Federation-registry: the renovated Configuration Management Database for dynamic cloud federation

Giovanni Savarese,[a,*] **Marica Antonacci**[a] **and Luca Giommi**[b]

[a] *INFN-BA,*
Via E. Orabona 4, Bari, Italy
[b] *INFN-CNAF,*
Viale Berti Pichat 6/2, Bologna, Italy
E-mail: giovanni.savarese@ba.infn.it, marica.antonacci@ba.infn.it, luca.giommi@cnaf.infn.it

[*] Speaker

The INDIGO PaaS orchestration system is an open-source middleware designed to seamlessly federate heterogeneous computing environments, including public and private clouds, container platforms, and more. Its primary function lies in orchestrating the deployment of virtual infrastructures, ranging from simple to intricate setups. These virtual infrastructures can implement high-level services, such as JupyterHub, Kubernetes, Spark, and HTCondor clusters, providing users with convenient access and operational control.
At the heart of the orchestration system lies its core component, the Orchestrator, supported by a suite of micro-services. These micro-services play a crucial role in assisting the Orchestrator by facilitating the selection of the optimal provider from the federated environments, based on the specific deployment request.
Within this architecture, a pivotal micro-service is dedicated to implement the information system of the federation. This crucial component records comprehensive details about all the providers, encompassing their characteristics and capabilities. The information stored plays a central role in the matchmaking process between user deployment requests and available providers. Currently, this functionality is implemented by the Configuration Management Database (CMDB) service, which stores and organizes information about resource providers, and the Service Level Agreement Manager (SLAM) which retains SLAs signed by users and resource provider administrators.
For instance, if a deployment request specifies the allocation of one or more Graphics Processing Units (GPUs), the Orchestrator relies on the information system to identify which providers within the federation, for which the user is entitled to allocate resources, offer GPU capabilities.
We have opted to replace the existing services due to the discontinuation of maintenance for the CMDB developed during the INDIGO-DataCloud project, which relies on outdated components. The forthcoming solution, the Federation-registry, is a state-of-the-art web application built on the FastAPI framework. It features a REST API secured by OpenID-Connect/OAuth2 authentication and authorization technologies and policies. This upgrade ensures a more robust and secure foundation for managing federation-related information.
The Federation-registry leverages neo4j, a highly flexible graph database, as opposed to the legacy CouchDB - a non-relational database - for storing and organizing data related to resource providers. Additionally, it adopts S3 object storage to securely store the signed SLA agreements.
This upgrade promises several advantages, including improved data organization, independence from outdated and unmaintained software, adherence to test-driven code practices, enhanced flexibility for accommodating various types of providers, and simplified database structure updates for the incorporation of new provider types. This contribution will outline the architectural decisions and delve into the specifics of the implementation.
The newly implemented Federation-registry service will be integrated into the INFN Cloud platform, which is already exploiting the INDIGO PaaS middleware to provide INFN scientific communities with a portfolio of high-level services supplied on-demand across geographically distributed cloud sites.

International Symposium on Grids and Clouds (ISGC2024)
24 -29 March, 2024
Academia Sinica Computing Centre (ASGC), Institute of Physics, Academia Sinica Taipei, Taiwan

1. Introduction

The availability of computational resources accessible in a simple and transparent way is a critical component for addressing the research challenges of the coming years. Projects such as HL-LHC, Belle II, Einstein Telescope, ePIC@EIC, will drastically increase the production of experimental data in the coming years. Equally demanding are the theoretical computational physics projects which require enormous computing capabilities. To satisfy this increasing need for resources, the leading infrastructure projects of the Italian National Recovery and Resilience Plan (NRRP), such as the National Center for Research in High Performance Computing (HPC), Big Data and Quantum Computing, TeRABIT, DARE and others, are progressing with the deployment of hardware and software resources for research projects and user communities. Alongside these, the INFN [1] is at the forefront of various innovative scientific projects that will be developed within the NRRP. These projects, often of an interdisciplinary nature, bring the INFN closer to research contexts such as in the medical field and represent new challenges, with new requirements to be respected.

All these activities could not be addressed without adequate computing infrastructures, to which the NRRP projects contribute substantially. INFN is engaged in various activities aimed at developing and using a continuum computing and data lake approach, which will allow the best use of the most modern technologies: the power of HPC architectures, the flexibility of Field Programmable Gate Array (FPGA) processors, or low-power ARM processors. In the software field, the effectiveness of the NRRP push can only materialize through the development of an efficient mechanism for accessing resources, and the implementation of a national research data lake for data management. This is the objective of the INFN Datacloud [2] project, with which the INFN aims to create a cloud platform to access resources within the INFN or provided by external providers, through a portfolio of scientific applications already operational and rapidly expanding. Federation of infrastructures, such as OpenStack providers [3] or Kubernetes clusters [4], which is the main focus of this article, is essential to achieve and improve the collaboration between different research institutions.

1.1 INDIGO PaaS Orchestrator

The **INDIGO PaaS Orchestrator** [5, 6] is a Java application providing Representational State Transfer (REST) Application Programming Interface (API) to create high-level services such as single virtual machines, Kubernetes clusters, Jupyter notebooks with persistent data, and more. These services can be deployed on any of the available federated cloud environments, based on the user's groups and the signed Service Level Agreements (SLA). Users can interact with the Orchestrator through the **Orchestrator dashboard** [7, 8]. This service is a Python3 flask [9] application providing a graphical user interface to manage user's *deployment procedures*. The deployment procedure is the action to instantiate one high-level service. The dashboard provides a page with the service catalog, which is a graphical representation of the available service templates, which can be configured to reach the user needs. Additionally, the service catalog can be extended to bear new applications; in fact, new high-level services for new use-cases can be integrated through an *on-boarding* process. Based on the user group's permissions, the list of the shown services can

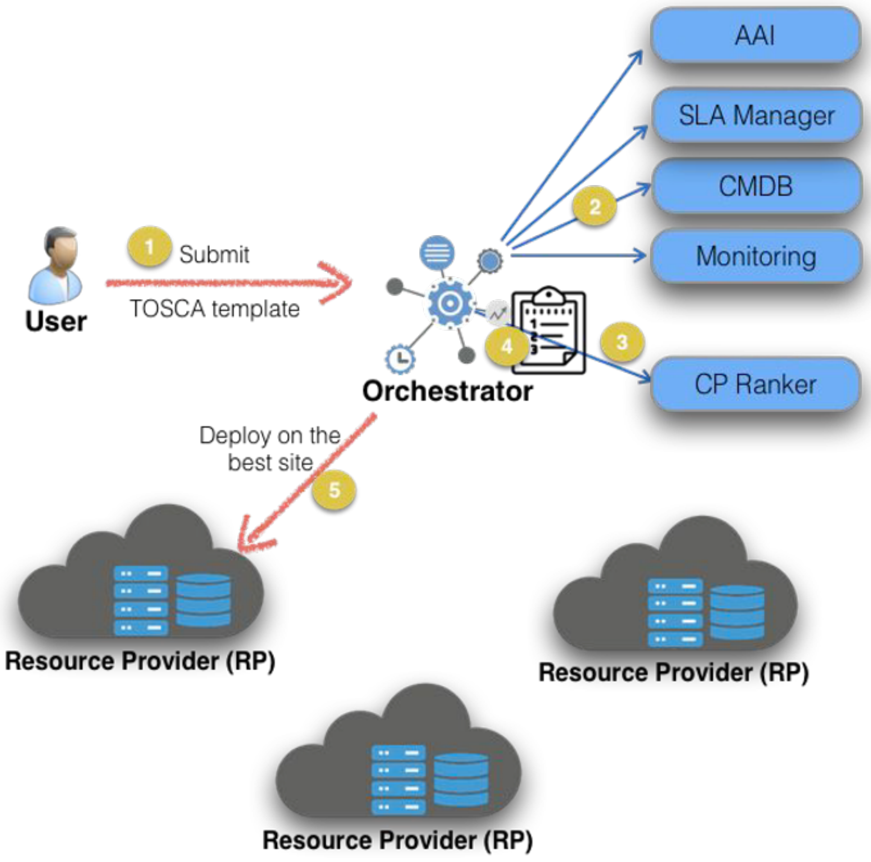

Figure 1: High-level architecture and workflow of the PaaS Orchestration system.

change. To retrieve user groups, the dashboard supports user authentication for any configured identity provider supporting the OpenID-Connect (OIDC) protocol.

To select the best federated provider, the Orchestrator interacts with multiple micro-services and uses TOSCA templates [10, 11] to define the resources to instantiate. Since access to the federated providers resources depends on the user group's authorizations, a user must choose a user group when performing a deployment. Figure 1 describes the deployment workflow.

At first, the Orchestrator retrieves from the **Service Level Agreement Tool** (SLAT) [12] the list of SLAs, and collects from the **Configuration Management Database** (CMDB) [13] the configurations of federated providers. An SLA is a document associating a user group with, in case of OpenStack providers, a federated provider's project; it also specifies the set of available resources for that user group. The providers list is filtered matching the chosen user group with the available SLAs and then matching the SLAs' services with the providers' configurations. Then, the Orchestrator uses this filtered list to retrieve from the **Monitoring** service the associated metrics such as deployment time, deployments success rate, resources usage and more. These values will be used by the **Cloud Provider Ranker** (CPR) to sort the list of the providers. Finally, the Orchestrator will use the best provider's configuration to fill the TOSCA template fields, and send it to the **Infrastructure Manager** (IM) [14], which is in charge of instantiating the needed elements on the target provider. If an attempt fails, the procedure proceeds with the next best provider in the sorted list until it succeeds or the list is empty.

All mentioned services are deployed through docker containers and have their own image on DockerHub. Currently, some of the micro-services involved in this procedure are no longer maintained, or uses deprecated libraries, or uses technologies no more suited for our use cases. Due to these reasons, the upgrade of these micro-services is a key element in the DataCloud project. The adopted services replacement strategy foresees the replacement of the existing services with new ones and the extension of the Orchestrator procedures to support both the legacy services or the new ones.

1.2 Configuration Management Database (CMDB)

The Configuration Management Database (CMDB) is a service written in Java 8, managing federated providers and corresponding services configurations. This service provides integration for **INDIGO-IAM** [15] and all requests must pass a valid token in the authorization header. Non-authenticated users cannot read data. Data are stored in an Apache CouchDB [16] database. This database provides a REST API expecting and generating JavaScript Object Notation (JSON) compliant data. Additionally, it does not have a schema definition, allowing for entities flexibility; as a matter of fact, it uses a document storage model to archive data. The drawback of document storage models is that they lose the advantages of using relationships.

This service is no longer maintained, the usage of CouchDB does not suite our needs and, in the vision of using the Python language, we can freely choose a better database management system exploiting information derived from the connections between providers and identity providers, or projects and available resources, or user groups and matching projects and more.

1.3 Cloud Info Provider (CIP)

The Cloud Info Provider (CIP) [17] is a Python script in charge of periodically populating the CDMB, using the CMDB's API, with the data stored in a set of YAML [18] files. The YAML files are manually generated and contain information about the federated projects, default values and granted resources for a specific provider to federate. Nowadays, not all stored details are relevant; indeed, some parameters were used for backwards compatibility with the grid paradigm, which will be no longer supported. Moreover, information about resource grants may not be up to date since the YAML file generation is not performed periodically. The population procedure is periodically executed starting a docker container running the script. One of the major vulnerabilities of this script is that it is written in Python2.7 and it is no longer maintained.

1.4 Service Level Agreement Tool (SLAT)

The Service Level Agreement Tool (SLAT) is a Python3 flask application providing a REST API to retrieve the quotas assigned to each user group on specific providers' services. It provides a graphical user interface through jinja2 [19] templates and, as the CMDB, it integrates authentication and authorization logic providing support for INDIGO-IAM. Data are stored in a MySQL [20] database through the well known SLQAlchemy [21] Python library. The data stored on this service are highly coupled, and sometimes duplicated, with the ones stored in the CMDB; so, the refactoring of the CMDB consequently involved the refactoring of the SLAT.

2. Federation-registry

The problems described in the previous sections highlight the need to substitute the CMDB, the CIP and consequently the SLAT with newer services. The **Federation-registry** is a new service which will replace both the CMDB and the SLAT. This service will be the single source of truth for the configurations of federated providers and grants to user groups for resource use. The Federation-registry is written in Python3 and uses the well known FastAPI [22] framework, and communicates with a neo4j [23] database. This service exposes a REST API, so, a deployment procedure can retrieve the providers' configurations, available for a specific user group, with a single HTTP GET request [24].

The seek of code simplicity and language ease of learning led the code migration from Java to Python. We evaluated the frequency of new deployments to be lower than ten instances every minute, and the response time of an HTTP request on the Federation-registry is negligible with respect to the overall deployment procedure. In fact, we are comparing operations spanning from hundreds of ms to tens of seconds with procedures lasting tens of minutes. For these reasons, the usage of Python does not introduce a significant delay and does not degrade the application's performances. As a matter of fact, the 80% of the deployment procedure time is occupied by the IM to instantiate and configure the resources in the target provider. FastAPI has been chosen for its performance on par with NodeJS and Go, its simplicity and because it minimizes code duplication; the online documentation is huge and rich of examples, with a large community supporting the code development. Moreover, this library implements the *open* standards for APIs, supporting OpenAPI and JSON schemas, and provides an automatic API documentation with the possibility to test query (with and without authentication).

Another relevant choice was to adopt neo4j, a graph database model, to store data. Graph databases bring the advantages of not needing a schema definition, allowing for data flexibility. This is an essential feature; in fact, the application database has been designed to include other kinds of providers and services and to not limit entities' attributes since new information can be integrated to implement future features and because federated providers can have really heterogeneous data. From this point of view, neo4j has a lot in common with CouchDB, but the main difference with document storage models is that graph models implement relationships. This way, the service can exploit the relationships between these data, and the data contained in the relationship, to perform complex query at database level, and to filter data. One drawback of using neo4j is that it provides limited database constraints, which must be implemented in the Python code.

2.1 Application design

The database stores the federated providers' current configuration. Database entities match the real items involved in the federation logic, such as identity providers, user groups, SLAs, resource providers, flavors, images and more. Figure 2 shows the database entities and their relationships.

Multiple **resource providers** with the same *name* but different *type* can exist; for example a site providing a Kubernetes cluster and an OpenStack infrastructure can assign the same name to them, but they will have different types. A resource provider can support one or more **regions**. For the sake of standardisation, a computing environment not supporting regions will be associated with a *fake* region. Each region can optionally be associated with a single **geographical location**.

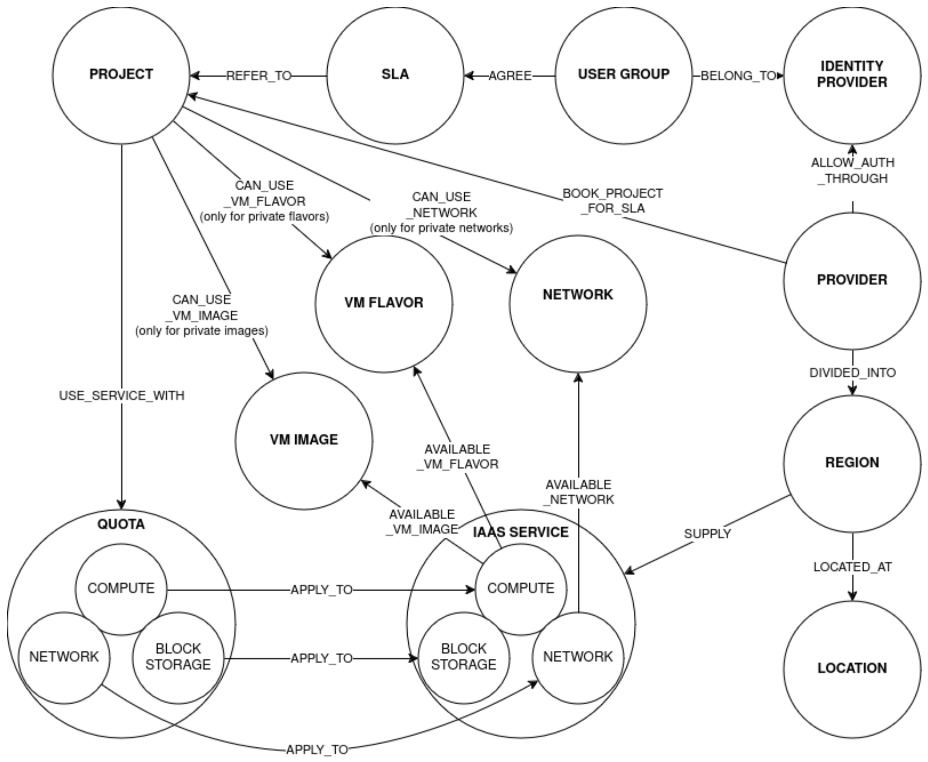

Figure 2: Schema of the Federation-registry's entities and relationships.

In fact, multiple regions of the same provider can belong to the same location or, for geographically distributed providers, each region can belong to a different location. Multiple providers can be installed in the same geographical location.

A provider can support authentication via a set of **identity providers**. Each identity provider is reachable at a specific endpoint and defines a specific group claim, which is the key, in the token, storing the user group name. An identity provider hosts multiple **user groups**. The *communication protocol* and the *identity provider name*, requested by the resource provider when authenticating a user through a specific identity provider, depend on the provider configuration.

Each provider can supply a set of **Infrastructure as a Service (IaaS) services**. Each service has an endpoint which must be unique in the database and, based on the service type, its own additional attributes and relationships. Here the list of the service types and their main characteristics:

- **Block-storage**: manages volume creation and usage.
- **Compute**: manages VM creation. Each project can use a set of flavors and images.
- **Identity**: manages user authentication when performing requests to the provider.
- **Network**: manages networks creation and usage. Each project can exploit a set of networks.

Block-storage, compute and network services impose to authorized **projects** a set of resource limitations (or **quotas**) such as maximum disk size, maximum number of virtual Central Processing Units (CPUs) and more. Public resources (favors, images) can be used by any project; private or shared ones can be accessed only by authorized projects. Private networks cannot be shared between projects; each project must have its own set of private networks. Each resource is uniquely identified by its *uuid* and *name*.

Resource providers allow user groups to access IaaS services' resources through projects. Each user group grants access to a provider through a single project. A user group can be associated with multiple projects each belonging to a different provider. The association between a user group and a project is defined in an **SLA**. An SLA has a *start date* and an *end date*, and it is associated with a physical document defining the resource limitations for a specific project of a provider; it can define different quotas for each region the project can access. In the same SLA, it is possible to associate with the same user group multiple projects, with the constraint that each project must belong to a different provider. Additional quotas to apply to single users, called *per-user* quotas, can be defined and will be checked by the Orchestrator. For easy user groups and projects associations through SLAs, we impose a limitation: users can access to a specific project only through a single user group (the one referenced by the SLA). For example, consider a physicist having access to user group G1 on identity provider IDP1, and user group G2 on identity provider IDP2. The user group G1 has been granted access to a set of SLAs, whereas G2 agreed to another subset of SLAs. The user cannot access to a project defined in an SLA associated to G1 using the user group G2.

2.2 Implementation choices

The usage of the neomodel [25] library was extremely handful to create the Python objects corresponding to the database entities and to perform database queries, similarly to what SQLAlchemy does for relational databases. To differentiate between different types of quotas and services, the application exploits Python inheritance logic which is mapped in the neo4j feature to support multiple labeled nodes. This choice allows to take direct advantage of the cypher language [26] to validate entity relationships in queries and avoid code duplication. Neomodel provides a decorator ensuring that the code is run within the same database transaction, ensuring data consistency and atomic operations. This is extremely useful in our case; as a matter of fact, we can wrap our REST API endpoints and execute them within the same database transaction. A bug was present in this feature of neomodel when used with multiple decorators; we fixed its behaviour and we opened a pull-request on the main branch [27, 28]; currently, we are using a fork of the neomodel repository.

Users can access the database data only through the REST APIs. This strategy is essential to enable checks not available at database level, to limit non-authenticated or not-authorized user access, and to secure the database from queries injection attacks. The available endpoints match the available node types; for example, if a user wants to list all projects, the application provides a dedicated endpoint. The application supports API versioning for each endpoint, and FastAPI has been configured to provide a separate documentation page for each API version.

All endpoint categories support multiple HTTP methods:

- GET method to retrieve the lists of items of the same kind (projects, images, flavors, providers, services and more). This endpoint type can: paginate items, limit the maximum number of returned values, skip a fixed amount of returned values, sort items by any attribute both in ascending (default) and descending order, view linked entities, and filter items specifying one or multiple attribute values.

- GET method to retrieve a single entity with or without linked items.
- PUT and PATCH methods to update a specific entity. The PUT methods can override entity's connections. Both methods always check that the new received data and the database ones differ. If there are no differences, the endpoint returns a message with status code **304 - Not modified** and no content.
- DELETE method to delete a specific entity.

PUT, PATCH and DELETE methods require the item *uid* attribute and can be executed only by authorized users. If the target entity does not exist in the database, the endpoint raises a 404 NOT FOUND error. All methods execute a preliminary validation of query and body attributes, through the usage of the pydantic [29] library and FastAPI dependencies. If these preliminary checks fail, the endpoints raise an error with status code **422 - Unprocessable entity** or **400 - Bad request**. The *resource providers* endpoint is the only one supporting a POST method. The standard procedure to add a new provider to the database is through the population script. In fact, the script creates or connects the related entities (regions, services and more).

2.3 Authentication and Authorization

Read-only endpoints can be accessed by both authenticated and non-authenticated users. The main difference is that non-authenticated users can read only a subset of the stored data, whereas authenticated ones can read a shrunk version of the data or the complete one. This design choice allows not only the Orchestrator but any other service to retrieve data from this service and allows to reduce the data payload when needed. For security reasons, only authorized users can perform operations that can change the database. Since this service has been designed to be used not only by the DataCloud project, the list of the authorized users and the list of the trusted identity providers are configurable. POST, PUT, PATCH and DELETE methods return an error message with status code **403 - Forbidden** if the user does not have write access rights.

One of the great advantages of this application is that it does not implement its own database to store users' data, instead it uses the flaat [30] library to validate the received token, against one of the configured identity providers. This way, the application can use external identity providers to authenticate users, whereas it locally implements the authorization logic based on the user's email. An interesting note is that, to allow authenticated and non-authenticated users to access the same endpoint, we opened a pull-request and fixed a bug in the flaat library [31].

3. Federation-registry-feeder

The Federation-registry-feeder is a Python3 script in charge of periodically reading directly from the federated providers their configurations and update the Federation-registry with always up-to-date data. This script is meant to ease data insertion and data update into the Federation-registry. The script emulates what the legacy CIP previously did to populate the CMDB: it reads a set of YAML files with the providers' configurations to populate the database. The main differences are that in the YAML files there are only the configurations which cannot be retrieved directly from the federated providers whereas it reads the current state of the resources from the federated providers and standardizes those data. In the following sections we describe the main steps of this population script.

3.1 YAML files

At first, the script loads the list of providers it has to contact from a list of YAML files. In each YAML file there are the essential information used to identify the provider and the configuration parameters not retrievable directly from those providers. Indeed, some details are known only to site admins, or are verbally agreed, or depend on the database architecture and must be configured manually. Usually we have a YAML file for each federated site, but we can also have a single YAML file with all the federated providers divided by category (such as OpenStack, Kubernetes and more). Each category expects a list of items; the items' name must be unique in the category list.

For each resource provider, we must specify the authentication url which the script will use to establish the connection. Additionally, we can add the list of the emails to contact to receive support for that provider, the list of tags to use to filter images or networks when retrieving those entities from an OpenStack instance and other details. Each resource provider can have one or multiple regions, each uniquely identified by its name. Optionally, for each region, it is possible to define a geographical location. A resource provider can trust one or multiple identity providers. We can choose to federate all of them or just a subset; anyway, the federated identity providers must also be listed in a separate section called *trusted idps*. In this section, each identity provider is uniquely identified by the issuer attribute. The same identity provider can be trusted by multiple resource providers and will appear in its *identity providers* attribute. The *identity providers* section and *trusted idps* one store different information; the first stores information related to that specific identity provider which depends on the resource provider (such as the communication protocol and the identity provider name); the latter stores the list of the user groups having at least one SLA pointing to one of the listed resource providers' projects. Each SLA has a start date, an end date and points to the uuid of the PDF document with the agreement's details (which also works as the unique identifier for the SLA).

Each resource provider can have one or multiple projects uniquely identified by their id and each project points to exactly one SLA defined in the *trusted idps* section. For each project we can specify a set of attributes which usually cannot be retrieved from the resource provider or that a specific version of that infrastructure instance does not make it available. For example, when a single project has access to multiple private/public networks, it is possible to specify the project's default private/public network, or the limitations to apply per single user to specific resources usage, which will override the group limitations. In some cases, a project can access multiple regions and sometimes, the project default values or per-user limitations can change based on the accessed region. For flexibility sake, it is possible to add, for each project, the section *per region props*. This section expects the target region's name and can receive quite all the project attributes. The attributes in this section will override the ones defined at project level (only when accessing to that region).

3.2 Token retrieval

To connect and read information from the providers, the script needs to authenticate itself. Since the same identity provider can be used by multiple resource providers, when loading from the YAML files the list of the identity providers, for each of them, the script retrieves the access token corresponding to a *service* user configured to have read access to all the federated projects. A useful tool to do so is **oidc-agent** [32]. This tool, pre-configured to have an account for each identity provider, with specific scopes grant, associated with the *service* user, can quickly return the requested access token. Having an account configured in oidc-agent for each identity provider, when loading the list of the allowed issuers we can immediately retrieve the access token for each of them and use it multiple times to connect to the providers.

3.3 Providers connection

Once the script has loaded all the YAML files, it generates a separated thread for each combination of region and project belonging to each provider. In case of OpenStack instances, for each thread, the script opens a connection with the target provider, specifying the target region, the project's id and the access token valid for that provider. With this strategy, we can retrieve in parallel the available resources of a target project from each accessible region. For forbidden combinations, the list of the retrieved resources will be empty. For each opened connection, the script retrieves: the project details, the quotas defined for the configured services, the list of the available flavors, images and networks with their details. For each item, the script maps the retrieved data into standardized pydantic objects and merge them with the ones received through the YAML file. These objects are the same pydantic classes that the Federation-registry endpoints expect as data inputs. If the data retrieval operation fails at some point due to a connection failure, a non-authorized operation, a missing endpoint or something else, the thread dies and logs an error but the overall procedure continues.

3.4 Database update

Finally, the script populates the Federation-registry with the new data. At first, it retrieves the list of the currently federated providers from the Federation-registry. Then, for each provider to federate, if the provider has not been already federated, the script executes a POST request to the service; otherwise, it performs a PUT request. If the new data, matches the current data, the PUT operation is skipped. PUT requests should be executed only by this script since they are destructive; for example, if a user group has no more access to a provider, the connection with the corresponding project and the related quotas will be deleted. Finally, once the script has iterated on all the providers to federate, if there are providers retrieved from the Federation-registry which do not match any of the new providers, to keep data consistent, they will be deleted from the database.

4. Development tools

In this section, we briefly discuss the relevant tools we used to develop the project.

Poetry [33] is a tool devoted to help developers to use the correct version of each package and resolve sub-dependencies. This tool is helpful to reproduce the same virtual environment on multiple machines easing code portability, reproducibility and development. Another critical point is the code formatting and linting. Ruff [34] is a tool configurable to format code, verify linting and more. It can sort the import section, check the document style, check for syntax bad practice and more. Instructing pre-commit [35] to run ruff checks and format operations before every commit ensures to commit only well-formatted code.

Service maintenance is always a key point to make a good service. Knowing this, for new software development, the community is moving toward using test-driven models. For Python, a well known library to develop unit and functional tests is pytest [36]. During the development of this service, we developed ad hoc tests with this library to validate pydantic schemas, neomodel classes, create-update-read-delete (CRUD) procedures, HTTP REST endpoints, authentication and authorization logic. Pytest supports a lot of extensions to achieve multiple features. An interesting extension is pytest-cov [37] which run tests and calculates code coverage. Pytest-mock [38] was another useful extension to mock calls to external libraries and services. For example, to test authentication procedures, this library can emulate the responses from a generic identity provider. Then, a helpful extension, was the pytest-cases [39] one. This library is useful to repeat multiple tests with different parameters avoiding code repetition and improving code readability.

Following the Continuous Integration (CI) and Continuous Development (CD) strategies we developed a pipeline exploiting github actions to test the code, calculate its coverage and upload it on SonarCloud [40]. SonarCloud is meant to analyze code, checking for security vulnerabilities, code smells and coverage visualization. This service has a dedicated extension on Visual Studio Code (VSCode) [41] to perform real-time code analysis. In the vision of having a reliable service, able to load balance users' requests, we foresee to deploy these services using a Kubernetes cluster providing self-healing, service high-availability and more. In fact, as previously done with the CMDB, CIP and SLAT, each service has its own docker image. To automate the image creation, we implemented a pipeline using github actions and triggered on new releases or when executing pull requests into the main branch. The pipeline uploads the new image to DockerHub. The same pipeline is reused, with the correct parameters, in both the Federation-registry and the Federation-registry-feeder repositories.

Finally, since a lot of developers use VSCode, for both projects there is the possibility to use devcontainers [42]. The main advantage of using this feature is that it starts all the needed services to build a running instance of the target service. For example, the devcontainer of the Federation-registry-feeder starts a container with the a neo4j database instance, a container with the oidc-agent tool, a container with a running instance of the Federation-registry (linked to the neo4j database) and a container to run and develop the Federation-registry-feeder.

5. Conclusions and future upgrades

Currently, we have deployed a test instance, to verify and analyze the service reliability, to find bugs and to test the migration of the existing configurations. The migration test is essential to verify that all the features provided by the CIP service are still available in the new service. The test instance was proved to be reliable and fast when querying and adding data: adding six providers, each supporting one identity provider, with at most two regions each and with a total of 40 projects, generated a total of 1150 nodes (considering also quotas, SLAs, resources, and all related entities). The Federation-registry-feeder execution took about 64 seconds to connect to those providers (which are the most time expending operations, although parallelized), to retrieve data and to update the database. The results obtained from these tests are promising and prove that this service will be useful not only to the INDIGO PaaS Orchestrator but to any other middleware service based on dynamic cloud federation.

Further steps are: the integration of new features, such as support for different volume types, the integration of new providers kinds, and the improvement of the filtering options on relevant endpoints based on user and service needs. At the current time, the service supports integration for OpenStack instances, but we have designed the database flexible enough to integrate Kubernetes configurations as well.

References

[1] INFN Home Page, https://home.infn.it/en/, (last access on 2024-02-20).

[2] INFN Cloud Home Page, https://www.cloud.infn.it/, (last access on 2024-02-20).

[3] Openstack Home Page, https://www.openstack.org/, (last access on 2024-02-20).

[4] Kubernetes Home Page, https://kubernetes.io/, (last access on 2024-02-20).

[5] D. Salomoni, I. Campos, L. Gaido et al. *INDIGO-DataCloud: a Platform to Facilitate Seamless Access to E-Infrastructures*. *J Grid Computing* **16**, 381–408 (2018).

[6] Orchestrator source code, https://github.com/indigo-paas/orchestrator, (last access on 2024-02-20).

[7] M. Antonacci, et al. *A Dynamic and Extensible Web Portal Enabling the Deployment of Scientific Virtual Computational Environments on Hybrid E-infrastructures*. *Zenodo*, (2023).

[8] Orchestrator-dashboard source code, https://github.com/indigo-paas/orchestrator-dashboard, (last access on 2024-02-20).

[9] Flask Home Page, https://flask.palletsprojects.com/en/3.0.x/, (last access on 2024-02-20).

[10] OASIS TOSCA Standard, https://www.oasis-open.org/committees/tosca/faq.php, (last access on 2024-02-20).

[11] M. Caballer, G. Donvito, G. Moltó, R. Rocha and M. Velten, *TOSCA-based orchestration of complex clusters at the IaaS level.* . *Phys. Conf. Ser.* **898** 082036 (2017)

[12] SLAT source code, https://github.com/maricaantonacci/slat, (last access on 2024-02-20).

[13] CMDB source code, https://github.com/indigo-dc/cmdb, (last access on 2024-02-20).

[14] M. Caballer, I. Blanquer, G. Moltó et al. *Dynamic Management of Virtual Infrastructures*. *J Grid Computing* **13**, 53–70 (2015).

[15] INDIGO-IAM source code, https://github.com/indigo-iam, (last access on 2024-02-20).

[16] CouchDb Home Page, https://docs.couchdb.org/en/stable/index.html, (last access on 2024-02-20).

[17] CIP source code, https://github.com/indigo-dc/cloud-info-provider-deep, (last access on 2024-02-20).

[18] YAML Home Page, https://yaml.org/spec/1.2.2/, (last access on 2024-02-20).

[19] Jinja2 Home Page, https://pypi.org/project/Jinja2/, (last access on 2024-02-20).

[20] MySQL Home Page, https://www.mysql.com/en/, (last access on 2024-02-20).

[21] SQLAlchemy Home Page, https://www.sqlalchemy.org/, (last access on 2024-02-20).

[22] FastAPI Home Page, https://fastapi.tiangolo.com/, (last access on 2024-02-20).

[23] Neo4j Home Page, https://neo4j.com/, (last access on 2024-02-20).

[24] HTTP methods, https://blog.postman.com/what-are-http-methods/, (last access on 2024-02-20).

[25] Neomodel Home Page, https://neomodel.readthedocs.io/en/latest/, (last access on 2024-02-20).

[26] Cypher language, https://neo4j.com/docs/cypher-manual/current/introduction/, (last access on 2024-02-20).

[27] Neomodel source code, https://github.com/neo4j-contrib/neomodel/, (last access on 2024-02-20).

[28] Pull-Request on Neomodel source code https://github.com/neo4j-contrib/neomodel/pull/777, (last access on 2024-02-20).

[29] Pydantic Home Page, https://docs.pydantic.dev/1.10/, (last access on 2024-02-20).

[30] Flaat source code, https://github.com/indigo-dc/flaat, (last access on 2024-02-20).

[31] Pull-Request on Flaat source code, https://github.com/indigo-dc/flaat/pull/75, (last access on 2024-02-20).

[32] OIDC Agent Home Page, https://indigo-dc.gitbook.io/oidc-agent/, (last access on 2024-02-20).

[33] Poetry Home Page, https://python-poetry.org/, (last access on 2024-02-20).

[34] Ruff Home Page, https://docs.astral.sh/ruff/, (last access on 2024-02-20).

[35] Pre-commit Home Page https://pre-commit.com/, (last access on 2024-02-20).

[36] Pytest Home Page, https://docs.pytest.org/en/8.0.x/, (last access on 2024-02-20).

[37] Pytest-cov documentation, https://pytest-cov.readthedocs.io/en/latest/, (last access on 2024-02-20).

[38] Pytest-mock documentation, https://pytest-mock.readthedocs.io/en/latest/index.html, (last access on 2024-02-20).

[39] Pytest-cases documentation, https://smarie.github.io/python-pytest-cases/, (last access on 2024-02-20).

[40] SonarCloud Home Page, https://www.sonarsource.com/products/sonarcloud/, (last access on 2024-02-20).

[41] Visual Studio Code Home Page, https://code.visualstudio.com/, (last access on 2024-02-20).

[42] Devcontainers documentation, https://code.visualstudio.com/docs/devcontainers/create-dev-container, (last access on 2024-02-20).