decision Tree, Random Forest, XGBoost, AdaBoost, and Neural Network. The Area Under the Curve (AUC) was then computed by aggregating predictions across all time windows, and the Receiver Operating Characteristic (ROC) curve was plotted (see Fig. 4). The highest AUC value (0.80) was achieved with the Random Forest model. For the regression task, different models were compared, highlighting the need for further improvements.

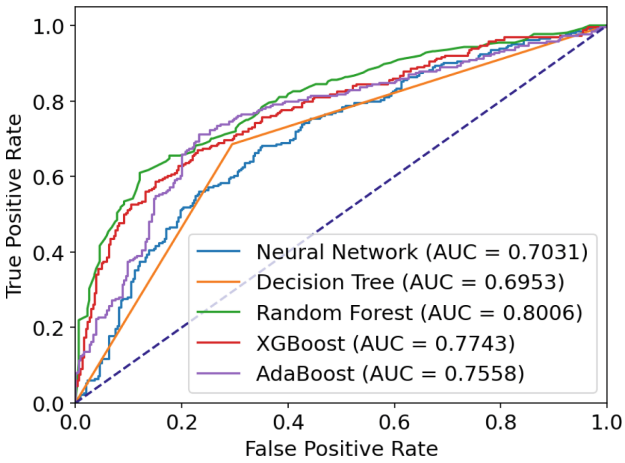

Fig. 4. ROC curves for the tested ML models.

5 Implementation of an AI-Based Service to Improve the Cloud Provider Ranking

The work presented in Sect. 4 served as a preparatory study to assess the feasibility of the proposed approach and to identify the available information and data sources. To enhance the Cloud provider ranking system and, consequently, improve the overall efficiency of the deployment process using AI techniques, we need to develop a service that can be integrated into the PaaS Orchestration system. The implementation consists of two main phases. The first phase focuses on introducing a service capable of collecting and aggregating heterogeneous information relevant to the use case, for which we selected Kafka as the core technology. The second phase involves the development of a service called AI-Ranker, dedicated to the training and inference of ML models, with MLflow as the core framework. In the following we provide details on both phases.

5.1 Data Collection and Aggregation Using Kafka

Apache Kafka is a distributed streaming platform designed for building real-time data pipelines and streaming applications. It provides a highly scalable, fault-tolerant, and durable mechanism for publishing and subscribing to streams of records. In Kafka, data is organized into queues called topics. Topics can be further divided into partitions, which are replicated across the cluster to ensure fault tolerance. A Kafka cluster can consist of a single node or multiple nodes (typically an odd number) to enhance fault tolerance. Actors that write data are called producers, while those that read data are called consumers. Kafka also offers features such as high-throughput and low-latency data processing, making it suitable for a wide range of applications, including log aggregation, stream processing, and event-driven microservices. Other technologies, such as RabbitMQ [23], can be used as alternatives to Kafka, as they provide similar key features, including fault tolerance, persistence, and ease of use. An internal evaluation determined that Kafka and RabbitMQ offer equivalent functionality. However, Kafka was preferred due to the existing expertise within the team with this technology.

In Fig. 5, we present the architecture of the Kafka-based service designed to collect and aggregate relevant information, make it available to the AI-Ranker service, and provide the Orchestrator with a sorted list of providers. The architecture consists of four main components: the Kafka cluster, the Kafka producers, the Kafka consumer, and the Kafka processors.

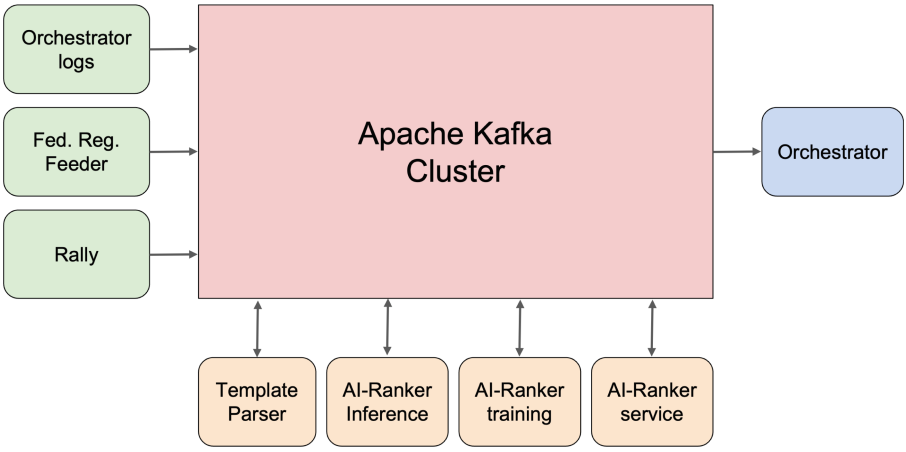

Fig. 5. Architecture of the Kafka-based service. At the center is the Kafka cluster, on the left are the producers, on the right is the Orchestrator as consumer, and below are the processors.

At the center is the Kafka cluster, which manages internal topics where messages are read and written. On the left are the producers, Kafka components

responsible for collecting information and data from outside the cluster and writing it into specific topics. On the right is the Orchestrator as consumer. Below the Kafka cluster are the processors, which are responsible for transforming messages read from some topics and writing them back into other topics. Unlike producers and consumers, processors interact exclusively with Kafka. JSON was chosen as the message format because it is easily readable by both machines and humans. The Kafka components were developed in Python when implemented in-house and then containerized, otherwise official Docker images were used. Currently, the Kafka cluster consists of a single broker, and the microservices related to the different components run on the same machine where the Kafka cluster is deployed. In the following, details about the producers and processors are provided.

Orchestrator Logs Producer. Orchestrator logs contain information about the history of submitted deployments. To reduce the number of sources from which Kafka retrieves information, we decided to enrich the logs with data previously obtained from the Orchestrator Dashboard database. The Orchestrator sends its logs to the SysLog server, and to make them available to Kafka, we configured the local rsyslog service to forward them to an additional component that listens on a specific port for incoming logs (in syslog format) and forwards them to the *orchestrator-logs* Kafka topic.

Rally Producer. The Rally producer collects Rally test outputs, extracts the most relevant information, and sends it to the Kafka cluster in the *rally* topic. Currently, the component does not interact directly with Rally (although it will in the future) but only with its result output files. The producer is executed every 5 min and checks if new files are present in a specified folder. If new files are found, it imports all data related to the Rally test outcomes and extracts the most relevant information. After this step, it reads all messages present in the *rally* topic. In this way, any information not already present in the topic will be written to it, avoiding duplicate messages. Each message contains information about the provider where the test was executed, the final state of the test (either *finished* or *crashed*), a tag that provides more information about the type of test executed, and finally the timestamp.

Federation Registry Feeder Producer. The Federation Registry Feeder producer collects monitoring information from Cloud providers within the INFN Cloud federation, converts it into a JSON format, and writes messages to the *federation-registry-feeder* topic. Currently, the component executes this procedure periodically every 15 min. For each Cloud provider, it collects information about available projects, and for each project, it gathers data on state, resource usage, quotas, available flavors, and images.

Template Parser Processor. This component reads the user-submitted data, i.e. the TOSCA template and parameters, and validates it. This information is collected from the Orchestrator logs and stored in the *orchestrator-logs* Kafka topic. The validation of the template involves checking its compliance with the YAML format, verifying that the user has provided values for all required parameters, ensuring these values are consistent with the parameter types, and performing other necessary checks. Once validated and enriched with user-provided parameters, the template is written in JSON format to the *validated-template* Kafka topic.

AI-Ranker Inference Processor. This component starts its execution when a new message is read from the *validated-template* topic. It writes a message to the *ai-ranker-inference* Kafka topic with information about the validated template and details of all federated sites that can be used to deploy the service requested by the user. The latter information is retrieved from the *federation-registry-feeder* and *rally* topics. From the *rally* topic, the percentage of failed Rally tests over three periods is collected: one day, one week, and one month. When the inference component of the AI-Ranker service receives this message, it applies AI models to return a sorted list of federated sites that can be used by the Orchestrator for the deployment submission (see Sect. 5.2), which is then written to the *ranked-providers* topic. In this context, the inference component of the AI-Ranker service can be considered as an additional processor.

AI-Ranker Training Processor. The training-data processor collects information about the outcomes of deployment attempts in the federated sites from the *orchestrator-logs* topic, uses it to compute additional metrics, merges it with messages from the *ai-ranker-inference* topic regarding the selected provider(s), and writes the resulting data to the *ai-ranker-training* Kafka topic, which is consumed by the training component of the AI-Ranker. Specifically, for each federated site where the deployment was attempted, it computes the submission timestamp, the time spent to complete the deployment (in seconds), the number of failures, the failure reason (if any), and the cumulative time spent on failed attempts.

5.2 The AI-Ranker Service

During the design phase of the AI-Ranker to be integrated into the PaaS Orchestration system, we evaluated different technologies that could support the various phases of the ML lifecycle. In particular, we focused on Kubeflow [13] and MLflow, analyzing their strengths, differences, and limitations [14].

– **Kubeflow**. A Kubernetes-native platform designed to support end-to-end ML workflows, from data preprocessing and model training to hyperparameter tuning, serving, and monitoring. It provides a scalable and modular architecture where different components, such as Kubeflow Pipelines, Katib for

hyperparameter optimization, and KFServing for model deployment, work together to automate and streamline ML operations. By leveraging Kubernetes, Kubeflow ensures efficient resource allocation and workload distribution, making it particularly suitable for large-scale deployments and production environments with high computational demands. However, its complex setup and steep learning curve pose significant challenges, requiring in-depth knowledge of Kubernetes and additional infrastructure management efforts.
- **MLflow**. A lightweight, open-source platform designed to manage the ML lifecycle with a focus on experiment tracking, model versioning, and deployment. Unlike Kubeflow, MLflow is not tied to Kubernetes and can be deployed on any Cloud or local environment. It consists of four main components: MLflow Tracking (for logging experiments), MLflow Projects (for packaging and reproducing ML code on any platform), MLflow Models (for model versioning and deployment), and MLflow Registry (for managing different model versions). MLflow is particularly effective when used in combination with existing ML training workflows, providing a straightforward and easy-to-use interface for tracking and managing models. However, it lacks native support for complex pipeline orchestration and automated resource management. Additionally, MLflow's model serving capabilities are limited compared to those of Kubernetes-based solutions like KFServing, which offer more advanced scaling and inference optimizations.

Kubeflow and MLflow address different needs within the ML lifecycle: while Kubeflow is a powerful, MLOps framework suitable for large-scale and production-ready deployments, MLflow strengths are simplicity, flexibility, and model management. Since our use case does not involve complex pipelines or large datasets, is not intended to be used directly by large user communities, and aims to be deployable on any infrastructure without requiring a Kubernetes cluster, we chose MLflow as the core technology for our AI-Ranker service. This choice ensures easier integration into the PaaS Orchestration system, greater deployment flexibility, and a solution better suited to the scale and complexity of our needs.

We developed the AI-Ranker in three main components: training service, model registry service, and inference service, all based on the MLflow framework. The training and inference services are written in Python and containerized using Docker, while for the model registry service, we used the official MLflow Docker image. We employed Ofelia [18] as a job scheduler to trigger the training service at predefined intervals. Specifically, the actions performed by the training component are listed below.

- Reading of all messages in the *ai-ranker-training* Kafka topic.
- Preprocessing of the data, including outlier removal, normalization, and feature engineering.
- Training of ML models using the preprocessed data, with the option to perform k-fold cross-validation and select the best model based on performance metrics. Two types of ML models are used: classification models to predict

deployment success or failure, and regression models to predict deployment creation time. Currently, only ML models based on the scikit-learn library [27] are supported.
– Storing the (best) trained ML models in the MLflow model registry, along with relevant metadata, performance metrics, and artifacts (e.g. the scaler).

One of the advantages of MLflow is that it allows logging a wealth of information about training runs on the server, enabling us to track model performance, the hyperparameters used, training time, the features considered, and their importance. This information can be monitored through the MLflow UI, which provides a user-friendly interface for visualizing and comparing different training runs. An example is shown in Fig. 6.

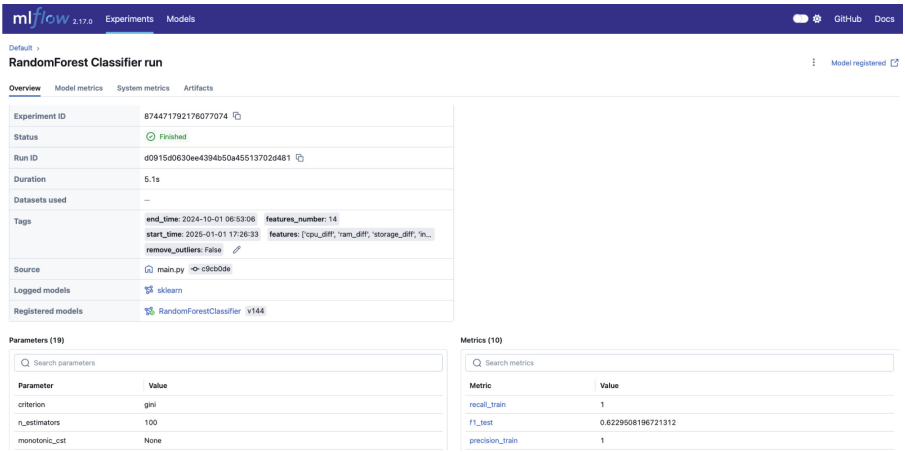

Fig. 6. Partial overview of the information saved in the MLflow registry after a training run.

The inference component of the AI-Ranker listens to the *ai-ranker-inference* Kafka topic, and when a new message is read, it performs the following actions:

– reads the message from the *ai-ranker-inference* Kafka topic;
– preprocesses the data;
– loads the classification and regression models from the MLflow model registry;
– performs inference using the loaded models and the preprocessed data;
– computes the ranking of providers by combining the predictions obtained from the two models, giving priority to those providers that offer the exact flavor(s) requested for the deployment;
– writes the ranked list of providers to the *ranked-providers* Kafka topic.

6 Conclusions and Outlook

In this paper, we presented the ongoing activity aimed at improving the Cloud provider ranking mechanism within the PaaS Orchestration system of the INFN Cloud platform. In Sect. 4, we described an initial study that helped us understand the available data sources and the information that can be extracted, as well as explore the potential use of AI techniques for this use case. The next step was to develop a service that could be integrated into the PaaS Orchestration system, as described in Sect. 5. The proposed solution involved the introduction of a Kafka-based service to collect, aggregate, and process relevant information from various sources, along with the development of the AI-Ranker service responsible for training and inference of ML models. Currently, the developed services are deployed in a pre-production environment, which helps assess the system's performance, determine if improvements are needed, and test their integration within the PaaS Orchestration system. In particular, for the Kafka-based service, the next steps should include implementing service monitoring, encrypting the messages exchanged between components, and expanding the cluster to three nodes to enhance fault tolerance and performance. For the AI-Ranker service, we plan to generate and store plots of the trained models' performance metrics in the MLflow server, integrate additional ML frameworks (such as Keras [12], TensorFlow [28], and PyTorch [22]), and allow users to upload the complete definition of the ML models to be used.

Acknowledgements. The work presented in this paper has been funded by the NextGenerationEU European initiative through the Italian Ministry of University and Research, PNRR Mission 4, Component 2 - TeRABIT: Investment 3.1, Project code IR0000022 - CUP I53C21000370006. The authors have no competing interests to declare that are relevant to the content of this article.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. AI-Ranker. https://github.com/infn-datacloud/ai-ranker. Accessed 04 Apr 2025
2. Ansible roles. https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_reuse_roles.html. Accessed 04 Apr 2025
3. Antonacci, M., Salomoni, D.: Leveraging TOSCA orchestration to enable fully automated cloud-based research environments on federated heterogeneous e-infrastructures. PoS **ISGC&HEPiX2023**, 020 (2023). https://doi.org/10.22323/1.434.0020
4. Apache Kafka. https://kafka.apache.org. Accessed 04 Apr 2025
5. Docker containers. https://www.docker.com/resources/what-container/. Accessed 04 Apr 2025
6. Helm charts. https://helm.sh/docs/topics/charts/. Accessed 04 Apr 2025

7. INDIGO-IAM. https://indigo-iam.github.io/v/v1.11.0/docs/. Accessed 04 Apr 2025
8. INFN Cloud dashboard. https://my.cloud.infn.it. Accessed 04 Apr 2025
9. The INFN Cloud Use Cases Documentation. https://guides.cloud.infn.it/docs/users-guides/en/latest/index.html. Accessed 04 Apr 2025
10. Infrastructure Manager. https://github.com/grycap/im. Accessed 04 Apr 2025
11. JSON Web Tokens. https://auth0.com/docs/secure/tokens/json-web-tokens. Accessed 04 Apr 2025
12. Keras. https://keras.io. Accessed 04 Apr 2025
13. Kubeflow. https://www.kubeflow.org. Accessed 04 Apr 2025
14. Kubeflow vs MLFlow: which one to choose?. https://ubuntu.com/blog/kubeflow-vs-mlflow. Accessed 04 Apr 2025
15. Kubernetes. https://kubernetes.io. Accessed 04 Apr 2025
16. Giommi, L., Debiase, F., Savarese, G., Vino, G., Antonacci, M., Donvito, G., Costantini, A.: AI-based approach for provider selection in the INDIGO PaaS Orchestration System of INFN Cloud. In: Proceedings of the Conference on Computing in High Energy and Nuclear Physics 2024 (2025). Submitted, under review
17. MLflow. https://mlflow.org. Accessed 04 Apr 2025
18. Ofelia - a job scheduler. https://github.com/mcuadros/ofelia. Accessed 04 Apr 2025
19. OpenSearch. https://opensearch.org. Accessed 04 Apr 2025
20. OpenStack. https://www.openstack.org. Accessed 04 Apr 2025
21. Orchestrator. https://github.com/infn-datacloud/orchestrator. Accessed 04 Apr 2025
22. PyTorch. https://pytorch.org. Accessed 04 Apr 2025
23. RabbitMQ. https://www.rabbitmq.com. Accessed 04 Apr 2025
24. Rally. https://rally.readthedocs.io/en/latest/overview/overview.html. Accessed 04 Apr 2025
25. Salomoni, D., et al.: INDIGO-Datacloud: foundations and architectural description of a platform as a service oriented to scientific computing. CoRR arxiv:abs/1603.09536 (2016). http://arxiv.org/abs/1603.09536
26. Savarese, G., Antonacci, M., Giommi, L.: Federation-registry: the renovated configuration management database for dynamic cloud federation. In: International Symposium on Grids & Clouds (ISGC) 2024, p. 21 (2024)
27. scikit-learn. https://scikit-learn.org. Accessed 04 Apr 2025
28. TensorFlow. https://www.tensorflow.org/. Accessed 04 Apr 2025
29. TOSCA Simple Profile in YAML Version 1.1. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/csprd01/TOSCA-Simple-Profile-YAML-v1.1-csprd01.html. Accessed 04 Apr 2025
30. TOSCA templates. https://baltig.infn.it/infn-cloud/tosca-templates. Accessed 04 Apr 2025
31. TOSCA types. https://baltig.infn.it/infn-cloud/tosca-types. Accessed 04 Apr 2025